



Axivion Bauhaus Suite Stylecheck Rules

6.9.13

Axivion GmbH

support@axivion.com

Table of Contents

Axivion Bauhaus Suite Stylecheck Documentation	51
Common Rule Configuration	51
Global Configuration	51
Rules in Group AutosarC++17_03	53
AutosarC++17_03-A0.1.1	53
AutosarC++17_03-A0.1.2	54
AutosarC++17_03-A0.1.3	54
AutosarC++17_03-A0.4.2	55
AutosarC++17_03-A1.1.1	55
AutosarC++17_03-A2.2.1	57
AutosarC++17_03-A2.5.1	58
AutosarC++17_03-A2.6.1	58
AutosarC++17_03-A2.8.1	58
AutosarC++17_03-A2.8.2	59
AutosarC++17_03-A2.8.3	59
AutosarC++17_03-A2.8.4	60
AutosarC++17_03-A2.9.1	60
AutosarC++17_03-A2.11.1	61
AutosarC++17_03-A2.11.2	62
AutosarC++17_03-A2.11.3	63
AutosarC++17_03-A2.11.4	63
AutosarC++17_03-A2.11.5	64
AutosarC++17_03-A2.14.1	64
AutosarC++17_03-A2.14.2	64
AutosarC++17_03-A2.14.3	65
AutosarC++17_03-A3.1.1	65
AutosarC++17_03-A3.1.2	66
AutosarC++17_03-A3.1.3	66
AutosarC++17_03-A3.1.4	67
AutosarC++17_03-A3.3.1	67
AutosarC++17_03-A3.3.2	68
AutosarC++17_03-A3.9.1	68
AutosarC++17_03-A4.5.1	68
AutosarC++17_03-A4.7.1	69
AutosarC++17_03-A4.10.1	69
AutosarC++17_03-A5.0.1	70
AutosarC++17_03-A5.0.2	70
AutosarC++17_03-A5.0.3	71
AutosarC++17_03-A5.1.1	71
AutosarC++17_03-A5.1.2	72
AutosarC++17_03-A5.1.5	72
AutosarC++17_03-A5.1.7	73
AutosarC++17_03-A5.1.8	73
AutosarC++17_03-A5.2.1	74
AutosarC++17_03-A5.2.2	74
AutosarC++17_03-A5.2.3	74
AutosarC++17_03-A5.2.4	75
AutosarC++17_03-A5.3.1	75
AutosarC++17_03-A5.5.1	76
AutosarC++17_03-A5.10.1	76

AutosarC++17_03-A5.16.1	76
AutosarC++17_03-A6.4.1	77
AutosarC++17_03-A6.5.1	77
AutosarC++17_03-A6.5.2	78
AutosarC++17_03-A6.6.1	78
AutosarC++17_03-A7.1.1	78
AutosarC++17_03-A7.1.3	79
AutosarC++17_03-A7.1.4	79
AutosarC++17_03-A7.1.5	80
AutosarC++17_03-A7.1.6	80
AutosarC++17_03-A7.1.7	81
AutosarC++17_03-A7.2.1	81
AutosarC++17_03-A7.2.2	81
AutosarC++17_03-A7.2.3	82
AutosarC++17_03-A7.2.4	82
AutosarC++17_03-A7.4.1	83
AutosarC++17_03-A7.5.1	83
AutosarC++17_03-A7.5.2	83
AutosarC++17_03-A8.2.1	84
AutosarC++17_03-A8.4.1	84
AutosarC++17_03-A8.4.2	84
AutosarC++17_03-A8.5.1	85
AutosarC++17_03-A8.5.2	85
AutosarC++17_03-A8.5.3	86
AutosarC++17_03-A8.5.4	86
AutosarC++17_03-A9.6.1	86
AutosarC++17_03-A10.1.1	87
AutosarC++17_03-A10.2.1	87
AutosarC++17_03-A10.3.1	87
AutosarC++17_03-A10.3.2	88
AutosarC++17_03-A10.3.3	88
AutosarC++17_03-A10.3.5	89
AutosarC++17_03-A11.0.1	89
AutosarC++17_03-A11.0.2	89
AutosarC++17_03-A11.3.1	90
AutosarC++17_03-A12.0.1	90
AutosarC++17_03-A12.1.1	91
AutosarC++17_03-A12.1.2	91
AutosarC++17_03-A12.1.3	92
AutosarC++17_03-A12.1.4	92
AutosarC++17_03-A12.4.1	92
AutosarC++17_03-A12.4.2	93
AutosarC++17_03-A12.6.1	93
AutosarC++17_03-A12.8.1	94
AutosarC++17_03-A12.8.2	94
AutosarC++17_03-A12.8.3	94
AutosarC++17_03-A12.8.4	95
AutosarC++17_03-A12.8.6	95
AutosarC++17_03-A12.8.7	96
AutosarC++17_03-A13.1.1	96
AutosarC++17_03-A13.1.2	96
AutosarC++17_03-A13.1.3	97
AutosarC++17_03-A13.2.1	97

AutosarC++17_03-A13.2.2	98
AutosarC++17_03-A13.2.3	98
AutosarC++17_03-A13.3.1	98
AutosarC++17_03-A13.5.1	99
AutosarC++17_03-A13.6.1	99
AutosarC++17_03-A15.1.1	99
AutosarC++17_03-A15.1.2	100
AutosarC++17_03-A15.1.3	100
AutosarC++17_03-A15.2.1	100
AutosarC++17_03-A15.3.1	101
AutosarC++17_03-A15.3.3	101
AutosarC++17_03-A15.3.4	102
AutosarC++17_03-A15.3.5	102
AutosarC++17_03-A15.4.1	103
AutosarC++17_03-A15.4.2	103
AutosarC++17_03-A15.4.3	104
AutosarC++17_03-A15.4.4	104
AutosarC++17_03-A15.4.5	105
AutosarC++17_03-A15.4.6	105
AutosarC++17_03-A15.5.1	106
AutosarC++17_03-A15.5.2	106
AutosarC++17_03-A15.5.3	107
AutosarC++17_03-A16.0.1	108
AutosarC++17_03-A16.2.1	109
AutosarC++17_03-A16.6.1	109
AutosarC++17_03-A16.7.1	110
AutosarC++17_03-A17.0.1	110
AutosarC++17_03-A18.0.1	111
AutosarC++17_03-A18.0.2	111
AutosarC++17_03-A18.0.3	111
AutosarC++17_03-A18.1.1	112
AutosarC++17_03-A18.1.2	112
AutosarC++17_03-A18.1.3	113
AutosarC++17_03-A18.1.4	113
AutosarC++17_03-A18.1.5	113
AutosarC++17_03-A18.5.1	114
AutosarC++17_03-A18.5.2	114
AutosarC++17_03-A18.5.3	114
AutosarC++17_03-A18.5.4	115
AutosarC++17_03-A18.9.1	115
AutosarC++17_03-A18.9.2	116
AutosarC++17_03-A18.9.3	116
AutosarC++17_03-A23.0.1	117
AutosarC++17_03-M0.1.1	117
AutosarC++17_03-M0.1.2	117
AutosarC++17_03-M0.1.3	118
AutosarC++17_03-M0.1.4	119
AutosarC++17_03-M0.1.5	119
AutosarC++17_03-M0.1.8	120
AutosarC++17_03-M0.1.9	120
AutosarC++17_03-M0.1.10	122
AutosarC++17_03-M0.2.1	123
AutosarC++17_03-M0.3.1	123

AutosarC++17_03-M0.3.2	126
AutosarC++17_03-M0.4.2	126
AutosarC++17_03-M2.10.1	127
AutosarC++17_03-M2.10.3	127
AutosarC++17_03-M2.10.6	128
AutosarC++17_03-M2.13.2	128
AutosarC++17_03-M2.13.3	128
AutosarC++17_03-M2.13.4	129
AutosarC++17_03-M3.1.2	129
AutosarC++17_03-M3.2.1	129
AutosarC++17_03-M3.2.2	130
AutosarC++17_03-M3.2.3	130
AutosarC++17_03-M3.2.4	131
AutosarC++17_03-M3.3.2	132
AutosarC++17_03-M3.4.1	132
AutosarC++17_03-M3.9.1	133
AutosarC++17_03-M3.9.3	133
AutosarC++17_03-M4.5.1	134
AutosarC++17_03-M4.5.3	134
AutosarC++17_03-M4.10.1	135
AutosarC++17_03-M4.10.2	135
AutosarC++17_03-M5.0.2	135
AutosarC++17_03-M5.0.3	136
AutosarC++17_03-M5.0.4	137
AutosarC++17_03-M5.0.5	137
AutosarC++17_03-M5.0.6	138
AutosarC++17_03-M5.0.7	139
AutosarC++17_03-M5.0.8	139
AutosarC++17_03-M5.0.9	140
AutosarC++17_03-M5.0.10	141
AutosarC++17_03-M5.0.11	141
AutosarC++17_03-M5.0.12	141
AutosarC++17_03-M5.0.14	142
AutosarC++17_03-M5.0.15	142
AutosarC++17_03-M5.0.16	142
AutosarC++17_03-M5.0.17	143
AutosarC++17_03-M5.0.18	143
AutosarC++17_03-M5.0.20	144
AutosarC++17_03-M5.0.21	144
AutosarC++17_03-M5.2.1	144
AutosarC++17_03-M5.2.2	145
AutosarC++17_03-M5.2.3	145
AutosarC++17_03-M5.2.6	145
AutosarC++17_03-M5.2.8	146
AutosarC++17_03-M5.2.9	147
AutosarC++17_03-M5.2.10	147
AutosarC++17_03-M5.2.11	148
AutosarC++17_03-M5.2.12	148
AutosarC++17_03-M5.3.1	148
AutosarC++17_03-M5.3.2	149
AutosarC++17_03-M5.3.3	149
AutosarC++17_03-M5.3.4	149
AutosarC++17_03-M5.8.1	150

AutosarC++17_03-M5.14.1	150
AutosarC++17_03-M5.17.1	151
AutosarC++17_03-M5.18.1	151
AutosarC++17_03-M5.19.1	151
AutosarC++17_03-M6.2.1	152
AutosarC++17_03-M6.2.2	152
AutosarC++17_03-M6.2.3	152
AutosarC++17_03-M6.3.1	153
AutosarC++17_03-M6.4.1	153
AutosarC++17_03-M6.4.2	153
AutosarC++17_03-M6.4.3	154
AutosarC++17_03-M6.4.4	156
AutosarC++17_03-M6.4.5	156
AutosarC++17_03-M6.4.6	157
AutosarC++17_03-M6.4.7	157
AutosarC++17_03-M6.5.2	158
AutosarC++17_03-M6.5.3	158
AutosarC++17_03-M6.5.4	158
AutosarC++17_03-M6.5.5	159
AutosarC++17_03-M6.5.6	159
AutosarC++17_03-M6.6.1	160
AutosarC++17_03-M6.6.2	160
AutosarC++17_03-M6.6.3	160
AutosarC++17_03-M7.1.2	161
AutosarC++17_03-M7.3.1	161
AutosarC++17_03-M7.3.2	161
AutosarC++17_03-M7.3.3	162
AutosarC++17_03-M7.3.4	162
AutosarC++17_03-M7.3.5	162
AutosarC++17_03-M7.3.6	163
AutosarC++17_03-M7.4.1	163
AutosarC++17_03-M7.4.2	163
AutosarC++17_03-M7.4.3	164
AutosarC++17_03-M7.5.1	164
AutosarC++17_03-M7.5.2	164
AutosarC++17_03-M8.0.1	165
AutosarC++17_03-M8.3.1	165
AutosarC++17_03-M8.4.2	166
AutosarC++17_03-M8.4.4	166
AutosarC++17_03-M8.5.1	166
AutosarC++17_03-M8.5.2	167
AutosarC++17_03-M9.3.1	167
AutosarC++17_03-M9.3.3	168
AutosarC++17_03-M9.6.1	168
AutosarC++17_03-M10.1.1	169
AutosarC++17_03-M10.1.2	169
AutosarC++17_03-M10.1.3	169
AutosarC++17_03-M10.2.1	170
AutosarC++17_03-M10.3.3	170
AutosarC++17_03-M11.0.1	170
AutosarC++17_03-M12.1.1	171
AutosarC++17_03-M14.5.2	171
AutosarC++17_03-M14.5.3	172

AutosarC++17_03-M14.6.1	172
AutosarC++17_03-M14.7.3	172
AutosarC++17_03-M14.8.1	173
AutosarC++17_03-M15.0.3	173
AutosarC++17_03-M15.1.1	173
AutosarC++17_03-M15.1.2	174
AutosarC++17_03-M15.1.3	174
AutosarC++17_03-M15.3.1	174
AutosarC++17_03-M15.3.3	175
AutosarC++17_03-M15.3.4	175
AutosarC++17_03-M15.3.6	176
AutosarC++17_03-M15.3.7	176
AutosarC++17_03-M16.0.1	176
AutosarC++17_03-M16.0.2	177
AutosarC++17_03-M16.0.5	177
AutosarC++17_03-M16.0.6	178
AutosarC++17_03-M16.0.7	178
AutosarC++17_03-M16.0.8	178
AutosarC++17_03-M16.1.1	178
AutosarC++17_03-M16.1.2	179
AutosarC++17_03-M16.2.3	179
AutosarC++17_03-M16.3.1	180
AutosarC++17_03-M16.3.2	180
AutosarC++17_03-M17.0.2	180
AutosarC++17_03-M17.0.3	181
AutosarC++17_03-M17.0.5	181
AutosarC++17_03-M18.0.3	182
AutosarC++17_03-M18.0.4	182
AutosarC++17_03-M18.0.5	182
AutosarC++17_03-M18.2.1	183
AutosarC++17_03-M18.7.1	183
AutosarC++17_03-M19.3.1	184
AutosarC++17_03-M27.0.1	184
Rules in Group AutosarC++17_10	185
AutosarC++17_10-A0.1.1	185
AutosarC++17_10-A0.1.2	185
AutosarC++17_10-A0.1.3	186
AutosarC++17_10-A0.1.4	186
AutosarC++17_10-A0.1.5	187
AutosarC++17_10-A0.4.2	187
AutosarC++17_10-A1.1.1	187
AutosarC++17_10-A2.2.1	189
AutosarC++17_10-A2.5.1	190
AutosarC++17_10-A2.6.1	190
AutosarC++17_10-A2.8.1	190
AutosarC++17_10-A2.8.2	191
AutosarC++17_10-A2.8.3	191
AutosarC++17_10-A2.8.4	192
AutosarC++17_10-A2.9.1	192
AutosarC++17_10-A2.11.1	193
AutosarC++17_10-A2.11.2	194
AutosarC++17_10-A2.11.3	195
AutosarC++17_10-A2.11.4	195

AutosarC++17_10-A2.11.5	196
AutosarC++17_10-A2.14.1	196
AutosarC++17_10-A2.14.2	196
AutosarC++17_10-A2.14.3	197
AutosarC++17_10-A3.1.1	197
AutosarC++17_10-A3.1.2	198
AutosarC++17_10-A3.1.3	198
AutosarC++17_10-A3.1.4	199
AutosarC++17_10-A3.3.1	199
AutosarC++17_10-A3.3.2	200
AutosarC++17_10-A3.9.1	200
AutosarC++17_10-A4.5.1	200
AutosarC++17_10-A4.7.1	201
AutosarC++17_10-A4.10.1	201
AutosarC++17_10-A5.0.1	202
AutosarC++17_10-A5.0.2	202
AutosarC++17_10-A5.0.3	203
AutosarC++17_10-A5.1.1	203
AutosarC++17_10-A5.1.2	204
AutosarC++17_10-A5.1.5	204
AutosarC++17_10-A5.1.7	205
AutosarC++17_10-A5.1.8	205
AutosarC++17_10-A5.2.1	206
AutosarC++17_10-A5.2.2	206
AutosarC++17_10-A5.2.3	206
AutosarC++17_10-A5.2.4	207
AutosarC++17_10-A5.3.1	207
AutosarC++17_10-A5.5.1	208
AutosarC++17_10-A5.10.1	208
AutosarC++17_10-A5.16.1	208
AutosarC++17_10-A6.4.1	209
AutosarC++17_10-A6.5.1	209
AutosarC++17_10-A6.5.2	210
AutosarC++17_10-A6.5.3	210
AutosarC++17_10-A6.6.1	210
AutosarC++17_10-A7.1.1	211
AutosarC++17_10-A7.1.3	211
AutosarC++17_10-A7.1.4	212
AutosarC++17_10-A7.1.5	212
AutosarC++17_10-A7.1.6	213
AutosarC++17_10-A7.1.7	213
AutosarC++17_10-A7.2.1	214
AutosarC++17_10-A7.2.2	214
AutosarC++17_10-A7.2.3	215
AutosarC++17_10-A7.2.4	215
AutosarC++17_10-A7.4.1	215
AutosarC++17_10-A7.5.1	216
AutosarC++17_10-A7.5.2	216
AutosarC++17_10-A8.2.1	216
AutosarC++17_10-A8.4.1	217
AutosarC++17_10-A8.4.2	217
AutosarC++17_10-A8.4.4	218
AutosarC++17_10-A8.5.1	218

AutosarC++17_10-A8.5.2	219
AutosarC++17_10-A8.5.3	219
AutosarC++17_10-A8.5.4	219
AutosarC++17_10-A9.5.1	220
AutosarC++17_10-A9.6.1	220
AutosarC++17_10-A10.1.1	221
AutosarC++17_10-A10.2.1	221
AutosarC++17_10-A10.3.1	221
AutosarC++17_10-A10.3.2	222
AutosarC++17_10-A10.3.3	222
AutosarC++17_10-A10.3.5	222
AutosarC++17_10-A11.0.1	223
AutosarC++17_10-A11.0.2	223
AutosarC++17_10-A11.3.1	224
AutosarC++17_10-A12.0.1	224
AutosarC++17_10-A12.1.1	225
AutosarC++17_10-A12.1.2	225
AutosarC++17_10-A12.1.3	226
AutosarC++17_10-A12.1.4	226
AutosarC++17_10-A12.1.5	226
AutosarC++17_10-A12.1.6	227
AutosarC++17_10-A12.4.1	227
AutosarC++17_10-A12.4.2	228
AutosarC++17_10-A12.6.1	228
AutosarC++17_10-A12.8.1	228
AutosarC++17_10-A12.8.2	229
AutosarC++17_10-A12.8.3	229
AutosarC++17_10-A12.8.4	230
AutosarC++17_10-A12.8.6	230
AutosarC++17_10-A12.8.7	231
AutosarC++17_10-A13.1.1	231
AutosarC++17_10-A13.1.2	231
AutosarC++17_10-A13.1.3	232
AutosarC++17_10-A13.2.1	232
AutosarC++17_10-A13.2.2	232
AutosarC++17_10-A13.2.3	233
AutosarC++17_10-A13.3.1	233
AutosarC++17_10-A13.5.1	234
AutosarC++17_10-A13.5.2	234
AutosarC++17_10-A13.6.1	234
AutosarC++17_10-A15.1.1	235
AutosarC++17_10-A15.1.2	235
AutosarC++17_10-A15.1.3	235
AutosarC++17_10-A15.2.1	236
AutosarC++17_10-A15.3.1	236
AutosarC++17_10-A15.3.3	236
AutosarC++17_10-A15.3.4	237
AutosarC++17_10-A15.3.5	238
AutosarC++17_10-A15.4.1	238
AutosarC++17_10-A15.4.2	238
AutosarC++17_10-A15.4.3	239
AutosarC++17_10-A15.4.4	239
AutosarC++17_10-A15.4.5	240

AutosarC++17_10-A15.4.6	241
AutosarC++17_10-A15.5.1	241
AutosarC++17_10-A15.5.2	241
AutosarC++17_10-A15.5.3	242
AutosarC++17_10-A16.0.1	243
AutosarC++17_10-A16.2.1	244
AutosarC++17_10-A16.6.1	244
AutosarC++17_10-A16.7.1	245
AutosarC++17_10-A17.0.1	245
AutosarC++17_10-A18.0.1	246
AutosarC++17_10-A18.0.2	246
AutosarC++17_10-A18.0.3	246
AutosarC++17_10-A18.1.1	247
AutosarC++17_10-A18.1.2	247
AutosarC++17_10-A18.1.3	248
AutosarC++17_10-A18.1.4	248
AutosarC++17_10-A18.1.5	248
AutosarC++17_10-A18.1.6	249
AutosarC++17_10-A18.5.1	249
AutosarC++17_10-A18.5.2	249
AutosarC++17_10-A18.5.3	250
AutosarC++17_10-A18.5.4	250
AutosarC++17_10-A18.5.8	251
AutosarC++17_10-A18.9.1	251
AutosarC++17_10-A18.9.2	252
AutosarC++17_10-A18.9.3	252
AutosarC++17_10-A23.0.1	252
AutosarC++17_10-M0.1.1	253
AutosarC++17_10-M0.1.2	253
AutosarC++17_10-M0.1.3	254
AutosarC++17_10-M0.1.4	255
AutosarC++17_10-M0.1.5	255
AutosarC++17_10-M0.1.8	255
AutosarC++17_10-M0.1.9	256
AutosarC++17_10-M0.1.10	257
AutosarC++17_10-M0.2.1	258
AutosarC++17_10-M0.3.1	258
AutosarC++17_10-M0.3.2	261
AutosarC++17_10-M0.4.2	261
AutosarC++17_10-M2.10.1	262
AutosarC++17_10-M2.10.3	262
AutosarC++17_10-M2.10.6	263
AutosarC++17_10-M2.13.2	263
AutosarC++17_10-M2.13.3	263
AutosarC++17_10-M2.13.4	264
AutosarC++17_10-M3.1.2	264
AutosarC++17_10-M3.2.1	264
AutosarC++17_10-M3.2.2	265
AutosarC++17_10-M3.2.3	265
AutosarC++17_10-M3.2.4	266
AutosarC++17_10-M3.3.2	267
AutosarC++17_10-M3.4.1	267
AutosarC++17_10-M3.9.1	268

AutosarC++17_10-M3.9.3	268
AutosarC++17_10-M4.5.1	269
AutosarC++17_10-M4.5.3	269
AutosarC++17_10-M4.10.1	270
AutosarC++17_10-M4.10.2	270
AutosarC++17_10-M5.0.2	270
AutosarC++17_10-M5.0.3	271
AutosarC++17_10-M5.0.4	272
AutosarC++17_10-M5.0.5	272
AutosarC++17_10-M5.0.6	273
AutosarC++17_10-M5.0.7	274
AutosarC++17_10-M5.0.8	274
AutosarC++17_10-M5.0.9	275
AutosarC++17_10-M5.0.10	276
AutosarC++17_10-M5.0.11	276
AutosarC++17_10-M5.0.12	276
AutosarC++17_10-M5.0.14	277
AutosarC++17_10-M5.0.15	277
AutosarC++17_10-M5.0.16	277
AutosarC++17_10-M5.0.17	278
AutosarC++17_10-M5.0.18	278
AutosarC++17_10-M5.0.20	279
AutosarC++17_10-M5.0.21	279
AutosarC++17_10-M5.2.1	279
AutosarC++17_10-M5.2.2	280
AutosarC++17_10-M5.2.3	280
AutosarC++17_10-M5.2.6	280
AutosarC++17_10-M5.2.8	281
AutosarC++17_10-M5.2.9	282
AutosarC++17_10-M5.2.10	282
AutosarC++17_10-M5.2.11	283
AutosarC++17_10-M5.2.12	283
AutosarC++17_10-M5.3.1	283
AutosarC++17_10-M5.3.2	284
AutosarC++17_10-M5.3.3	284
AutosarC++17_10-M5.3.4	284
AutosarC++17_10-M5.8.1	285
AutosarC++17_10-M5.14.1	285
AutosarC++17_10-M5.17.1	286
AutosarC++17_10-M5.18.1	286
AutosarC++17_10-M5.19.1	286
AutosarC++17_10-M6.2.1	287
AutosarC++17_10-M6.2.2	287
AutosarC++17_10-M6.2.3	287
AutosarC++17_10-M6.3.1	288
AutosarC++17_10-M6.4.1	288
AutosarC++17_10-M6.4.2	288
AutosarC++17_10-M6.4.3	289
AutosarC++17_10-M6.4.4	291
AutosarC++17_10-M6.4.5	291
AutosarC++17_10-M6.4.6	292
AutosarC++17_10-M6.4.7	292
AutosarC++17_10-M6.5.2	293

AutosarC++17_10-M6.5.3	293
AutosarC++17_10-M6.5.4	293
AutosarC++17_10-M6.5.5	294
AutosarC++17_10-M6.5.6	294
AutosarC++17_10-M6.6.1	295
AutosarC++17_10-M6.6.2	295
AutosarC++17_10-M6.6.3	295
AutosarC++17_10-M7.1.2	296
AutosarC++17_10-M7.3.1	296
AutosarC++17_10-M7.3.2	296
AutosarC++17_10-M7.3.3	297
AutosarC++17_10-M7.3.4	297
AutosarC++17_10-M7.3.5	297
AutosarC++17_10-M7.3.6	298
AutosarC++17_10-M7.4.1	298
AutosarC++17_10-M7.4.2	298
AutosarC++17_10-M7.4.3	299
AutosarC++17_10-M7.5.1	299
AutosarC++17_10-M7.5.2	299
AutosarC++17_10-M8.0.1	300
AutosarC++17_10-M8.3.1	300
AutosarC++17_10-M8.4.2	301
AutosarC++17_10-M8.4.4	301
AutosarC++17_10-M8.5.1	301
AutosarC++17_10-M8.5.2	302
AutosarC++17_10-M9.3.1	302
AutosarC++17_10-M9.3.3	303
AutosarC++17_10-M9.6.1	303
AutosarC++17_10-M10.1.1	304
AutosarC++17_10-M10.1.2	304
AutosarC++17_10-M10.1.3	304
AutosarC++17_10-M10.2.1	305
AutosarC++17_10-M10.3.3	305
AutosarC++17_10-M11.0.1	305
AutosarC++17_10-M12.1.1	306
AutosarC++17_10-M14.5.2	306
AutosarC++17_10-M14.5.3	307
AutosarC++17_10-M14.6.1	307
AutosarC++17_10-M14.7.3	307
AutosarC++17_10-M14.8.1	308
AutosarC++17_10-M15.0.3	308
AutosarC++17_10-M15.1.1	308
AutosarC++17_10-M15.1.2	309
AutosarC++17_10-M15.1.3	309
AutosarC++17_10-M15.3.1	309
AutosarC++17_10-M15.3.3	310
AutosarC++17_10-M15.3.4	310
AutosarC++17_10-M15.3.6	311
AutosarC++17_10-M15.3.7	311
AutosarC++17_10-M16.0.1	311
AutosarC++17_10-M16.0.2	312
AutosarC++17_10-M16.0.5	312
AutosarC++17_10-M16.0.6	313

AutosarC++17_10-M16.0.7	313
AutosarC++17_10-M16.0.8	313
AutosarC++17_10-M16.1.1	313
AutosarC++17_10-M16.1.2	314
AutosarC++17_10-M16.2.3	314
AutosarC++17_10-M16.3.1	315
AutosarC++17_10-M16.3.2	315
AutosarC++17_10-M17.0.2	315
AutosarC++17_10-M17.0.3	316
AutosarC++17_10-M17.0.5	316
AutosarC++17_10-M18.0.3	317
AutosarC++17_10-M18.0.4	317
AutosarC++17_10-M18.0.5	317
AutosarC++17_10-M18.2.1	318
AutosarC++17_10-M18.7.1	318
AutosarC++17_10-M19.3.1	319
AutosarC++17_10-M27.0.1	319
Rules in Group AutosarC++18_03	320
AutosarC++18_03-A0.1.1	320
AutosarC++18_03-A0.1.2	320
AutosarC++18_03-A0.1.3	321
AutosarC++18_03-A0.1.4	321
AutosarC++18_03-A0.1.5	322
AutosarC++18_03-A0.1.6	322
AutosarC++18_03-A0.4.2	322
AutosarC++18_03-A1.1.1	323
AutosarC++18_03-A1.4.3	324
AutosarC++18_03-A2.3.1	325
AutosarC++18_03-A2.5.1	325
AutosarC++18_03-A2.5.2	326
AutosarC++18_03-A2.7.1	326
AutosarC++18_03-A2.7.2	326
AutosarC++18_03-A2.7.3	327
AutosarC++18_03-A2.8.1	328
AutosarC++18_03-A2.8.2	329
AutosarC++18_03-A2.10.1	329
AutosarC++18_03-A2.10.4	330
AutosarC++18_03-A2.10.5	331
AutosarC++18_03-A2.10.6	331
AutosarC++18_03-A2.11.1	332
AutosarC++18_03-A2.13.1	332
AutosarC++18_03-A2.13.2	332
AutosarC++18_03-A2.13.3	333
AutosarC++18_03-A2.13.4	333
AutosarC++18_03-A2.13.5	334
AutosarC++18_03-A2.13.6	334
AutosarC++18_03-A3.1.1	335
AutosarC++18_03-A3.1.2	335
AutosarC++18_03-A3.1.3	335
AutosarC++18_03-A3.1.4	336
AutosarC++18_03-A3.1.6	336
AutosarC++18_03-A3.3.1	337
AutosarC++18_03-A3.3.2	337

AutosarC++18_03-A3.9.1	338
AutosarC++18_03-A4.5.1	338
AutosarC++18_03-A4.7.1	338
AutosarC++18_03-A4.10.1	339
AutosarC++18_03-A5.0.1	339
AutosarC++18_03-A5.0.2	340
AutosarC++18_03-A5.0.3	340
AutosarC++18_03-A5.0.4	341
AutosarC++18_03-A5.1.1	341
AutosarC++18_03-A5.1.2	342
AutosarC++18_03-A5.1.3	342
AutosarC++18_03-A5.1.6	343
AutosarC++18_03-A5.1.7	343
AutosarC++18_03-A5.1.8	344
AutosarC++18_03-A5.1.9	344
AutosarC++18_03-A5.2.1	344
AutosarC++18_03-A5.2.2	345
AutosarC++18_03-A5.2.3	345
AutosarC++18_03-A5.2.4	346
AutosarC++18_03-A5.2.6	346
AutosarC++18_03-A5.3.1	346
AutosarC++18_03-A5.6.1	347
AutosarC++18_03-A5.10.1	347
AutosarC++18_03-A5.16.1	348
AutosarC++18_03-A6.4.1	348
AutosarC++18_03-A6.5.1	348
AutosarC++18_03-A6.5.2	349
AutosarC++18_03-A6.5.3	349
AutosarC++18_03-A6.5.4	350
AutosarC++18_03-A6.6.1	350
AutosarC++18_03-A7.1.1	350
AutosarC++18_03-A7.1.3	351
AutosarC++18_03-A7.1.4	351
AutosarC++18_03-A7.1.5	352
AutosarC++18_03-A7.1.6	352
AutosarC++18_03-A7.1.7	353
AutosarC++18_03-A7.1.9	353
AutosarC++18_03-A7.2.1	353
AutosarC++18_03-A7.2.2	354
AutosarC++18_03-A7.2.3	354
AutosarC++18_03-A7.2.4	355
AutosarC++18_03-A7.3.1	355
AutosarC++18_03-A7.4.1	355
AutosarC++18_03-A7.5.1	356
AutosarC++18_03-A7.5.2	356
AutosarC++18_03-A7.6.1	356
AutosarC++18_03-A8.2.1	357
AutosarC++18_03-A8.4.1	357
AutosarC++18_03-A8.4.2	358
AutosarC++18_03-A8.4.4	358
AutosarC++18_03-A8.4.5	359
AutosarC++18_03-A8.4.6	359
AutosarC++18_03-A8.4.7	360

AutosarC++18_03-A8.4.8	360
AutosarC++18_03-A8.4.9	361
AutosarC++18_03-A8.5.1	361
AutosarC++18_03-A8.5.2	362
AutosarC++18_03-A8.5.3	362
AutosarC++18_03-A8.5.4	363
AutosarC++18_03-A9.5.1	363
AutosarC++18_03-A9.6.1	363
AutosarC++18_03-A10.1.1	364
AutosarC++18_03-A10.2.1	364
AutosarC++18_03-A10.3.1	364
AutosarC++18_03-A10.3.2	365
AutosarC++18_03-A10.3.3	365
AutosarC++18_03-A10.3.5	366
AutosarC++18_03-A11.0.1	366
AutosarC++18_03-A11.0.2	366
AutosarC++18_03-A11.3.1	367
AutosarC++18_03-A12.0.1	367
AutosarC++18_03-A12.1.1	368
AutosarC++18_03-A12.1.2	368
AutosarC++18_03-A12.1.3	369
AutosarC++18_03-A12.1.4	369
AutosarC++18_03-A12.1.5	370
AutosarC++18_03-A12.1.6	370
AutosarC++18_03-A12.4.1	370
AutosarC++18_03-A12.4.2	371
AutosarC++18_03-A12.6.1	371
AutosarC++18_03-A12.8.1	372
AutosarC++18_03-A12.8.2	372
AutosarC++18_03-A12.8.3	373
AutosarC++18_03-A12.8.4	373
AutosarC++18_03-A12.8.6	374
AutosarC++18_03-A12.8.7	374
AutosarC++18_03-A13.1.2	374
AutosarC++18_03-A13.1.3	375
AutosarC++18_03-A13.2.1	375
AutosarC++18_03-A13.2.2	375
AutosarC++18_03-A13.2.3	376
AutosarC++18_03-A13.3.1	376
AutosarC++18_03-A13.5.1	377
AutosarC++18_03-A13.5.2	377
AutosarC++18_03-A13.5.3	377
AutosarC++18_03-A13.5.4	378
AutosarC++18_03-A13.6.1	378
AutosarC++18_03-A14.7.2	378
AutosarC++18_03-A14.8.2	379
AutosarC++18_03-A15.1.1	379
AutosarC++18_03-A15.1.2	379
AutosarC++18_03-A15.1.3	380
AutosarC++18_03-A15.2.1	380
AutosarC++18_03-A15.3.3	381
AutosarC++18_03-A15.3.4	381
AutosarC++18_03-A15.3.5	382

AutosarC++18_03-A15.4.1	382
AutosarC++18_03-A15.4.2	383
AutosarC++18_03-A15.4.3	383
AutosarC++18_03-A15.4.4	384
AutosarC++18_03-A15.4.5	384
AutosarC++18_03-A15.5.1	385
AutosarC++18_03-A15.5.2	385
AutosarC++18_03-A15.5.3	386
AutosarC++18_03-A16.0.1	387
AutosarC++18_03-A16.2.1	388
AutosarC++18_03-A16.6.1	388
AutosarC++18_03-A16.7.1	389
AutosarC++18_03-A17.0.1	389
AutosarC++18_03-A17.6.1	390
AutosarC++18_03-A18.0.1	390
AutosarC++18_03-A18.0.2	391
AutosarC++18_03-A18.0.3	391
AutosarC++18_03-A18.1.1	392
AutosarC++18_03-A18.1.2	392
AutosarC++18_03-A18.1.3	393
AutosarC++18_03-A18.1.4	393
AutosarC++18_03-A18.1.6	393
AutosarC++18_03-A18.5.1	394
AutosarC++18_03-A18.5.2	394
AutosarC++18_03-A18.5.3	394
AutosarC++18_03-A18.5.4	395
AutosarC++18_03-A18.5.8	395
AutosarC++18_03-A18.9.1	396
AutosarC++18_03-A18.9.2	396
AutosarC++18_03-A18.9.3	396
AutosarC++18_03-A21.8.1	397
AutosarC++18_03-A23.0.1	397
AutosarC++18_03-A26.5.1	398
AutosarC++18_03-A26.5.2	398
AutosarC++18_03-A27.0.4	398
AutosarC++18_03-M0.1.1	399
AutosarC++18_03-M0.1.2	399
AutosarC++18_03-M0.1.3	400
AutosarC++18_03-M0.1.4	401
AutosarC++18_03-M0.1.8	401
AutosarC++18_03-M0.1.9	401
AutosarC++18_03-M0.1.10	403
AutosarC++18_03-M0.2.1	404
AutosarC++18_03-M0.3.1	404
AutosarC++18_03-M0.3.2	407
AutosarC++18_03-M0.4.2	407
AutosarC++18_03-M2.7.1	408
AutosarC++18_03-M2.10.1	408
AutosarC++18_03-M2.13.2	408
AutosarC++18_03-M2.13.3	409
AutosarC++18_03-M2.13.4	409
AutosarC++18_03-M3.1.2	410
AutosarC++18_03-M3.2.1	410

AutosarC++18_03-M3.2.2	411
AutosarC++18_03-M3.2.3	411
AutosarC++18_03-M3.2.4	412
AutosarC++18_03-M3.3.2	412
AutosarC++18_03-M3.4.1	413
AutosarC++18_03-M3.9.1	413
AutosarC++18_03-M3.9.3	414
AutosarC++18_03-M4.5.1	414
AutosarC++18_03-M4.5.3	415
AutosarC++18_03-M4.10.1	415
AutosarC++18_03-M4.10.2	415
AutosarC++18_03-M5.0.2	416
AutosarC++18_03-M5.0.3	416
AutosarC++18_03-M5.0.4	417
AutosarC++18_03-M5.0.5	418
AutosarC++18_03-M5.0.6	419
AutosarC++18_03-M5.0.7	419
AutosarC++18_03-M5.0.8	420
AutosarC++18_03-M5.0.9	421
AutosarC++18_03-M5.0.10	421
AutosarC++18_03-M5.0.11	421
AutosarC++18_03-M5.0.12	422
AutosarC++18_03-M5.0.14	422
AutosarC++18_03-M5.0.15	422
AutosarC++18_03-M5.0.16	423
AutosarC++18_03-M5.0.17	423
AutosarC++18_03-M5.0.18	424
AutosarC++18_03-M5.0.20	424
AutosarC++18_03-M5.0.21	424
AutosarC++18_03-M5.2.2	425
AutosarC++18_03-M5.2.3	425
AutosarC++18_03-M5.2.6	425
AutosarC++18_03-M5.2.8	426
AutosarC++18_03-M5.2.9	427
AutosarC++18_03-M5.2.10	428
AutosarC++18_03-M5.2.11	428
AutosarC++18_03-M5.2.12	428
AutosarC++18_03-M5.3.1	429
AutosarC++18_03-M5.3.2	429
AutosarC++18_03-M5.3.3	429
AutosarC++18_03-M5.3.4	430
AutosarC++18_03-M5.8.1	430
AutosarC++18_03-M5.14.1	430
AutosarC++18_03-M5.17.1	431
AutosarC++18_03-M5.18.1	431
AutosarC++18_03-M5.19.1	432
AutosarC++18_03-M6.2.1	432
AutosarC++18_03-M6.2.2	432
AutosarC++18_03-M6.2.3	433
AutosarC++18_03-M6.3.1	433
AutosarC++18_03-M6.4.1	433
AutosarC++18_03-M6.4.2	434
AutosarC++18_03-M6.4.3	434

AutosarC++18_03-M6.4.4	436
AutosarC++18_03-M6.4.5	436
AutosarC++18_03-M6.4.6	437
AutosarC++18_03-M6.4.7	437
AutosarC++18_03-M6.5.2	438
AutosarC++18_03-M6.5.3	438
AutosarC++18_03-M6.5.4	438
AutosarC++18_03-M6.5.5	439
AutosarC++18_03-M6.5.6	439
AutosarC++18_03-M6.6.1	440
AutosarC++18_03-M6.6.2	440
AutosarC++18_03-M6.6.3	440
AutosarC++18_03-M7.1.2	441
AutosarC++18_03-M7.3.1	441
AutosarC++18_03-M7.3.2	441
AutosarC++18_03-M7.3.3	442
AutosarC++18_03-M7.3.4	442
AutosarC++18_03-M7.3.6	442
AutosarC++18_03-M7.4.1	443
AutosarC++18_03-M7.4.2	443
AutosarC++18_03-M7.4.3	443
AutosarC++18_03-M7.5.1	444
AutosarC++18_03-M7.5.2	444
AutosarC++18_03-M8.0.1	444
AutosarC++18_03-M8.3.1	445
AutosarC++18_03-M8.4.2	445
AutosarC++18_03-M8.4.4	446
AutosarC++18_03-M8.5.2	446
AutosarC++18_03-M9.3.1	446
AutosarC++18_03-M9.3.3	447
AutosarC++18_03-M9.6.1	447
AutosarC++18_03-M10.1.1	448
AutosarC++18_03-M10.1.2	448
AutosarC++18_03-M10.1.3	448
AutosarC++18_03-M10.2.1	449
AutosarC++18_03-M10.3.3	449
AutosarC++18_03-M11.0.1	449
AutosarC++18_03-M12.1.1	450
AutosarC++18_03-M14.5.3	450
AutosarC++18_03-M14.6.1	451
AutosarC++18_03-M15.0.3	451
AutosarC++18_03-M15.1.1	451
AutosarC++18_03-M15.1.2	452
AutosarC++18_03-M15.1.3	452
AutosarC++18_03-M15.3.1	452
AutosarC++18_03-M15.3.3	453
AutosarC++18_03-M15.3.4	453
AutosarC++18_03-M15.3.6	454
AutosarC++18_03-M15.3.7	454
AutosarC++18_03-M16.0.1	454
AutosarC++18_03-M16.0.2	455
AutosarC++18_03-M16.0.5	455
AutosarC++18_03-M16.0.6	455

AutosarC++18_03-M16.0.7	456
AutosarC++18_03-M16.0.8	456
AutosarC++18_03-M16.1.1	456
AutosarC++18_03-M16.1.2	457
AutosarC++18_03-M16.2.3	457
AutosarC++18_03-M16.3.1	458
AutosarC++18_03-M16.3.2	458
AutosarC++18_03-M17.0.2	458
AutosarC++18_03-M17.0.3	459
AutosarC++18_03-M17.0.5	459
AutosarC++18_03-M18.0.3	460
AutosarC++18_03-M18.0.4	460
AutosarC++18_03-M18.0.5	460
AutosarC++18_03-M18.2.1	461
AutosarC++18_03-M18.7.1	461
AutosarC++18_03-M19.3.1	462
AutosarC++18_03-M27.0.1	462
Rules in Group AutosarC++18_10	463
AutosarC++18_10-A0.1.1	463
AutosarC++18_10-A0.1.2	463
AutosarC++18_10-A0.1.3	464
AutosarC++18_10-A0.1.4	464
AutosarC++18_10-A0.1.5	465
AutosarC++18_10-A0.1.6	465
AutosarC++18_10-A0.4.2	465
AutosarC++18_10-A1.1.1	466
AutosarC++18_10-A1.4.3	467
AutosarC++18_10-A2.3.1	468
AutosarC++18_10-A2.5.1	468
AutosarC++18_10-A2.5.2	469
AutosarC++18_10-A2.7.1	469
AutosarC++18_10-A2.7.2	469
AutosarC++18_10-A2.7.3	470
AutosarC++18_10-A2.8.1	471
AutosarC++18_10-A2.8.2	472
AutosarC++18_10-A2.10.1	472
AutosarC++18_10-A2.10.4	473
AutosarC++18_10-A2.10.5	474
AutosarC++18_10-A2.10.6	474
AutosarC++18_10-A2.11.1	475
AutosarC++18_10-A2.13.1	475
AutosarC++18_10-A2.13.2	475
AutosarC++18_10-A2.13.3	476
AutosarC++18_10-A2.13.4	476
AutosarC++18_10-A2.13.5	477
AutosarC++18_10-A2.13.6	477
AutosarC++18_10-A3.1.1	478
AutosarC++18_10-A3.1.2	478
AutosarC++18_10-A3.1.3	478
AutosarC++18_10-A3.1.4	479
AutosarC++18_10-A3.1.6	479
AutosarC++18_10-A3.3.1	480
AutosarC++18_10-A3.3.2	480

AutosarC++18_10-A3.9.1	481
AutosarC++18_10-A4.5.1	481
AutosarC++18_10-A4.7.1	481
AutosarC++18_10-A4.10.1	482
AutosarC++18_10-A5.0.1	482
AutosarC++18_10-A5.0.2	483
AutosarC++18_10-A5.0.3	483
AutosarC++18_10-A5.0.4	484
AutosarC++18_10-A5.1.1	484
AutosarC++18_10-A5.1.2	485
AutosarC++18_10-A5.1.3	485
AutosarC++18_10-A5.1.6	486
AutosarC++18_10-A5.1.7	486
AutosarC++18_10-A5.1.8	487
AutosarC++18_10-A5.1.9	487
AutosarC++18_10-A5.2.1	487
AutosarC++18_10-A5.2.2	488
AutosarC++18_10-A5.2.3	488
AutosarC++18_10-A5.2.4	489
AutosarC++18_10-A5.2.6	489
AutosarC++18_10-A5.3.1	489
AutosarC++18_10-A5.6.1	490
AutosarC++18_10-A5.10.1	490
AutosarC++18_10-A5.16.1	491
AutosarC++18_10-A6.4.1	491
AutosarC++18_10-A6.5.1	491
AutosarC++18_10-A6.5.2	492
AutosarC++18_10-A6.5.3	492
AutosarC++18_10-A6.5.4	493
AutosarC++18_10-A6.6.1	493
AutosarC++18_10-A7.1.1	493
AutosarC++18_10-A7.1.3	494
AutosarC++18_10-A7.1.4	494
AutosarC++18_10-A7.1.5	495
AutosarC++18_10-A7.1.6	495
AutosarC++18_10-A7.1.7	496
AutosarC++18_10-A7.1.9	496
AutosarC++18_10-A7.2.1	496
AutosarC++18_10-A7.2.2	497
AutosarC++18_10-A7.2.3	497
AutosarC++18_10-A7.2.4	498
AutosarC++18_10-A7.3.1	498
AutosarC++18_10-A7.4.1	498
AutosarC++18_10-A7.5.1	499
AutosarC++18_10-A7.5.2	499
AutosarC++18_10-A7.6.1	499
AutosarC++18_10-A8.2.1	500
AutosarC++18_10-A8.4.1	500
AutosarC++18_10-A8.4.2	501
AutosarC++18_10-A8.4.4	501
AutosarC++18_10-A8.4.5	502
AutosarC++18_10-A8.4.6	502
AutosarC++18_10-A8.4.7	503

AutosarC++18_10-A8.4.8	503
AutosarC++18_10-A8.4.9	504
AutosarC++18_10-A8.5.1	504
AutosarC++18_10-A8.5.2	505
AutosarC++18_10-A8.5.3	505
AutosarC++18_10-A9.5.1	506
AutosarC++18_10-A10.1.1	506
AutosarC++18_10-A10.3.1	506
AutosarC++18_10-A10.3.2	507
AutosarC++18_10-A10.3.3	507
AutosarC++18_10-A10.3.5	508
AutosarC++18_10-A11.0.1	508
AutosarC++18_10-A11.0.2	508
AutosarC++18_10-A12.0.1	509
AutosarC++18_10-A12.1.1	509
AutosarC++18_10-A12.1.2	510
AutosarC++18_10-A12.1.3	510
AutosarC++18_10-A12.1.4	511
AutosarC++18_10-A12.1.5	511
AutosarC++18_10-A12.1.6	512
AutosarC++18_10-A12.4.1	512
AutosarC++18_10-A12.4.2	512
AutosarC++18_10-A12.6.1	513
AutosarC++18_10-A12.8.1	513
AutosarC++18_10-A12.8.2	514
AutosarC++18_10-A12.8.3	514
AutosarC++18_10-A12.8.4	515
AutosarC++18_10-A12.8.6	515
AutosarC++18_10-A12.8.7	516
AutosarC++18_10-A13.1.2	516
AutosarC++18_10-A13.1.3	516
AutosarC++18_10-A13.2.1	517
AutosarC++18_10-A13.2.2	517
AutosarC++18_10-A13.2.3	518
AutosarC++18_10-A13.3.1	518
AutosarC++18_10-A13.5.1	518
AutosarC++18_10-A13.5.2	519
AutosarC++18_10-A13.5.3	519
AutosarC++18_10-A13.5.4	519
AutosarC++18_10-A13.6.1	520
AutosarC++18_10-A14.7.2	520
AutosarC++18_10-A14.8.2	520
AutosarC++18_10-A15.1.1	521
AutosarC++18_10-A15.1.2	521
AutosarC++18_10-A15.1.3	521
AutosarC++18_10-A15.2.1	522
AutosarC++18_10-A15.3.3	522
AutosarC++18_10-A15.3.4	523
AutosarC++18_10-A15.3.5	523
AutosarC++18_10-A15.4.1	524
AutosarC++18_10-A15.4.2	524
AutosarC++18_10-A15.4.4	525
AutosarC++18_10-A15.4.5	525

AutosarC++18_10-A15.5.1	526
AutosarC++18_10-A15.5.2	526
AutosarC++18_10-A15.5.3	527
AutosarC++18_10-A16.0.1	528
AutosarC++18_10-A16.2.1	529
AutosarC++18_10-A16.6.1	529
AutosarC++18_10-A16.7.1	530
AutosarC++18_10-A17.0.1	530
AutosarC++18_10-A17.6.1	531
AutosarC++18_10-A18.0.1	531
AutosarC++18_10-A18.0.2	532
AutosarC++18_10-A18.0.3	532
AutosarC++18_10-A18.1.1	533
AutosarC++18_10-A18.1.2	533
AutosarC++18_10-A18.1.3	534
AutosarC++18_10-A18.1.4	534
AutosarC++18_10-A18.1.6	534
AutosarC++18_10-A18.5.1	535
AutosarC++18_10-A18.5.3	535
AutosarC++18_10-A18.5.4	535
AutosarC++18_10-A18.5.8	536
AutosarC++18_10-A18.9.1	536
AutosarC++18_10-A18.9.2	537
AutosarC++18_10-A18.9.3	537
AutosarC++18_10-A21.8.1	538
AutosarC++18_10-A23.0.1	538
AutosarC++18_10-A26.5.1	538
AutosarC++18_10-A26.5.2	539
AutosarC++18_10-A27.0.4	539
AutosarC++18_10-M0.1.1	540
AutosarC++18_10-M0.1.2	540
AutosarC++18_10-M0.1.3	541
AutosarC++18_10-M0.1.4	542
AutosarC++18_10-M0.1.8	542
AutosarC++18_10-M0.1.9	542
AutosarC++18_10-M0.1.10	544
AutosarC++18_10-M0.2.1	545
AutosarC++18_10-M0.3.1	545
AutosarC++18_10-M0.3.2	548
AutosarC++18_10-M0.4.2	548
AutosarC++18_10-M2.7.1	549
AutosarC++18_10-M2.10.1	549
AutosarC++18_10-M2.13.2	549
AutosarC++18_10-M2.13.3	550
AutosarC++18_10-M2.13.4	550
AutosarC++18_10-M3.1.2	551
AutosarC++18_10-M3.2.1	551
AutosarC++18_10-M3.2.2	552
AutosarC++18_10-M3.2.3	552
AutosarC++18_10-M3.2.4	553
AutosarC++18_10-M3.3.2	553
AutosarC++18_10-M3.4.1	554
AutosarC++18_10-M3.9.1	554

AutosarC++18_10-M3.9.3	555
AutosarC++18_10-M4.5.1	555
AutosarC++18_10-M4.5.3	556
AutosarC++18_10-M4.10.1	556
AutosarC++18_10-M4.10.2	556
AutosarC++18_10-M5.0.2	557
AutosarC++18_10-M5.0.3	557
AutosarC++18_10-M5.0.4	558
AutosarC++18_10-M5.0.5	559
AutosarC++18_10-M5.0.6	560
AutosarC++18_10-M5.0.7	560
AutosarC++18_10-M5.0.8	561
AutosarC++18_10-M5.0.9	562
AutosarC++18_10-M5.0.10	562
AutosarC++18_10-M5.0.11	562
AutosarC++18_10-M5.0.12	563
AutosarC++18_10-M5.0.14	563
AutosarC++18_10-M5.0.15	563
AutosarC++18_10-M5.0.16	564
AutosarC++18_10-M5.0.17	564
AutosarC++18_10-M5.0.18	565
AutosarC++18_10-M5.0.20	565
AutosarC++18_10-M5.0.21	565
AutosarC++18_10-M5.2.2	566
AutosarC++18_10-M5.2.3	566
AutosarC++18_10-M5.2.6	566
AutosarC++18_10-M5.2.8	567
AutosarC++18_10-M5.2.9	568
AutosarC++18_10-M5.2.10	569
AutosarC++18_10-M5.2.11	569
AutosarC++18_10-M5.2.12	569
AutosarC++18_10-M5.3.1	570
AutosarC++18_10-M5.3.2	570
AutosarC++18_10-M5.3.3	570
AutosarC++18_10-M5.3.4	571
AutosarC++18_10-M5.8.1	571
AutosarC++18_10-M5.14.1	571
AutosarC++18_10-M5.17.1	572
AutosarC++18_10-M5.18.1	572
AutosarC++18_10-M5.19.1	573
AutosarC++18_10-M6.2.1	573
AutosarC++18_10-M6.2.2	573
AutosarC++18_10-M6.2.3	574
AutosarC++18_10-M6.3.1	574
AutosarC++18_10-M6.4.1	574
AutosarC++18_10-M6.4.2	575
AutosarC++18_10-M6.4.3	575
AutosarC++18_10-M6.4.4	577
AutosarC++18_10-M6.4.5	577
AutosarC++18_10-M6.4.6	578
AutosarC++18_10-M6.4.7	578
AutosarC++18_10-M6.5.2	579
AutosarC++18_10-M6.5.3	579

AutosarC++18_10-M6.5.4	579
AutosarC++18_10-M6.5.5	580
AutosarC++18_10-M6.5.6	580
AutosarC++18_10-M6.6.1	581
AutosarC++18_10-M6.6.2	581
AutosarC++18_10-M6.6.3	581
AutosarC++18_10-M7.1.2	582
AutosarC++18_10-M7.3.1	582
AutosarC++18_10-M7.3.2	582
AutosarC++18_10-M7.3.3	583
AutosarC++18_10-M7.3.4	583
AutosarC++18_10-M7.3.6	583
AutosarC++18_10-M7.4.1	584
AutosarC++18_10-M7.4.2	584
AutosarC++18_10-M7.4.3	584
AutosarC++18_10-M7.5.1	585
AutosarC++18_10-M7.5.2	585
AutosarC++18_10-M8.0.1	585
AutosarC++18_10-M8.3.1	586
AutosarC++18_10-M8.4.2	586
AutosarC++18_10-M8.4.4	587
AutosarC++18_10-M8.5.2	587
AutosarC++18_10-M9.3.1	587
AutosarC++18_10-M9.3.3	588
AutosarC++18_10-M9.6.1	588
AutosarC++18_10-M9.6.4	589
AutosarC++18_10-M10.1.1	589
AutosarC++18_10-M10.1.2	589
AutosarC++18_10-M10.1.3	590
AutosarC++18_10-M10.2.1	590
AutosarC++18_10-M10.3.3	590
AutosarC++18_10-M11.0.1	591
AutosarC++18_10-M12.1.1	591
AutosarC++18_10-M14.5.3	592
AutosarC++18_10-M14.6.1	592
AutosarC++18_10-M15.0.3	592
AutosarC++18_10-M15.1.1	593
AutosarC++18_10-M15.1.2	593
AutosarC++18_10-M15.1.3	593
AutosarC++18_10-M15.3.1	594
AutosarC++18_10-M15.3.3	594
AutosarC++18_10-M15.3.4	594
AutosarC++18_10-M15.3.6	595
AutosarC++18_10-M15.3.7	595
AutosarC++18_10-M16.0.1	596
AutosarC++18_10-M16.0.2	596
AutosarC++18_10-M16.0.5	596
AutosarC++18_10-M16.0.6	597
AutosarC++18_10-M16.0.7	597
AutosarC++18_10-M16.0.8	597
AutosarC++18_10-M16.1.1	598
AutosarC++18_10-M16.1.2	598
AutosarC++18_10-M16.2.3	598

AutosarC++18_10-M16.3.1	599
AutosarC++18_10-M16.3.2	599
AutosarC++18_10-M17.0.2	600
AutosarC++18_10-M17.0.3	600
AutosarC++18_10-M17.0.5	600
AutosarC++18_10-M18.0.3	601
AutosarC++18_10-M18.0.4	601
AutosarC++18_10-M18.0.5	602
AutosarC++18_10-M18.2.1	602
AutosarC++18_10-M18.7.1	603
AutosarC++18_10-M19.3.1	603
AutosarC++18_10-M27.0.1	604
Rules in Group AutosarC++19_03	604
AutosarC++19_03-A0.1.1	604
AutosarC++19_03-A0.1.2	605
AutosarC++19_03-A0.1.3	605
AutosarC++19_03-A0.1.4	606
AutosarC++19_03-A0.1.5	606
AutosarC++19_03-A0.1.6	607
AutosarC++19_03-A0.4.2	607
AutosarC++19_03-A1.1.1	608
AutosarC++19_03-A1.4.3	609
AutosarC++19_03-A2.3.1	610
AutosarC++19_03-A2.5.1	610
AutosarC++19_03-A2.5.2	611
AutosarC++19_03-A2.7.1	611
AutosarC++19_03-A2.7.2	611
AutosarC++19_03-A2.7.3	612
AutosarC++19_03-A2.8.1	613
AutosarC++19_03-A2.8.2	614
AutosarC++19_03-A2.10.1	614
AutosarC++19_03-A2.10.4	615
AutosarC++19_03-A2.10.5	616
AutosarC++19_03-A2.10.6	616
AutosarC++19_03-A2.11.1	617
AutosarC++19_03-A2.13.1	617
AutosarC++19_03-A2.13.2	617
AutosarC++19_03-A2.13.3	618
AutosarC++19_03-A2.13.4	618
AutosarC++19_03-A2.13.5	619
AutosarC++19_03-A2.13.6	619
AutosarC++19_03-A3.1.1	620
AutosarC++19_03-A3.1.2	620
AutosarC++19_03-A3.1.3	620
AutosarC++19_03-A3.1.4	621
AutosarC++19_03-A3.1.6	621
AutosarC++19_03-A3.3.1	622
AutosarC++19_03-A3.3.2	622
AutosarC++19_03-A3.9.1	623
AutosarC++19_03-A4.5.1	623
AutosarC++19_03-A4.7.1	623
AutosarC++19_03-A4.10.1	624
AutosarC++19_03-A5.0.1	624

AutosarC++19_03-A5.0.2	625
AutosarC++19_03-A5.0.3	625
AutosarC++19_03-A5.0.4	626
AutosarC++19_03-A5.1.1	626
AutosarC++19_03-A5.1.2	627
AutosarC++19_03-A5.1.3	627
AutosarC++19_03-A5.1.6	628
AutosarC++19_03-A5.1.7	628
AutosarC++19_03-A5.1.8	629
AutosarC++19_03-A5.1.9	629
AutosarC++19_03-A5.2.1	629
AutosarC++19_03-A5.2.2	630
AutosarC++19_03-A5.2.3	630
AutosarC++19_03-A5.2.4	631
AutosarC++19_03-A5.2.6	631
AutosarC++19_03-A5.3.1	631
AutosarC++19_03-A5.6.1	632
AutosarC++19_03-A5.10.1	632
AutosarC++19_03-A5.16.1	633
AutosarC++19_03-A6.4.1	633
AutosarC++19_03-A6.5.1	633
AutosarC++19_03-A6.5.2	634
AutosarC++19_03-A6.5.3	634
AutosarC++19_03-A6.5.4	635
AutosarC++19_03-A6.6.1	635
AutosarC++19_03-A7.1.1	635
AutosarC++19_03-A7.1.3	636
AutosarC++19_03-A7.1.4	636
AutosarC++19_03-A7.1.5	637
AutosarC++19_03-A7.1.6	637
AutosarC++19_03-A7.1.7	638
AutosarC++19_03-A7.1.9	638
AutosarC++19_03-A7.2.1	638
AutosarC++19_03-A7.2.2	639
AutosarC++19_03-A7.2.3	639
AutosarC++19_03-A7.2.4	640
AutosarC++19_03-A7.3.1	640
AutosarC++19_03-A7.4.1	640
AutosarC++19_03-A7.5.1	641
AutosarC++19_03-A7.5.2	641
AutosarC++19_03-A7.6.1	641
AutosarC++19_03-A8.2.1	642
AutosarC++19_03-A8.4.1	642
AutosarC++19_03-A8.4.2	643
AutosarC++19_03-A8.4.4	643
AutosarC++19_03-A8.4.5	644
AutosarC++19_03-A8.4.6	644
AutosarC++19_03-A8.4.7	645
AutosarC++19_03-A8.4.8	645
AutosarC++19_03-A8.4.9	646
AutosarC++19_03-A8.5.1	646
AutosarC++19_03-A8.5.2	647
AutosarC++19_03-A8.5.3	647

AutosarC++19_03-A9.5.1	648
AutosarC++19_03-A10.1.1	648
AutosarC++19_03-A10.3.1	648
AutosarC++19_03-A10.3.2	649
AutosarC++19_03-A10.3.3	649
AutosarC++19_03-A10.3.5	650
AutosarC++19_03-A11.0.1	650
AutosarC++19_03-A11.0.2	650
AutosarC++19_03-A12.0.1	651
AutosarC++19_03-A12.1.1	651
AutosarC++19_03-A12.1.2	652
AutosarC++19_03-A12.1.3	652
AutosarC++19_03-A12.1.4	653
AutosarC++19_03-A12.1.5	653
AutosarC++19_03-A12.1.6	654
AutosarC++19_03-A12.4.1	654
AutosarC++19_03-A12.4.2	654
AutosarC++19_03-A12.6.1	655
AutosarC++19_03-A12.8.1	655
AutosarC++19_03-A12.8.2	656
AutosarC++19_03-A12.8.3	656
AutosarC++19_03-A12.8.4	657
AutosarC++19_03-A12.8.6	657
AutosarC++19_03-A12.8.7	658
AutosarC++19_03-A13.1.2	658
AutosarC++19_03-A13.1.3	658
AutosarC++19_03-A13.2.1	659
AutosarC++19_03-A13.2.2	659
AutosarC++19_03-A13.2.3	660
AutosarC++19_03-A13.3.1	660
AutosarC++19_03-A13.5.1	660
AutosarC++19_03-A13.5.2	661
AutosarC++19_03-A13.5.3	661
AutosarC++19_03-A13.5.4	661
AutosarC++19_03-A13.6.1	662
AutosarC++19_03-A14.7.2	662
AutosarC++19_03-A14.8.2	662
AutosarC++19_03-A15.1.1	663
AutosarC++19_03-A15.1.2	663
AutosarC++19_03-A15.1.3	663
AutosarC++19_03-A15.2.1	664
AutosarC++19_03-A15.3.3	664
AutosarC++19_03-A15.3.4	665
AutosarC++19_03-A15.3.5	665
AutosarC++19_03-A15.4.1	666
AutosarC++19_03-A15.4.2	666
AutosarC++19_03-A15.4.4	667
AutosarC++19_03-A15.4.5	667
AutosarC++19_03-A15.5.1	668
AutosarC++19_03-A15.5.2	668
AutosarC++19_03-A15.5.3	669
AutosarC++19_03-A16.0.1	670
AutosarC++19_03-A16.2.1	671

AutosarC++19_03-A16.6.1	671
AutosarC++19_03-A16.7.1	672
AutosarC++19_03-A17.0.1	672
AutosarC++19_03-A17.6.1	673
AutosarC++19_03-A18.0.1	673
AutosarC++19_03-A18.0.2	674
AutosarC++19_03-A18.0.3	674
AutosarC++19_03-A18.1.1	675
AutosarC++19_03-A18.1.2	675
AutosarC++19_03-A18.1.3	676
AutosarC++19_03-A18.1.4	676
AutosarC++19_03-A18.1.6	676
AutosarC++19_03-A18.5.1	677
AutosarC++19_03-A18.5.3	677
AutosarC++19_03-A18.5.4	677
AutosarC++19_03-A18.5.8	678
AutosarC++19_03-A18.9.1	678
AutosarC++19_03-A18.9.2	679
AutosarC++19_03-A18.9.3	679
AutosarC++19_03-A21.8.1	680
AutosarC++19_03-A23.0.1	680
AutosarC++19_03-A26.5.1	680
AutosarC++19_03-A26.5.2	681
AutosarC++19_03-A27.0.4	681
AutosarC++19_03-M0.1.1	682
AutosarC++19_03-M0.1.2	682
AutosarC++19_03-M0.1.3	683
AutosarC++19_03-M0.1.4	684
AutosarC++19_03-M0.1.8	684
AutosarC++19_03-M0.1.9	684
AutosarC++19_03-M0.1.10	686
AutosarC++19_03-M0.2.1	687
AutosarC++19_03-M0.3.1	687
AutosarC++19_03-M0.3.2	690
AutosarC++19_03-M0.4.2	690
AutosarC++19_03-M2.7.1	691
AutosarC++19_03-M2.10.1	691
AutosarC++19_03-M2.13.2	691
AutosarC++19_03-M2.13.3	692
AutosarC++19_03-M2.13.4	692
AutosarC++19_03-M3.1.2	693
AutosarC++19_03-M3.2.1	693
AutosarC++19_03-M3.2.2	694
AutosarC++19_03-M3.2.3	694
AutosarC++19_03-M3.2.4	695
AutosarC++19_03-M3.3.2	695
AutosarC++19_03-M3.4.1	696
AutosarC++19_03-M3.9.1	696
AutosarC++19_03-M3.9.3	697
AutosarC++19_03-M4.5.1	697
AutosarC++19_03-M4.5.3	698
AutosarC++19_03-M4.10.1	698
AutosarC++19_03-M4.10.2	698

AutosarC++19_03-M5.0.2	699
AutosarC++19_03-M5.0.3	699
AutosarC++19_03-M5.0.4	700
AutosarC++19_03-M5.0.5	701
AutosarC++19_03-M5.0.6	702
AutosarC++19_03-M5.0.7	702
AutosarC++19_03-M5.0.8	703
AutosarC++19_03-M5.0.9	704
AutosarC++19_03-M5.0.10	704
AutosarC++19_03-M5.0.11	704
AutosarC++19_03-M5.0.12	705
AutosarC++19_03-M5.0.14	705
AutosarC++19_03-M5.0.15	705
AutosarC++19_03-M5.0.16	706
AutosarC++19_03-M5.0.17	706
AutosarC++19_03-M5.0.18	707
AutosarC++19_03-M5.0.20	707
AutosarC++19_03-M5.0.21	707
AutosarC++19_03-M5.2.2	708
AutosarC++19_03-M5.2.3	708
AutosarC++19_03-M5.2.6	708
AutosarC++19_03-M5.2.8	709
AutosarC++19_03-M5.2.9	710
AutosarC++19_03-M5.2.10	711
AutosarC++19_03-M5.2.11	711
AutosarC++19_03-M5.2.12	711
AutosarC++19_03-M5.3.1	712
AutosarC++19_03-M5.3.2	712
AutosarC++19_03-M5.3.3	712
AutosarC++19_03-M5.3.4	713
AutosarC++19_03-M5.8.1	713
AutosarC++19_03-M5.14.1	713
AutosarC++19_03-M5.17.1	714
AutosarC++19_03-M5.18.1	714
AutosarC++19_03-M5.19.1	715
AutosarC++19_03-M6.2.1	715
AutosarC++19_03-M6.2.2	715
AutosarC++19_03-M6.2.3	716
AutosarC++19_03-M6.3.1	716
AutosarC++19_03-M6.4.1	716
AutosarC++19_03-M6.4.2	717
AutosarC++19_03-M6.4.3	717
AutosarC++19_03-M6.4.4	719
AutosarC++19_03-M6.4.5	719
AutosarC++19_03-M6.4.6	720
AutosarC++19_03-M6.4.7	720
AutosarC++19_03-M6.5.2	721
AutosarC++19_03-M6.5.3	721
AutosarC++19_03-M6.5.4	721
AutosarC++19_03-M6.5.5	722
AutosarC++19_03-M6.5.6	722
AutosarC++19_03-M6.6.1	723
AutosarC++19_03-M6.6.2	723

AutosarC++19_03-M6.6.3	723
AutosarC++19_03-M7.1.2	724
AutosarC++19_03-M7.3.1	724
AutosarC++19_03-M7.3.2	724
AutosarC++19_03-M7.3.3	725
AutosarC++19_03-M7.3.4	725
AutosarC++19_03-M7.3.6	725
AutosarC++19_03-M7.4.1	726
AutosarC++19_03-M7.4.2	726
AutosarC++19_03-M7.4.3	726
AutosarC++19_03-M7.5.1	727
AutosarC++19_03-M7.5.2	727
AutosarC++19_03-M8.0.1	727
AutosarC++19_03-M8.3.1	728
AutosarC++19_03-M8.4.2	728
AutosarC++19_03-M8.4.4	729
AutosarC++19_03-M8.5.2	729
AutosarC++19_03-M9.3.1	729
AutosarC++19_03-M9.3.3	730
AutosarC++19_03-M9.6.1	730
AutosarC++19_03-M9.6.4	731
AutosarC++19_03-M10.1.1	731
AutosarC++19_03-M10.1.2	731
AutosarC++19_03-M10.1.3	732
AutosarC++19_03-M10.2.1	732
AutosarC++19_03-M10.3.3	732
AutosarC++19_03-M11.0.1	733
AutosarC++19_03-M12.1.1	733
AutosarC++19_03-M14.5.3	734
AutosarC++19_03-M14.6.1	734
AutosarC++19_03-M15.0.3	734
AutosarC++19_03-M15.1.1	735
AutosarC++19_03-M15.1.2	735
AutosarC++19_03-M15.1.3	735
AutosarC++19_03-M15.3.1	736
AutosarC++19_03-M15.3.3	736
AutosarC++19_03-M15.3.4	736
AutosarC++19_03-M15.3.6	737
AutosarC++19_03-M15.3.7	737
AutosarC++19_03-M16.0.1	738
AutosarC++19_03-M16.0.2	738
AutosarC++19_03-M16.0.5	738
AutosarC++19_03-M16.0.6	739
AutosarC++19_03-M16.0.7	739
AutosarC++19_03-M16.0.8	739
AutosarC++19_03-M16.1.1	740
AutosarC++19_03-M16.1.2	740
AutosarC++19_03-M16.2.3	740
AutosarC++19_03-M16.3.1	741
AutosarC++19_03-M16.3.2	741
AutosarC++19_03-M17.0.2	742
AutosarC++19_03-M17.0.3	742
AutosarC++19_03-M17.0.5	742

AutosarC++19_03-M18.0.3	743
AutosarC++19_03-M18.0.4	743
AutosarC++19_03-M18.0.5	744
AutosarC++19_03-M18.2.1	744
AutosarC++19_03-M18.7.1	745
AutosarC++19_03-M19.3.1	745
AutosarC++19_03-M27.0.1	746
Rules in Group C#	746
C#-BitfieldHasNoPlurals	746
C#-BracesForSingleStatementBodies	746
C#-CSharpCommentsConvention	747
C#-CSharpNamingConvention	747
C#-CSharpNoBackwardGotoJumps	747
C#-CSharpNoCatchAllExceptionsClause	748
C#-CSharpNoRefAndOutParameters	748
C#-CSharpReferencedLabelInSameBlockAsGoto	748
C#-EnumeratorRequiresInitialization	749
C#-FieldNotPrivateNorProtected	749
C#-InvalidDependencyPropertyDeclaration	749
C#-MethodShouldBeDeclaredStatic	749
C#-NoHidingOfBaseClassMethods	750
C#-VirtualCallInConstructor	750
Rules in Group CertC	750
CertC-PRE00	750
CertC-PRE01	753
CertC-PRE02	755
CertC-PRE03	756
CertC-PRE04	757
CertC-PRE05	759
CertC-PRE06	761
CertC-PRE07	762
CertC-PRE08	763
CertC-PRE09	765
CertC-PRE10	766
CertC-PRE11	768
CertC-PRE12	770
CertC-PRE13	771
CertC-PRE30	774
CertC-PRE31	775
CertC-PRE32	778
CertC-DCL00	779
CertC-DCL01	781
CertC-DCL02	783
CertC-DCL03	785
CertC-DCL04	787
CertC-DCL05	788
CertC-DCL06	790
CertC-DCL07	794
CertC-DCL09	796
CertC-DCL11	798
CertC-DCL12	800
CertC-DCL13	802
CertC-DCL15	804

CertC-DCL16	806
CertC-DCL18	807
CertC-DCL19	807
CertC-DCL20	809
CertC-DCL21	811
CertC-DCL23	813
CertC-DCL30	815
CertC-DCL31	817
CertC-DCL36	819
CertC-DCL37	821
CertC-DCL38	825
CertC-DCL39	827
CertC-DCL40	831
CertC-DCL41	834
CertC-EXP00	836
CertC-EXP02	837
CertC-EXP05	838
CertC-EXP07	841
CertC-EXP10	842
CertC-EXP12	844
CertC-EXP14	845
CertC-EXP15	846
CertC-EXP19	847
CertC-EXP20	849
CertC-EXP30	852
CertC-EXP32	855
CertC-EXP33	857
CertC-EXP34	862
CertC-EXP35	864
CertC-EXP36	866
CertC-EXP37	869
CertC-EXP40	873
CertC-EXP42	874
CertC-EXP44	876
CertC-EXP45	879
CertC-EXP46	881
CertC-EXP47	882
CertC-INT00	884
CertC-INT01	887
CertC-INT05	889
CertC-INT07	891
CertC-INT08	892
CertC-INT09	894
CertC-INT12	895
CertC-INT13	897
CertC-INT15	898
CertC-INT17	901
CertC-INT34	902
CertC-INT36	905
CertC-FLP02	907
CertC-FLP04	910
CertC-FLP06	912
CertC-FLP07	914

CertC-FLP30	915
CertC-FLP32	917
CertC-FLP37	922
CertC-ARR01	924
CertC-ARR02	925
CertC-ARR30	927
CertC-ARR36	932
CertC-ARR37	934
CertC-ARR39	936
CertC-STR04	938
CertC-STR05	939
CertC-STR07	941
CertC-STR09	943
CertC-STR10	944
CertC-STR11	945
CertC-STR30	947
CertC-STR31	949
CertC-STR32	956
CertC-STR34	959
CertC-STR37	962
CertC-STR38	963
CertC-MEM01	965
CertC-MEM02	966
CertC-MEM30	969
CertC-MEM31	972
CertC-MEM33	974
CertC-MEM34	977
CertC-MEM35	979
CertC-MEM36	981
CODE	982
OUTPUT	982
Compliant Solution	982
Compliant Solution (Windows)	983
Risk Assessment	983
Bibliography	983
CertC-ENV30	983
CertC-ENV32	986
CertC-ENV33	988
CertC-FI030	992
CertC-FI034	994
CertC-FI037	997
CertC-FI038	999
CertC-FI047	1000
CertC-SIG30	1003
CertC-SIG31	1009
CertC-SIG34	1013
CertC-SIG35	1015
CertC-ERR30	1016
CertC-ERR32	1020
CertC-ERR33	1023
CertC-ERR34	1033
CertC-CON32	1035
CertC-CON40	1039
CertC-MSC24	1041
CertC-MSC30	1045

CertC-MSC32	1046
CertC-MSC33	1049
CertC-MSC37	1050
CertC-POS30	1053
CertC-POS33	1055
CertC-POS34	1056
CertC-POS35	1058
CertC-POS36	1059
CertC-POS37	1062
CertC-POS39	1065
Portability Details	1066
Risk Assessment	1066
Bibliography	1066
CertC-POS47	1067
CertC-POS49	1070
CertC-POS54	1073
Rules in Group CertC++	1075
CertC---DCL30	1075
CertC---DCL39	1078
CertC---DCL40	1081
CertC---DCL50	1085
CertC---DCL51	1087
CertC---DCL52	1090
CertC---DCL55	1092
CertC---DCL57	1096
CertC---DCL58	1099
CertC---DCL59	1102
CertC---DCL60	1106
CertC---EXP34	1109
CertC---EXP35	1112
CertC---EXP36	1114
CertC---EXP37	1116
CertC---EXP42	1120
CertC---EXP45	1122
CertC---EXP46	1125
CertC---EXP47	1126
CertC---EXP50	1128
CertC---EXP52	1131
CertC---EXP55	1133
CertC---EXP59	1136
CertC---INT34	1138
CertC---INT36	1141
CertC---INT50	1143
CertC---ARR30	1145
CertC---ARR37	1150
CertC---ARR39	1152
CertC---CTR56	1154
CertC---STR30	1156
CertC---STR31	1159
CertC---STR32	1165
CertC---STR34	1169
CertC---STR37	1171
CertC---STR38	1173
CertC---MEM30	1175

CertC++-MEM31	1178
CertC++-MEM34	1180
CertC++-MEM35	1182
CertC++-MEM36	1184
CODE	1185
OUTPUT	1185
Compliant Solution	1185
Compliant Solution [Windows]	1186
Risk Assessment	1186
Bibliography	1186
CertC++-MEM50	1186
CertC++-MEM51	1190
CertC++-FI030	1195
CertC++-FI034	1198
CertC++-FI037	1201
CertC++-FI038	1202
CertC++-FI047	1203
CertC++-ERR30	1207
CertC++-ERR32	1211
CertC++-ERR33	1213
CertC++-ERR34	1224
CertC++-ERR51	1226
CertC++-ERR52	1228
CertC++-ERR53	1229
CertC++-ERR54	1231
CertC++-ERR55	1232
CertC++-ERR58	1234
CertC++-ERR61	1236
CertC++-ERR62	1238
CertC++-OOP50	1240
CertC++-OOP52	1242
CertC++-OOP53	1244
CertC++-CON40	1246
CertC++-CON52	1248
CertC++-ENV30	1252
CertC++-ENV32	1254
CertC++-ENV33	1256
CertC++-FLP30	1260
CertC++-FLP32	1262
CertC++-FLP37	1267
CertC++-MSC30	1269
CertC++-MSC32	1270
CertC++-MSC33	1272
CertC++-MSC37	1274
CertC++-MSC50	1277
CertC++-MSC51	1278
CertC++-MSC52	1280
CertC++-MSC53	1282
CertC++-PRE30	1283
CertC++-PRE31	1284
CertC++-PRE32	1287
CertC++-SIG31	1288
CertC++-SIG34	1292
CertC++-SIG35	1294
Rules in Group Expressions.Assignments	1295

Expressions.Assignments-StrongTypeViolations	1295
Rules in Group Generic	1297
Generic-AnsiStringUse	1297
Generic-BusyHeaders	1298
Generic-CComments	1298
Generic-CPPComments	1299
Generic-CapitalizeFunctions	1299
Generic-DoNotMixLogicalOperators	1300
Generic-DoxygenCommentAtDefinition	1300
Generic-DoxygenCommentInHeader	1301
Generic-DuplicateIncludeGuard	1301
Generic-Filemarker	1302
Generic-FORBIDDENFUNCTIONS	1302
Generic-FORBIDDENMACROS	1302
Generic-FORBIDDENTOKENS	1303
Generic-FormatSpecifier	1303
Generic-IncludeKind	1304
Generic-InitializeAllFieldsInConstructor	1304
Generic-InitializeAllVariables	1305
Generic-LineBreaks	1305
Generic-LocalInclude	1306
Generic-LocalScope	1308
Generic-MaxComplexity	1309
Generic-MaxConditions	1309
Generic-MaxNesting	1310
Generic-MaxOneStmtPerLine	1310
Generic-MaxParams	1310
Generic-MissingConstructor	1311
Generic-MissingDestructor	1311
Generic-MissingIncludeGuard	1311
Generic-MissingInlineDefinition	1312
Generic-MissingOverride	1312
Generic-MissingParameterAssert	1313
Generic-MissingSelfHeaderInclude	1313
Generic-NamingConvention	1313
Generic-NoAbsoluteInclude	1314
Generic-NoAssembler	1314
Generic-NoAutoType	1315
Generic-NoCCasts	1315
Generic-NoCFunctionCall	1315
Generic-NoCHeaderInclude	1316
Generic-NoCPPStructs	1316
Generic-NoCharPointer	1316
Generic-NoCommaSequence	1317
Generic-NoCompilerWarnings	1317
Generic-NoConditionalOperator	1317
Generic-NoConstCasts	1318
Generic-NoConstOnRHS	1318
Generic-NoDebugMacro	1318
Generic-NoDiamondInheritance	1319
Generic-NoDiscardedReturnCode	1319
Generic-NoDoubleUnderscoreInMacro	1320
Generic-NoEllipsis	1320

Generic-NoEmptyLoops	1320
Generic-NoEmptyStructs	1321
Generic-NoExternInImpl	1321
Generic-NoFriendClass	1321
Generic-NoFunctionCommentInImpl	1322
Generic-NoFunctionDefinitionInHeader	1322
Generic-NoFunctionMacroInvocation	1322
Generic-NoFunctionPrototypeInImpl	1323
Generic-NoIfdefInHeader	1323
Generic-NoImplicitTypeConversion	1323
Generic-NoIncludePaths	1324
Generic-NoIrregularInclude	1324
Generic-NoLeakingReferenceToLocal	1324
Generic-NoLinkerWarnings	1325
Generic-NoMagicNumbers	1325
Generic-NoMalloc	1326
Generic-NoMixOfPtrAndIntArithmetic	1326
Generic-NoMultipleInheritance	1327
Generic-NoNewWithArrays	1327
Generic-NoOverloadedOperators	1328
Generic-NoPrecisionLoss	1328
Generic-NoPublicDataMembers	1328
Generic-NoReferenceToLocalVariable	1329
Generic-NoReferenceToPrivateDataMember	1330
Generic-NoSemicolonAtEndOfMacro	1330
Generic-NoSignedDivision	1331
Generic-NoSingleCharIdentifier	1331
Generic-NoStaticInHeader	1331
Generic-NoStdStringInternals	1332
Generic-NoTabs	1332
Generic-NoTrailingWhitespace	1332
Generic-NoTypeConversionToBool	1333
Generic-NoUncheckedMalloc	1333
Generic-NoUncheckedPointerParamDereference	1334
Generic-NoUnnamedNamespaceInHeader	1334
Generic-NoUnsafeMacro	1334
Generic-NoUsingNamespaceInHeader	1335
Generic-NoVirtualDestructor	1335
Generic-NoVirtualInheritance	1335
Generic-NoWhitespaceMemberSelection	1336
Generic-NoWhitespaceUnaryOperator	1336
Generic-PCHIncludes	1336
Generic-RuleOfThree	1337
Generic-Templates	1338
Generic-ThrowByValueCatchByReference	1338
Generic-TooManyIncludes	1339
Generic-TypedefCheck	1339
Generic-WrongIncludeCasing	1339
Rules in Group Metric	1340
Metric-Calling.CPP	1340
Metric-Clone_Ratio	1340
Metric-Comment.Density	1341
Metric-Comment.Internal	1342

Metric-Comment.Preceding	1342
Metric-Coupling	1343
Metric-Extended_M McCabe	1344
Metric-Fan.In	1345
Metric-Fan.Out	1345
Metric-HIS.AP(CG)_CYCLE	1346
Metric-HIS.CALLING	1347
Metric-HIS.CALLS	1348
Metric-HIS.COMF	1349
Metric-HIS.GOTO	1350
Metric-HIS.LEVEL	1351
Metric-HIS.PARAM	1352
Metric-HIS.PATH	1353
Metric-HIS.RETURN	1354
Metric-HIS.STMT	1355
Metric-HIS.VG	1356
Metric-HIS.VOCF	1357
Metric-Halstead.Different_Operands	1358
Metric-Halstead.Different_Operators	1359
Metric-Halstead.Difficulty	1360
Metric-Halstead.Effort	1361
Metric-Halstead.Length	1362
Metric-Halstead.Total_Operands	1363
Metric-Halstead.Total_Operators	1364
Metric-Halstead.Vocabulary	1365
Metric-Halstead.Vocabulary_Frequency	1366
Metric-Halstead.Volume	1367
Metric-Includes.Direct_Includers	1368
Metric-Includes.Direct_Includes	1368
Metric-Includes.Include_Burden	1369
Metric-Includes.Maximum_Include_Depth	1370
Metric-Includes.Maximum_Includer_Depth	1370
Metric-Includes.Transitive_Includers	1371
Metric-Includes.Transitive_Includes	1371
Metric-Inverse_Coupling	1372
Metric-LOC	1373
Metric-Lines.Class.Code	1373
Metric-Lines.Class.Comment	1374
Metric-Lines.Class.Comment_Ratio	1374
Metric-Lines.Class.LOC	1375
Metric-Lines.Code	1375
Metric-Lines.Comment	1376
Metric-Lines.Empty	1377
Metric-Lines.File.Code	1377
Metric-Lines.File.Comment	1378
Metric-Lines.File.Comment_Ratio	1379
Metric-Lines.File.LOC	1379
Metric-Lines.LOC	1380
Metric-Lines.Only_Comment	1381
Metric-Lines.PP_Define	1382
Metric-Lines.PP_Elif	1382
Metric-Lines.PP_Else	1383
Metric-Lines.PP_Endif	1383

Metric-Lines.PP_Error	1384
Metric-Lines.PP_Ident	1385
Metric-Lines.PP_If	1385
Metric-Lines.PP_Indef	1386
Metric-Lines.PP_Indef	1386
Metric-Lines.PP_Include	1387
Metric-Lines.PP_Line	1388
Metric-Lines.PP_Pragma	1388
Metric-Lines.PP_Sharp	1389
Metric-Lines.PP_Sharp_Sharp	1389
Metric-Lines.PP_Undef	1390
Metric-Lines.PP_Warning	1390
Metric-Lines.Preproc	1391
Metric-Lines.Routine.Code	1392
Metric-Lines.Routine.Comment	1392
Metric-Lines.Routine.Comment_Ratio	1393
Metric-Lines.Routine.Comment_Ratio.LOC_Qualified	1393
Metric-Lines.Routine.LOC	1394
Metric-LogNPath_Ceiling	1394
Metric-LogNPath_Floor	1395
Metric-MI_Per_Routine	1396
Metric-Maximum_Extended_Nesting	1396
Metric-Maximum_Nesting	1397
Metric-McCabe_Complexity	1398
Metric-NPath	1399
Metric-Number_Of_Called_Routines	1400
Metric-Number_Of_Calling_Routines	1401
Metric-Number_Of_Gotos	1401
Metric-Number_Of_Invocations	1402
Metric-Number_Of_Parameters	1402
Metric-Number_Of_Returns	1403
Metric-Number_Of(SCCs)	1404
Metric-Number_Of_Statements	1404
Metric-OO.CBO	1405
Metric-OO.CBOa	1406
Metric-OO.CB0s	1407
Metric-OO.DAC	1408
Metric-OO.DIT	1409
Metric-OO.LCOM	1410
Metric-OO.LCOMs	1411
Metric-OO.NIVOC	1412
Metric-OO.NOA	1413
Metric-OO.NOCC	1414
Metric-OO.NOFA	1414
Metric-OO.NOMA	1415
Metric-OO.NOMO	1416
Metric-OO.NOPC	1416
Metric-OO.RFC	1417
Metric-OO.WMC.McCabe_Complexity	1418
Metric-OO.WMC.One	1419
Metric-Switch_Complexity	1420
Metric-TokenFileMetric	1421
Metric-TokenMetric	1422

Rules in Group MisraC	1423
MisraC-1.1	1423
MisraC-1.2	1425
MisraC-2.1	1425
MisraC-2.2	1426
MisraC-2.3	1426
MisraC-2.4	1426
MisraC-3.1	1427
MisraC-3.4	1428
MisraC-4.1	1429
MisraC-4.2	1429
MisraC-5.1	1429
MisraC-5.2	1430
MisraC-5.3	1431
MisraC-5.4	1431
MisraC-5.5	1432
MisraC-5.6	1432
MisraC-5.7	1433
MisraC-6.1	1433
MisraC-6.2	1433
MisraC-6.3	1433
MisraC-6.4	1434
MisraC-6.5	1434
MisraC-7.1	1434
MisraC-8.1	1435
MisraC-8.2	1435
MisraC-8.3	1435
MisraC-8.4	1436
MisraC-8.5	1436
MisraC-8.6	1437
MisraC-8.7	1437
MisraC-8.8	1437
MisraC-8.9	1438
MisraC-8.10	1438
MisraC-8.11	1439
MisraC-8.12	1439
MisraC-9.1	1439
MisraC-9.2	1440
MisraC-9.3	1440
MisraC-10.1	1440
MisraC-10.2	1441
MisraC-10.3	1442
MisraC-10.4	1443
MisraC-10.5	1444
MisraC-10.6	1444
MisraC-11.1	1445
MisraC-11.2	1446
MisraC-11.3	1446
MisraC-11.4	1447
MisraC-11.5	1447
MisraC-12.1	1448
MisraC-12.2	1448
MisraC-12.3	1449

MisraC-12.4	1449
MisraC-12.5	1449
MisraC-12.6	1450
MisraC-12.7	1450
MisraC-12.8	1450
MisraC-12.9	1451
MisraC-12.10	1451
MisraC-12.11	1451
MisraC-12.12	1451
MisraC-12.13	1452
MisraC-13.1	1452
MisraC-13.2	1452
MisraC-13.3	1452
MisraC-13.4	1453
MisraC-13.5	1453
MisraC-13.6	1453
MisraC-13.7	1453
MisraC-14.1	1454
MisraC-14.2	1454
MisraC-14.3	1455
MisraC-14.4	1455
MisraC-14.5	1455
MisraC-14.6	1456
MisraC-14.7	1456
MisraC-14.8	1456
MisraC-14.9	1456
MisraC-14.10	1457
MisraC-15.0	1457
MisraC-15.1	1459
MisraC-15.2	1459
MisraC-15.3	1459
MisraC-15.4	1460
MisraC-15.5	1460
MisraC-16.1	1460
MisraC-16.2	1460
MisraC-16.3	1461
MisraC-16.4	1461
MisraC-16.5	1461
MisraC-16.6	1462
MisraC-16.7	1462
MisraC-16.8	1462
MisraC-16.9	1463
MisraC-16.10	1463
MisraC-17.1	1463
MisraC-17.2	1464
MisraC-17.3	1464
MisraC-17.4	1464
MisraC-17.5	1465
MisraC-17.6	1465
MisraC-18.1	1465
MisraC-18.2	1465
MisraC-18.4	1466
MisraC-19.1	1466

MisraC-19.2	1466
MisraC-19.3	1467
MisraC-19.4	1467
MisraC-19.5	1467
MisraC-19.6	1467
MisraC-19.7	1468
MisraC-19.8	1468
MisraC-19.9	1468
MisraC-19.10	1468
MisraC-19.11	1469
MisraC-19.12	1469
MisraC-19.13	1469
MisraC-19.14	1469
MisraC-19.15	1470
MisraC-19.16	1470
MisraC-19.17	1470
MisraC-20.1	1471
MisraC-20.2	1471
MisraC-20.3	1472
MisraC-20.4	1472
MisraC-20.5	1472
MisraC-20.6	1473
MisraC-20.7	1473
MisraC-20.8	1473
MisraC-20.9	1474
MisraC-20.10	1474
MisraC-20.11	1474
MisraC-20.12	1475
MisraC-21.1	1475
Rules in Group MisraC++	1478
MisraC++-0.1.1	1478
MisraC++-0.1.2	1478
MisraC++-0.1.3	1479
MisraC++-0.1.4	1479
MisraC++-0.1.5	1480
MisraC++-0.1.6	1480
MisraC++-0.1.7	1480
MisraC++-0.1.8	1481
MisraC++-0.1.9	1481
MisraC++-0.1.10	1482
MisraC++-0.1.11	1483
MisraC++-0.1.12	1483
MisraC++-0.2.1	1483
MisraC++-0.3.1	1484
MisraC++-0.3.2	1486
MisraC++-0.4.2	1486
MisraC++-1.0.1	1487
MisraC++-2.3.1	1488
MisraC++-2.5.1	1488
MisraC++-2.7.1	1489
MisraC++-2.7.2	1489
MisraC++-2.7.3	1489
MisraC++-2.10.1	1490

MisraC++-2.10.2	1490
MisraC++-2.10.3	1491
MisraC++-2.10.4	1491
MisraC++-2.10.5	1492
MisraC++-2.10.6	1492
MisraC++-2.13.1	1492
MisraC++-2.13.2	1493
MisraC++-2.13.3	1493
MisraC++-2.13.4	1493
MisraC++-2.13.5	1494
MisraC++-3.1.1	1494
MisraC++-3.1.2	1494
MisraC++-3.1.3	1494
MisraC++-3.2.1	1495
MisraC++-3.2.2	1495
MisraC++-3.2.3	1496
MisraC++-3.2.4	1496
MisraC++-3.3.1	1497
MisraC++-3.3.2	1497
MisraC++-3.4.1	1497
MisraC++-3.9.1	1498
MisraC++-3.9.2	1498
MisraC++-3.9.3	1499
MisraC++-4.5.1	1499
MisraC++-4.5.2	1499
MisraC++-4.5.3	1500
MisraC++-4.10.1	1500
MisraC++-4.10.2	1500
MisraC++-5.0.1	1500
MisraC++-5.0.2	1501
MisraC++-5.0.3	1501
MisraC++-5.0.4	1502
MisraC++-5.0.5	1502
MisraC++-5.0.6	1503
MisraC++-5.0.7	1503
MisraC++-5.0.8	1504
MisraC++-5.0.9	1505
MisraC++-5.0.10	1505
MisraC++-5.0.11	1505
MisraC++-5.0.12	1505
MisraC++-5.0.13	1506
MisraC++-5.0.14	1506
MisraC++-5.0.15	1506
MisraC++-5.0.16	1507
MisraC++-5.0.17	1507
MisraC++-5.0.18	1507
MisraC++-5.0.19	1508
MisraC++-5.0.20	1508
MisraC++-5.0.21	1508
MisraC++-5.2.1	1508
MisraC++-5.2.2	1509
MisraC++-5.2.3	1509
MisraC++-5.2.4	1509

MisraC++-5.2.5	1510
MisraC++-5.2.6	1510
MisraC++-5.2.7	1511
MisraC++-5.2.8	1511
MisraC++-5.2.9	1512
MisraC++-5.2.10	1512
MisraC++-5.2.11	1512
MisraC++-5.2.12	1513
MisraC++-5.3.1	1513
MisraC++-5.3.2	1513
MisraC++-5.3.3	1513
MisraC++-5.3.4	1514
MisraC++-5.8.1	1514
MisraC++-5.14.1	1514
MisraC++-5.17.1	1515
MisraC++-5.18.1	1515
MisraC++-5.19.1	1515
MisraC++-6.2.1	1516
MisraC++-6.2.2	1516
MisraC++-6.2.3	1516
MisraC++-6.3.1	1516
MisraC++-6.4.1	1517
MisraC++-6.4.2	1517
MisraC++-6.4.3	1517
MisraC++-6.4.4	1519
MisraC++-6.4.5	1519
MisraC++-6.4.6	1519
MisraC++-6.4.7	1520
MisraC++-6.4.8	1520
MisraC++-6.5.1	1520
MisraC++-6.5.2	1521
MisraC++-6.5.3	1521
MisraC++-6.5.4	1521
MisraC++-6.5.5	1521
MisraC++-6.5.6	1522
MisraC++-6.6.1	1522
MisraC++-6.6.2	1522
MisraC++-6.6.3	1523
MisraC++-6.6.4	1523
MisraC++-6.6.5	1523
MisraC++-7.1.1	1523
MisraC++-7.1.2	1524
MisraC++-7.2.1	1524
MisraC++-7.3.1	1525
MisraC++-7.3.2	1525
MisraC++-7.3.3	1525
MisraC++-7.3.4	1525
MisraC++-7.3.5	1525
MisraC++-7.3.6	1526
MisraC++-7.4.1	1526
MisraC++-7.4.2	1526
MisraC++-7.4.3	1527
MisraC++-7.5.1	1527

MisraC++-7.5.2	1527
MisraC++-7.5.3	1527
MisraC++-7.5.4	1528
MisraC++-8.0.1	1528
MisraC++-8.3.1	1528
MisraC++-8.4.1	1529
MisraC++-8.4.2	1529
MisraC++-8.4.3	1529
MisraC++-8.4.4	1529
MisraC++-8.5.1	1530
MisraC++-8.5.2	1530
MisraC++-8.5.3	1531
MisraC++-9.3.1	1531
MisraC++-9.3.2	1531
MisraC++-9.3.3	1532
MisraC++-9.5.1	1532
MisraC++-9.6.1	1532
MisraC++-9.6.2	1533
MisraC++-9.6.3	1533
MisraC++-9.6.4	1533
MisraC++-10.1.1	1533
MisraC++-10.1.2	1534
MisraC++-10.1.3	1534
MisraC++-10.2.1	1534
MisraC++-10.3.1	1534
MisraC++-10.3.2	1535
MisraC++-10.3.3	1535
MisraC++-11.0.1	1535
MisraC++-12.1.1	1536
MisraC++-12.1.2	1536
MisraC++-12.1.3	1536
MisraC++-12.8.1	1536
MisraC++-12.8.2	1537
MisraC++-14.5.1	1537
MisraC++-14.5.2	1537
MisraC++-14.5.3	1538
MisraC++-14.6.1	1538
MisraC++-14.6.2	1538
MisraC++-14.7.1	1538
MisraC++-14.7.2	1539
MisraC++-14.7.3	1539
MisraC++-14.8.1	1539
MisraC++-14.8.2	1539
MisraC++-15.0.1	1540
MisraC++-15.0.2	1540
MisraC++-15.0.3	1540
MisraC++-15.1.1	1540
MisraC++-15.1.2	1541
MisraC++-15.1.3	1541
MisraC++-15.3.1	1541
MisraC++-15.3.2	1542
MisraC++-15.3.3	1542
MisraC++-15.3.4	1542

MisraC++-15.3.5	1543
MisraC++-15.3.6	1543
MisraC++-15.3.7	1543
MisraC++-15.4.1	1543
MisraC++-15.5.1	1544
MisraC++-15.5.2	1544
MisraC++-15.5.3	1544
MisraC++-16.0.1	1545
MisraC++-16.0.2	1546
MisraC++-16.0.3	1546
MisraC++-16.0.4	1546
MisraC++-16.0.5	1546
MisraC++-16.0.6	1547
MisraC++-16.0.7	1547
MisraC++-16.0.8	1547
MisraC++-16.1.1	1547
MisraC++-16.1.2	1548
MisraC++-16.2.1	1548
MisraC++-16.2.2	1548
MisraC++-16.2.3	1549
MisraC++-16.2.4	1549
MisraC++-16.2.5	1549
MisraC++-16.2.6	1550
MisraC++-16.3.1	1550
MisraC++-16.3.2	1550
MisraC++-16.6.1	1550
MisraC++-17.0.1	1551
MisraC++-17.0.2	1551
MisraC++-17.0.3	1551
MisraC++-17.0.5	1552
MisraC++-18.0.1	1552
MisraC++-18.0.2	1552
MisraC++-18.0.3	1553
MisraC++-18.0.4	1553
MisraC++-18.0.5	1553
MisraC++-18.2.1	1554
MisraC++-18.4.1	1554
MisraC++-18.7.1	1554
MisraC++-19.3.1	1555
MisraC++-27.0.1	1555
Rules in Group MisraC2012	1555
MisraC2012-1.1	1556
MisraC2012-1.2	1559
MisraC2012-1.3	1560
MisraC2012-2.1	1561
MisraC2012-2.2	1561
MisraC2012-2.3	1563
MisraC2012-2.4	1563
MisraC2012-2.5	1564
MisraC2012-2.6	1564
MisraC2012-2.7	1564
MisraC2012-3.1	1565
MisraC2012-3.2	1565

MisraC2012-4.1	1565
MisraC2012-4.2	1565
MisraC2012-5.1	1566
MisraC2012-5.2	1566
MisraC2012-5.3	1566
MisraC2012-5.4	1567
MisraC2012-5.5	1568
MisraC2012-5.6	1568
MisraC2012-5.7	1568
MisraC2012-5.8	1569
MisraC2012-5.9	1569
MisraC2012-6.1	1569
MisraC2012-6.2	1570
MisraC2012-7.1	1570
MisraC2012-7.2	1570
MisraC2012-7.3	1571
MisraC2012-7.4	1571
MisraC2012-8.1	1571
MisraC2012-8.2	1571
MisraC2012-8.3	1572
MisraC2012-8.4	1572
MisraC2012-8.5	1573
MisraC2012-8.6	1573
MisraC2012-8.7	1573
MisraC2012-8.8	1574
MisraC2012-8.9	1574
MisraC2012-8.10	1575
MisraC2012-8.11	1575
MisraC2012-8.12	1575
MisraC2012-8.13	1575
MisraC2012-8.14	1576
MisraC2012-9.1	1576
MisraC2012-9.2	1576
MisraC2012-9.3	1577
MisraC2012-9.4	1577
MisraC2012-9.5	1577
MisraC2012-10.1	1578
MisraC2012-10.2	1578
MisraC2012-10.3	1578
MisraC2012-10.4	1578
MisraC2012-10.5	1579
MisraC2012-10.6	1580
MisraC2012-10.7	1580
MisraC2012-10.8	1580
MisraC2012-11.1	1581
MisraC2012-11.2	1582
MisraC2012-11.3	1583
MisraC2012-11.4	1584
MisraC2012-11.5	1584
MisraC2012-11.6	1585
MisraC2012-11.7	1585
MisraC2012-11.8	1586
MisraC2012-11.9	1586

MisraC2012-12.1	1587
MisraC2012-12.2	1587
MisraC2012-12.3	1587
MisraC2012-12.4	1587
MisraC2012-12.5	1588
MisraC2012-13.1	1588
MisraC2012-13.2	1588
MisraC2012-13.3	1589
MisraC2012-13.4	1589
MisraC2012-13.5	1589
MisraC2012-13.6	1590
MisraC2012-14.1	1590
MisraC2012-14.2	1590
MisraC2012-14.3	1591
MisraC2012-14.4	1592
MisraC2012-15.1	1592
MisraC2012-15.2	1593
MisraC2012-15.3	1593
MisraC2012-15.4	1593
MisraC2012-15.5	1593
MisraC2012-15.6	1594
MisraC2012-15.7	1594
MisraC2012-16.1	1594
MisraC2012-16.2	1595
MisraC2012-16.3	1596
MisraC2012-16.4	1596
MisraC2012-16.5	1597
MisraC2012-16.6	1597
MisraC2012-16.7	1597
MisraC2012-17.1	1597
MisraC2012-17.2	1598
MisraC2012-17.3	1598
MisraC2012-17.4	1598
MisraC2012-17.5	1599
MisraC2012-17.6	1599
MisraC2012-17.7	1599
MisraC2012-17.8	1600
MisraC2012-18.1	1600
MisraC2012-18.2	1600
MisraC2012-18.3	1600
MisraC2012-18.4	1601
MisraC2012-18.5	1601
MisraC2012-18.6	1601
MisraC2012-18.7	1602
MisraC2012-18.8	1602
MisraC2012-19.1	1602
MisraC2012-19.2	1602
MisraC2012-20.1	1603
MisraC2012-20.2	1603
MisraC2012-20.3	1603
MisraC2012-20.4	1603
MisraC2012-20.5	1604
MisraC2012-20.6	1604

MisraC2012-20.7	1604
MisraC2012-20.8	1604
MisraC2012-20.9	1605
MisraC2012-20.10	1605
MisraC2012-20.11	1605
MisraC2012-20.12	1605
MisraC2012-20.13	1606
MisraC2012-20.14	1606
MisraC2012-21.1	1606
MisraC2012-21.2	1607
MisraC2012-21.3	1607
MisraC2012-21.4	1608
MisraC2012-21.5	1608
MisraC2012-21.6	1608
MisraC2012-21.7	1609
MisraC2012-21.8	1609
MisraC2012-21.9	1610
MisraC2012-21.10	1610
MisraC2012-21.11	1611
MisraC2012-21.12	1611
MisraC2012-21.14	1612
MisraC2012-21.15	1612
MisraC2012-21.16	1612
MisraC2012-21.19	1613
MisraC2012-21.20	1613
MisraC2012-22.1	1614
MisraC2012-22.2	1614
MisraC2012-22.3	1614
MisraC2012-22.4	1615
MisraC2012-22.5	1615
MisraC2012-22.6	1615
MisraC2012-22.7	1616
MisraC2012-22.8	1616
MisraC2012-22.9	1616
MisraC2012-22.10	1617
Rules in Group MisraC2012Directive	1617
MisraC2012Directive-1.1	1617
MisraC2012Directive-2.1	1618
MisraC2012Directive-4.1	1619
MisraC2012Directive-4.2	1621
MisraC2012Directive-4.3	1621
MisraC2012Directive-4.4	1622
MisraC2012Directive-4.5	1622
MisraC2012Directive-4.6	1623
MisraC2012Directive-4.7	1623
MisraC2012Directive-4.8	1623
MisraC2012Directive-4.9	1624
MisraC2012Directive-4.10	1624
MisraC2012Directive-4.11	1624
MisraC2012Directive-4.12	1625
MisraC2012Directive-4.13	1625
Rules in Group Parallelism	1625
Parallelism-IncorrectCriticalSection	1625

Parallelism-UnsafeVarAccess	1626
Rules in Group Style.Indentation	1627
Style.Indentation-AllmanBraces	1627
Style.Indentation-WhitesmithBraces	1627
Rules in Group Style.Metrics	1628
Style.Metrics-MaxComplexity	1628
Style.Metrics-MaxConditions	1628
Style.Metrics-MaxNesting	1629
Style.Metrics-MaxOneStmtPerLine	1629
Style.Metrics-MaxParams	1630
Style.Metrics-MaximumLineLength	1630
Style.Metrics-TooManyIncludes	1630
Rules in Group Style.Naming	1631
Style.Naming-CSharpNamingConvention	1631
Style.Naming-CapitalizeFunctions	1631
Style.Naming-FileExtensionNaming	1631
Style.Naming-FilenameNaming	1632
Style.Naming-NamingConvention	1632
Style.Naming-NoDoubleUnderscoreInMacro	1633
Style.Naming-NoSingleCharIdentifier	1633
Style.Naming-SourceFileNaming	1633
Rules in Group Style.Parens	1634
Style.Parens-MissingLogicalOperandParens	1634
Style.Parens-MissingParens	1634
Style.Parens-MissingParensForPrecedenceLevel	1634
Style.Parens-ParensDuplicatingAlgebraicOrder	1635
Style.Parens-SizeofMissingParens	1635
Style.Parens-SuperfluousRHSParens	1635
Style.Parens-SuperfluousUnaryOperatorParens	1635
Rules in Group Style.Whitespace	1636
Style.Whitespace-LineBreaks	1636
Style.Whitespace-NoBlankLinesAtBraces	1636
Style.Whitespace-NoTabs	1636
Style.Whitespace-NoTrailingWhitespace	1637
Style.Whitespace-NoWhitespaceMemberSelection	1637
Style.Whitespace-NoWhitespacePointerReference	1637
Style.Whitespace-NoWhitespaceUnaryOperator	1638
Style.Whitespace-WhitespaceNextToOperator	1638

Axivion Bauhaus Suite Stylecheck Documentation

This file is automatically generated to document the available rules in the Axivion Bauhaus Suite standard installation for the stylecheck tool.

Common Rule Configuration

These options are available for all rules. Modifying such an option for a rule does not affect other rules. The individual rules can have pre-configured default values differing from the general ones shown here.

Name	Explanation	Value
additional_rulehtml	Additional HTML snippet that is appended to the rule explanation shown in the dashboard.	
apply_to_unit	Whether check should be enforced for single units as inputs as well.	False
disabled	Allows disabling of individual rules.	False
exclude_in_macro	If set to true, violations with primary SLoc inside a macro invocation will not be reported.	False
exclude_in_macros	Macro name patterns for macros where this rule should not be applied.	set([])
exclude_messages_in_system_headers	Whether to exclude files marked as system headers. The default value of this option can be controlled with stylecheck command line arguments.	True
excludes	Filename patterns for files to exclude from all rules (via command-line).	set([])
extend_exclude_to_macro_invocations	When True, messages inside macro invocations will not be reported if the macro is from an excluded file.	False
includes	Filename patterns to restrict all rules to (via command-line). If given, the rules only check files matching this pattern, and this set of files can be further reduced with the exclude patterns	set([])
indicate_macro_context	Whether some macro context should be printed in addition to normal position. (Deprecated; only affects output on command-line)	False
individual_excludes	Filename patterns for files to exclude from the individual rule.	set([])
individual_includes	Filename patterns to restrict this individual rule to. Maybe combined with the global rules.includes (the rule then is applied to more locations than the global include patterns) and with the exclude patterns.	set([])
justification_checker	Can be set to a callable classmethod taking (and possibly modifying) a style violation instance. Will be called per issue and can be used to check a disabled issue's justification, for example, against permitted ones.	None
languages	Source languages the rule applies to.	['C', 'C++', 'C++/CLI']
msg	Static description of messages possibly generated by this rule. See the Reference Guide chapter on stylecheck configuration for examples on how this can be used to, for example, disable or change the text of individual kinds of messages.	dict(...)
provider	Provider of this check, appears in the dashboard.	axivion
report_at_macro_invocation	Select position to use in case there is a macro context: Macro definition or macro invocation.	True
rulename	Unique name for the rule.	None
rulesortkey	A primary sorting key per rule/group.	1
ruletext	Short rule title.	
severities	Maps message-keys to severities; can be used to assign different severities to different messages	dict(...)
severity	Severity to classify violations, appears in the dashboard.	error

Global Configuration

These options are globally available (in CONFIG) and affect multiple rules.

Name	Explanation	Value

Additional_Basic_Characters	Regular expression string for additional characters that should be treated as basic characters.	
Allow_Single_Bitfield_As_Bool	If true, bitfields of size "1 bit" are effectively bool	False
Base_View_Name	Name of the base view used to calculate the metric values. If None, a default value is used. This setting can be overridden per rule.	None
Effect_Free_Functions	Names of functions that should be assumed to have no side effects. This causes the analysis to ignore the body of the function. For C++, use qualified names without template arguments, e.g. "std::numeric_limits::max"	[]
Effect_Free_Volatile_Variables	Names of variables that can be read without side-effect despite being volatile. If this list contains a macro name, any volatile reads within the macro expansion are treated as free of side effects; as well as any volatile variables declared within the macro expansion. This can be used for memory-mapped IO when the hardware guarantees that reads have no side effect. Write accesses continue to be seen as side-effect.	[]
Enums_As_Integer_Constants	List of enum type name patterns which should be seen as collection of integer constants.	[]
External_Functions_Returning_Bool	Set of qualified function names for external functions which should be seen as returning bool	set([])
Fixed_Size_Int_Typedefs	List of regular expression strings for names of typedefs for fixed-size integers, e.g. 'char(\d)*_t\$', 'u?int\d+_t\$', 'float\d+_t\$'	[]
Hierarchy_Edge_Name	Name of the edge used to traverse the hierarchy view.	Belongs_To
Hierarchy_View_Name	Name of the hierarchy view used to propagate the metric values. If None, a default value is used. This setting can be overridden per rule.	None
IRAnalysis_Allow_Function_Pointer_Signature_Mismatch	Whether function pointer targets should be allowed or filtered out if their signature does not match the function pointer use in an indirect call.	False
IRAnalysis_Assume_Globals_Are_Initialized	Whether global and local static variables should be treated as initialized for the uninit check.	False
IRAnalysis_Contexts	When iranalysis-based checks are executed: number of calling contexts to distinguish	4
IRAnalysis_Local_Mode	When iranalysis-based checks are executed: Perform checks by analysing each routine in isolation.	False
IRAnalysis_Safe_Mode	When iranalysis-based checks are executed: Skip unsound heuristics that are used to exclude findings.	False
IRAnalysis_Uses_UncalledEntries	When iranalysis-based checks are executed: whether defined but uncalled functions should be treated as additional entry points into the callgraph	False
IRAnalysis_With_AbstralR	When iranalysis-based checks are executed: use abstract interpretation module as additional postprocessing step. This option enables all abstract-interpretation-based analyses: symbolic expression analysis for div-by-zero and overflow checks as well as additional "null analysis" useful for null dereferences and div-by-zero. If you do not wish to activate all additional abstract-interpretation-based analyses but only a subset thereof, you can use the config setting IRAnalysis_With_AbstralR_Null and configuration options of individual stylecheck rules like NoNullPointerDereferenceRule and NoIntegerOverflowRule to activate the analyses selectively.	False
IRAnalysis_With_AbstralR_Null	When iranalysis-based checks are executed: use abstract-interpretation-based "null analysis" as additional analysis step (can reduce the number of findings in particular for null dereferences and div-by-zero). This switch is automatically enabled if IRAnalysis_With_AbstralR is set to True.	False
Ignore_Tokens_In_Disabled_Sections	Whether tokens in #if 0 sections should be ignored.	True
Local_Effect_Functions	Names of functions for which side effects should not be propagated up the call graph. Calls to these functions can be seen as having an effect within the caller function (thus there is no "Statement has no side effect" violation when calling these functions); but the overall effect of the caller function will exclude any effects from the specified function. This allows marking of non-persistent side effects. For C++, use qualified names without template arguments, e.g. "std::__assert_fail"	[]

Misra_Essential_Types_Treat_Char_As_Integer	If enabled, 'char' will be treated as essentially signed/unsigned type (depending on the compiler configuration for 'char'). This effectively disables the 'essentially character' category, and allows implicit conversions between 'char' and integer types of the same sign. Affects the MisraC2012-10.* rules.	False
Mixed_Headers_Are_C	Whether header files being included from both C and C++ units should only be seen as C files (thus not checked with C++ rules).	False
Propagate_Metrics	Propagate metric values in the RFG with respect to the hierarchy configured in the metric rule.	True
Report_All_Metric_Values	Reports all metric values, not only violations.	False
Report_Propagated_Metric_Values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values.	root
Run_IRAnalysis_Checks	If True, checks requiring iranalysis are executed, else skipped.	False
Side_Effect_Functions	Names of functions that cause an external side effect, e.g. file operations. For C++, use qualified names without template arguments, e.g. "std::vector::push_back"	('fopen', 'freopen', 'fputs', 'fclose', 'fprintf', 'printf', 'tmpfile')
Use_Bool_Cache	Whether results of computing is_effectively_bool should be cached for improved performance.	True
Use_Comment_Cache	Whether comments should be scanned and cached once to improve performance, or whether to prefer scanning repeatedly to reduce memory usage.	True
Use_ConstValue_Cache	Whether results of computing integer constant expressions should be cached for improved performance.	False
Use_Type_Cache	Whether results of computing essential/underlying types should be cached for improved performance.	True
User_Bool_Type	The name of a typedef or enum to be used as boolean type in C. Can also be a list of such names.	

Rules in Group AutosarC++17_03

AutosarC++17_03-A0.1.1

A project shall not contain instances of non-volatile variables being given values that are never subsequently used.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_dead_initializations	Whether initialisations may be reported as dead.	True
target		implementation

Possible Messages

Name	Message
conditional_unused_def	Result of assignment is not used along some path(s)
init_used_in_other_isr	Initialization is only used in some interrupt handler
unused_def	Result of assignment is not used
unused_init	Unused initialization
used_in_other_isr	Result of assignment is only used in some interrupt handler

AutosarC++17_03-A0.1.2

The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed_functions	Calls to these functions are ignored.	frozenset(['strncpy', 'memmove', 'memset', 'strcat', 'strcpy', 'memcpy', 'strncpy'])
enforcement		automated
inspect_template_instances	Whether calls in template instances should be reported.	False
level		required
target		implementation

Possible Messages

Name	Message
discarded_return_with_entity	Return value of function discarded.
discarded_return_without_entity	Return value of function discarded.

AutosarC++17_03-A0.1.3

Every defined function shall be called at least once.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_copy_assignment_operators	Whether uncalled copy assignment operators are allowed. Other rules like the "rule of three" might require copy assignment operators as part of good class design, even if there are currently no callers of the copy assignment operator.	False
ignore_copy_constructors	Whether uncalled copy constructors are allowed. Other rules like the "rule of three" might require copy constructors as part of good class design, even if there are currently no callers of the copy constructor.	False
level		required
only_check_internal_or_private_methods	Whether only function with internal linkage or private functions should be checked.	False
only_check_static_or_private_methods	Whether only static functions or private functions should be checked.	True
only_check_unit_locals	Whether only global static functions should be checked.	False
target		implementation

Possible Messages

Name	Message
defined_function_not_called	Defined function is not called

AutosarC++17_03-A0.4.2

Type long double shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
forbidden_types	The ir.Logical Types not to be used e.g. `C_Long_Double_Type`.	['C_Long_Double_Type']
level		required
target		implementation

Possible Messages

Name	Message
type_used	The type shall not be used.

AutosarC++17_03-A1.1.1

All code shall conform to ISO/IEC 14882:2014 - Programming Language C++ and shall not use deprecated features.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	The set of messages regarding syntax and constraint violations.	set([2464, 2465, 1444, 2567, 2381, 2221, 1909, 1215])
reported_severities	List of severities to display.	('error', 'warning')
show_include_path	If True, violations regarding nested includes print the #include path.	False
standards	List of allowed frontend command-line options to select language standard.	['--strict', '-A']
target		implementation
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

Possible Messages

Name	Message
access_control	Number of access control declarations is higher than translation limit (which is {})
at_quick_exit_functions	Number of functions registered with at_quick_exit() is higher than translation limit (which is {})
atexit_functions	Number of functions registered with atexit() is higher than translation limit (which is {})
base_classes	Number of direct and indirect base classes is higher than translation limit (which is {})
call_arguments	Number of arguments is higher than translation limit (which is {})
catch_handlers	Number of catch handlers is higher than translation limit (which is {})
chars_in_line	Number of characters in logical line is higher than translation limit (which is {})
chars_in_string	Number of characters in string is higher than translation limit (which is {})
cpp_members	Number of members is higher than translation limit (which is {})
data_members	Number of data members is higher than translation limit (which is {})
direct_bases	Number of direct base classes is higher than translation limit (which is {})
enumerators	Number of enumerators is higher than translation limit (which is {})
expr_in_core_constant	Number of full expressions evaluated within a core constant expression is higher than translation limit (which is {})
external_identifiers	Number of external identifiers in this unit is higher than translation limit (which is {})
final_virtual_functions	Number of final overriding virtual functions is higher than translation limit (which is {})
friends	Number of friend declarations is higher than translation limit (which is {})
function_parameters	Number of parameters is higher than translation limit (which is {})
macro_arguments	Number of macro arguments is higher than translation limit (which is {})
macro_parameters	Number of macro parameters is higher than translation limit (which is {})
macros	Number of macro definitions in this unit is higher than translation limit (which is {})
member_initializers	Number of member initializers is higher than translation limit (which is {})
nested_blocks	Number of nested blocks is higher than translation limit (which is {})
nested_composites	Nesting level of structs and unions is higher than translation limit (which is {})
nested_declarators	Number of pointer/array/function declarators is higher than translation limit (which is {})

nested_externals	Number of nested external specifications is higher than translation limit (which is {})
nested_includes	Nesting level of #includes is higher than translation limit (which is {})
nested_preprocessor_ifs	Nesting level of #if is higher than translation limit (which is {})
nested_template_instantiations	Number of nested template instantiations is higher than translation limit (which is {})
object_size	Number of bytes in object is higher than translation limit (which is {})
parenthesized_declarators	Nesting level of parenthesized declarators is higher than translation limit (which is {})
parenthesized_expressions	Nesting level of parenthesized subexpressions is higher than translation limit (which is {})
placeholders	Number of placeholders is higher than translation limit (which is {})
recursive_constexpr	Number of recursive constexpr function invocations is higher than translation limit (which is {})
scope_qualifications	Number of scope qualifications is higher than translation limit (which is {})
significant_chars_external	Number of significant initial characters for external identifier is higher than translation limit (which is {})
significant_chars_internal	Number of significant initial characters for internal identifier is higher than translation limit (which is {})
static_members	Number of static members is higher than translation limit (which is {})
std_compile	You have to compile with option {}
switch_cases	Number of cases is higher than translation limit (which is {})
template_params	Number of template parameters is higher than translation limit (which is {})
throw_spec	Number of throw specifications is higher than translation limit (which is {})
variables_in_block	Number of variables in block is higher than translation limit (which is {})
virtual_bases	Number of virtual base classes is higher than translation limit (which is {})

AutosarC++17_03-A2.2.1

Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_comments	Whether to allow characters not in the basic source character set in comments.	False
allow_in_unicode_strings	Whether to allow characters not in the basic source character set in unicode string literals.	False
allow_in_wide_strings	Whether to allow characters not in the basic source character set in wide string literals.	True
basic_characters_list	The basic character list as per rule.	\ \\ \ \\\ abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_\\ {}\\[]#\\()\\<\\>%\\:\\;.\\?*\\+\\-\\^\\&\\ \\~\\!=\\,\\\"\\`\\\\@\\
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
forbidden_characters	Forbidden characters used.

AutosarC++17_03-A2.5.1

Trigraphs shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_comments	Allow the use of trigraphs in comments	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
trigraph_use	Use of trigraph.

AutosarC++17_03-A2.6.1

Digraphs should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
digraph_use	Digraph used.

AutosarC++17_03-A2.8.1

The character \ shall not occur as a last character of a C++ comment.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
line_splicing_in_cpp_comment	Line-splicing shall not be used in // comments.

AutosarC++17_03-A2.8.2

Sections of code shall not be "commented out".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
files_to_check	Files to be checked, e.g. Primary_File or File.	File
level		required
normal_text	Tokens that indicate non-code comments. This is the first step of checking a comment. If any of these words exist in the comment text, it is regarded as not containing code.	(' and ', ' or ', ' but ', ' now ', ' to ', ' is ', ' are ', ' only ', ' be ', ' has ', ' the ', ' with ', ' because ', ' when ', ' oder ', ' und ', ' // ', '###', 'AXIVION', '++++', '----', '====')
normal_text_regex	Regular expression to match normal non-code text. This is the second step of checking a comment. If this regex matches the comment text, the comment is regarded as not containing code.	\b[\\w\\d_]+\\s+[\\w\\d_]+\\s+[\\w\\d_]+\\b
suspicious	Tokens that indicate code in comments. This is the final step of checking a comment. If the first two steps did not exclude the comment from further inspection, the comment text is searched for any of these suspicious tokens. The comment is regarded as containing code, if any suspicious token is found.	(' = ', ' == ', ' >= ', ' <= ', '[', ']', ':', '->', '->*', ' ::*', 'if', 'while', 'for', 'if (', 'while (', 'for (', '#pragma', '#else', '#endif', '#if', '#include', '++', '--')
target		implementation

Possible Messages

Name	Message
code_in_c_comment	Sections of code shall not be commented out using C-style comments.
code_in_cpp_comment	Sections of code should not be commented out using C++ comments.

AutosarC++17_03-A2.8.3

All declarations of "user-defined" types, static and non-static data members, functions and methods shall be preceded by documentation using "///" comments and "@tag" tags.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_inherited	If True, a definition does not need documentation, if a corresponding declaration is documented.	False
allow_missing_documentation_on_private	If True, a class-member definition does not need documentation, if it is `private`.	False
allow_missing_documentation_on_protected	If True, a class-member definition does not need documentation, if it is `protected`.	False
doxygen_start	Start of a valid Doxygen comment.	{'/**', '///'}
enforcement		automated
ignore_defaulted	If True, defaulted function declarations are not checked for comments.	False
ignore_deleted	If True, deleted function declarations are not checked for comments.	False
ignore_redefinitions	If True, method redefinitions are not checked as they can 'inherit' the comment from the redefined method.	False
ignore_tool_comments	An optional compiled regular expression. Comments where this regex finds a matching substring are ignored in the search for a doxygen comment (e.g. control-comments of other tools).	None
level		required
node_types	IR node types to check for preceding Doxygen comment.	{'Routine_Definition', 'Routine_Declaration', 'Named_Type_Definition', 'Field_Definition'}
target		implementation

Possible Messages

Name	Message
missing_doxygen_comment_before_def	No Doxygen comment before declaration.

AutosarC++17_03-A2.8.4

C-style comments shall not be used.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
exception	Start of a valid comment, even if it has an invalid prefix.	None
invalid	Start of an invalid comment.	/*
level		required
target		implementation

Possible Messages

Name	Message
cpp_comment_style	Use of invalid comment style in C++ unit.

AutosarC++17_03-A2.9.1

A header file name shall be identical to a type name declared in it if it declares a type.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
file_type	Type of header file to examine.	User_Include_File
junk_characters	Characters that should not be considered in the filename to type similarity.	_-::
level		required
named_node_types	Physical IR node types to inspect for logical names to compare against the file name	Named_Type_Interface
similarity_threshold	Ratio of how similar the names need to be.	0.9
target		implementation

Possible Messages

Name	Message
source_file_name_not_type	The header should be named as a type it declares.

AutosarC++17_03-A2.11.1

Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_classes	Whether identifiers in local classes can hide surrounding ones	True
allow_in_lambdas	Whether identifiers in lambdas can hide surrounding ones	True
allow_in_local_classes	Whether identifiers in local classes can hide surrounding ones	False
allow_in_namespaces	Whether identifiers in namespaces can hide surrounding ones	True
allow_parameter_in_function_declaration	Whether identifiers of parameters in function declarations may hide surrounding ones, given the function declaration uses a different identifier for the same parameter that does not hide a surrounding one.	False
check_inheritance_hiding	Whether to check for hiding by inheritance.	False
check_nested_classes	Whether symbols from inner classes should be reported as hiding symbols of outer classes.	False
distinguish_name_spaces	Whether C name spaces should be distinguished.	False
enforcement		automated
hiding_entities	LIR node types to consider symbols that hide other symbols.	{'Variable', 'Typedef_Type', 'Field', 'Parameter'}
level		required
maxlen	Number of significant characters {or None}	None
target		implementation
tolerate_constructor_parameters_hiding_fields	Allow constructor parameters to hide fields	False
tolerate_function_prototype_argument_hiding	Whether to tolerate arguments of a function prototype, that itself is a function argument, that have the same identifier as a argument of the outer function. If 'True' is given, the rule will still report cases, where argument identifiers match up and the function prototype refers to arguments of the outer function (e.g. by `decltype()`).	False
tolerate_macro_local_identifier	Whether an identifier being local to a macro is allowed to hide identifiers from outside the macro	False
unchecked_types	Deprecated legacy option, kept to avoid breaking old configurations.	{'Routine_Call', 'Stack_Object_Definition'}

Possible Messages

Name	Message
hiding	{ } hides { }

AutosarC++17_03-A2.11.2

A "using" name shall be a unique identifier within a namespace.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
reused_using	Using name reused.

AutosarC++17_03-A2.11.3

A "user-defined" type name shall be a unique identifier within a namespace.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
check_types_only_with_external_linkage	Whether the types should only be checked when they have external linkage	False
enforcement		automated
level		required
target		implementation
tolerate_macros	Whether #define and #undef using the typedef's name is allowed	False
tolerate_typedef	Whether a typedef to the tag may have the same name.	False
tolerate_typedef_entity	Whether the entity forming the typedef's underlying type can have the same name.	False

Possible Messages

Name	Message
composite_defined_multiple_times	Composite type has multiple definitions.
enum_defined_multiple_times	Enumeration type has multiple definitions.
reused_tag	Tag name reused.
reused_typedef	Typedef name reused.

AutosarC++17_03-A2.11.4

The identifier name of a non-member object or function with static storage duration should not be reused.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_static_no_linkage	Whether to allow same identifiers for static objects without linkage (e.g. function local static objects)	False
enforcement		automated
func_filter	Restricts which functions are considered.	None
level		required
target		implementation
use_unqualified_names	If set to True, ns1::Type is considered equal to ns2::Type.	False
var_filter	Restricts which variables are considered.	None

Possible Messages

Name	Message
reused_global_function_name	Name of object or function with static storage duration reused.
reused_global_variable_name	Name of object or function with static storage duration reused.

AutosarC++17_03-A2.11.5

The identifier name of a non-member object or function with static storage duration should not be reused.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_static_no_linkage	Whether to allow same identifiers for static objects without linkage (e.g. function local static objects)	False
enforcement		automated
func_filter	Restricts which functions are considered.	None
level		required
target		implementation
use_unqualified_names	If set to True, ns1::Type is considered equal to ns2::Type.	True
var_filter	Restricts which variables are considered.	None

Possible Messages

Name	Message
reused_global_function_name	Name of object or function with static storage duration reused.
reused_global_variable_name	Name of object or function with static storage duration reused.

AutosarC++17_03-A2.14.1

Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed	List of allowed characters after backslash.	"\"?\\abfnrtv
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
nonstandard_escape_sequence	Use of non-standard escape sequence.

AutosarC++17_03-A2.14.2

Narrow and wide string literals shall not be concatenated.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
cpp11_mode	Use rules as defined in the C++11 standard.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
mixed_string_concatenation	Concatenation of mixed string encodings
narrow_wide_concat	Concatenation of narrow and wide string literal

AutosarC++17_03-A2.14.3

Type wchar_t shall not be used.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
forbidden_types	The ir.Logical Types not to be used e.g. `C_Long_Double_Type`.	['CPP_WChar_Type']
level		required
target		implementation

Possible Messages

Name	Message
type_used	The type shall not be used.

AutosarC++17_03-A3.1.1

It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
accept_const_fields	Whether const-qualified static fields in header files should be tolerated.	True
accept_const_variables	Whether global const variables in header files should be tolerated.	True
enforcement		automated
find_non_external_function_declaration	Whether function declarations with non-external linkage should be reported.	True
level		required
target		implementation

Possible Messages

Name	Message
function_definition_in_header	Definition in header file.
static_field_def_in_header	Definition in header file.
variable_definition_in_header	Definition in header file.

AutosarC++17_03-A3.1.2

Header files, that are defined locally in the project, shall have a file name' extension of one of: ".h", ".hpp" or "..hxx".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed_extensions	Which file extensions are allowed.	set[{'h', '.hxx', 'hpp'}]
enforcement		automated
file_type	The files to check the extensions of.	User_Include_File
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
illegal_file_extension	Use only certain extensions for file names.

AutosarC++17_03-A3.1.3

Implementation files, that are defined locally in the project, should have a file name extension of ".cpp".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed_extensions	Which file extensions are allowed.	set(['.cpp'])
enforcement		automated
file_type	The files to check the extensions of.	Primary_File
level		advisory
target		architecture/design/implementation

Possible Messages

Name	Message
illegal_file_extension	Use only certain extensions for file names.

AutosarC++17_03-A3.1.4

When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_extern	Whether to consider only extern declared arrays	True
report_definitions	Whether definitions of array variables should also be reported	True
target		implementation

Possible Messages

Name	Message
unbounded_array	Array declaration with unknown size.

AutosarC++17_03-A3.3.1

Objects or functions with external linkage shall be declared in a header file.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_function_declaration_in_header	Object or function with external linkage shall be declared in a header file
missing_variable_declaration_in_header	Object or function with external linkage shall be declared in a header file

AutosarC++17_03-A3.3.2

Non-POD type objects with static storage duration shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
static_nonpod_variable	Do not use non-POD type objects with static storage duration.

AutosarC++17_03-A3.9.1

Typedefs that indicate size and signedness should be used in place of the basic numerical types.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_floating_point	Whether floating point types without should not be reported.	True
enforcement		automated
ignore_inherited	If true, missing typedefs in inherited methods are not reported.	False
level		required
target		implementation
treat_plain_char_as_numerical	Whether the char type without any signedness should be reported.	True

Possible Messages

Name	Message
missing_integer_typedef	Use of base type outside typedef.
wrong_integer_typedef	Use of badly named typedef for base type.

AutosarC++17_03-A4.5.1

Expressions with type enum shall not be used as operands to built-in operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_use_in_operator_calls	Whether enum arguments in calls to forbidden overloaded operators should be reported.	True
target		implementation

Possible Messages

Name	Message
enum_operand_outside_comparison	Use of enum operand in arithmetic or similar context

AutosarC++17_03-A4.7.1

An integer expression shall not lead to data loss.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
abstract_interpretation_overflow	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cast_overflow	Cast on result of arithmetic computation may cause overflow
cast_truncate	Cast may truncate value
cast_underflow	Cast on result of arithmetic computation may cause underflow
overflow	Arithmetic computation may cause overflow
underflow	Arithmetic computation may cause underflow

AutosarC++17_03-A4.10.1

Only nullptr literal shall be used as the null-pointer-constant.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
require_nullptr	Require C++11 nullptr instead of macro NULL or literal zero {0}.	True
target		architecture/design/implementation

Possible Messages

Name	Message
null_constant	Only nullptr literal shall be used as the null-pointer-constant.
zero_as_null	Use of literal zero {0} as null-pointer-constant, use {} instead

AutosarC++17_03-A5.0.1

The value of an expression shall be the same under any order of evaluation that the standard permits.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_exceptions	Ignore possible exceptions in the computation of side effects. (this option only has an effect if report_calls is enabled)	False
level		required
report_calls	If True, unsequenced function calls are reported.	True
target		implementation

Possible Messages

Name	Message
unsequenced_call_volatile	Function calls unsequenced with volatile access
unsequenced_calls	Unsequenced function calls
unsequenced_read_write	Unsequenced read and write accesses
unsequenced_volatiles	Unsequenced volatile accesses
unsequenced_writes	Unsequenced writes

AutosarC++17_03-A5.0.2

The condition of an if-statement and the condition of an iteration-statement shall have type bool.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_in_conditions	If True, only logical operators inside conditions are checked.	True
report_do_while_false	Whether do ... while[0] should be reported.	True
report_while_true	Whether while[1] ... should be reported.	True
target		implementation

Possible Messages

Name	Message
nonbool_if_condition	Condition must have type bool
nonbool_logical_operator_operand	Sub-condition must have type bool
nonbool_loop_condition	Condition must have type bool

AutosarC++17_03-A5.0.3

The declaration of objects should contain no more than two levels of pointer indirection.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
max_levels	Maximum number of allowed pointer-indirection levels.	2
target		implementation

Possible Messages

Name	Message
too_many_pointer_levels	More than {} level{} of pointer indirection

AutosarC++17_03-A5.1.1

Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_nonconst_variable_initialization	If set to `True`, allow the initialization of a non-const variable with a string literal in addition to the initialization of a const variable. A reassignment of such a variable is still a violation.	False
allowed	Literal values that are ok.	{0.0, 1.0, 2.0}
allowed_contexts	Optional set of PIR classes or functions {[node] -> bool} for allowed contexts, e.g. Case_Label.	set[<function allow_in_type_initialization at 0x7dfa84459b0>]
allowed_logging_contexts	List of fully qualified function types that are considered logging contexts [using operator<>]. If this is non-empty, `std::throw()` is considered a valid logging context as well.	['std::basic_ostream']
allowed_string_contexts	Optional set of PIR classes or functions {[node] -> bool} for allowed contexts.	set[<function allow_in_type_initialization at 0x7dfa84459b0>]
allowed_strings	Literal values that are ok.	['', ' ']
enforcement		partially automated
exceptions	Optional predicate to filter out cases that should be allowed as an exception. The predicate takes a literal node and returns True if the given node is such an exception.	None
exclude_pp_literals	If True, literals in conditions of #if are ignored.	True
exclude_single_uses	If True, report only literals used more than once.	False
level		required
target		implementation

Possible Messages

Name	Message
magic_number	Use of magic literal.
magic_number_without_token	Use of magic literal.
magic_string	Use of magic string literal.
possible_magic_number	Potential use of magic literal.

AutosarC++17_03-A5.1.2

Variables shall not be implicitly captured in a lambda expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
implicit_lambda_capture	Variables shall not be implicitly captured in a lambda expression.

AutosarC++17_03-A5.1.5

If a lambda expression is used in the same scope in which it has been defined, the lambda should capture objects by reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
level		advisory
target		implementation

Possible Messages

Name	Message
local_lambda	An only locally used lambda should capture by reference.

AutosarC++17_03-A5.1.7

The underlying type of lambda expression shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
underlying_lambda_type	The underlying type of lambda expression shall not be used.

AutosarC++17_03-A5.1.8

Lambda expressions should not be defined inside another lambda expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
nested_lambda	Lambda expressions should not be defined inside another lambda expression

AutosarC++17_03-A5.2.1

dynamic_cast should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
dynamic_cast	dynamic_cast should not be used.

AutosarC++17_03-A5.2.2

Traditional C-style casts shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_void_cast	If True, [void] is always allowed, else only for return value of calls.	False
allow_void_cast_on_call	If True, [void] is allowed, for return value of calls.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
c_cast	Use of C-style cast in C++ unit.

AutosarC++17_03-A5.2.3

A cast shall not remove any const or volatile qualification from the type of a pointer or reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
check_const	Whether casts affecting const-qualifiers should be reported	True
check_volatile	Whether casts affecting volatile-qualifiers should be reported	True
enforcement		automated
level		required
only_top_level	Whether the check only applies to the top-level pointee or recursively to deeper pointer levels as well	False
target		implementation

Possible Messages

Name	Message
cast_removes_const	Cast removes const qualification
cast_removes_volatile	Cast removes volatile qualification
implicit_cast_removes_const	Implicit cast removes const qualification
implicit_cast_removes_volatile	Implicit cast removes volatile qualification

AutosarC++17_03-A5.2.4

reinterpret_cast shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
reinterpret_cast	reinterpret_cast should not be used.

AutosarC++17_03-A5.3.1

Evaluation of the operand to the typeid operator shall not contain side effects.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
level		required
target		implementation

Possible Messages

Name	Message
side_effect_in_typeid	Operand of "typeid" shall not contain side effects

AutosarC++17_03-A5.5.1

The right hand operand of the integer division or remainder operators shall not be equal to zero.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
abstract_interpretation_div_by_zero	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
division_by_zero	Division by zero
modulo_by_zero	Modulo by zero
possible_division_by_zero	Possible division by zero
possible_modulo_by_zero	Possible modulo by zero

AutosarC++17_03-A5.10.1

A pointer to member virtual function shall only be tested for equality with null-pointer-constant.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
member_virtual_function_comparison	Comparison of a member virtual function with non-nullptr.

AutosarC++17_03-A5.16.1

The ternary conditional operator shall not be used as a sub-expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_nested_conditional_operator	Use of nested conditional operator.

AutosarC++17_03-A6.4.1

Every switch statement shall have at least one case-clause.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
minimum_switch_cases	The number of cases a switch statement should at least have.	2
target		implementation

Possible Messages

Name	Message
empty_switch	Switch has no "case" clause.
minimum_switch_cases	Switch has too little "case" clauses.

AutosarC++17_03-A6.5.1

A for-loop that loops through all elements of the container and does not use its loop-counter shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
exclude_simple_for_loops	If set to `True` this rule will not report unused loop counters in the simple cases where the C for-loop only references loop-counters in its condition but no other variables.	True
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
could_be_range_based	For loop could be a range-based for loop.
unused_loop_counter	For loop does not use its loop counter.

AutosarC++17_03-A6.5.2

A for loop shall contain a single loop-counter which shall not have floating type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
loop_float_counter	Loop-counter of for loop shall not have floating type
loop_missing_counter	For loop has no loop-counter
loop_multiple_counters	For loop shall have only a single loop-counter

AutosarC++17_03-A6.6.1

The goto statement shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
goto	Do not use goto.

AutosarC++17_03-A7.1.1

Constexpr or const specifiers shall be used for immutable data declaration.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_pointer_variables	Whether variables of pointer type should be ignored.	False
level		required
only_check_unit_locals	Whether only local variables and global static variables should be checked.	False
only_immutable_data	Whether only declarations with immutable data should be checked.	True
target		implementation

Possible Messages

Name	Message
variable_missing_const	An immutable variable shall be const/constexpr qualified.

AutosarC++17_03-A7.1.3

CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_nonpointer_typedefs	Whether to allow the qualifier on the rhs of a type, if it is a non-pointer typedef.	False
allowed_typedefs	Set of names for typedefs/usings that are allowed (e.g. int32_t).	set(['int32_t', 'uint_least64_t', 'intptr_t', 'uintmax_t', 'int_fast16_t', 'intmax_t', 'int_fast8_t', 'int64_t', 'size_t', 'int_fast64_t', 'time_t', 'uint8_t', 'lldiv_t', 'int_least8_t', 'div_t', 'uint_least16_t', 'clock_t', 'uint_least32_t', 'int_least64_t', 'int_least16_t', 'int_least32_t', 'uint_least8_t', 'uintptr_t', 'max_align_t', 'int8_t', 'fpos_t', 'ldiv_t', 'uint_fast32_t', 'uint_fast64_t', 'nullptr_t', 'int_fast32_t', 'uint_fast16_t', 'uint32_t', 'ptrdiff_t', 'int16_t', 'uint64_t', 'uint16_t', 'uint_fast8_t'])
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
lhs_cv_qualifier	CV qualifier on the lhs of a type.

AutosarC++17_03-A7.1.4

The register keyword shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
register_keyword	The register keyword shall not be used

AutosarC++17_03-A7.1.5

The auto specifier shall not be used apart from following cases: (1) to declare that a variable has the same type as return type of a function call, (2) to declare that a variable has the same type as initializer of non-fundamental type, (3) to declare parameters of a generic lambda expression, (4) to declare a function template using trailing return type syntax.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_for_loop_counter	Disables message if auto is used to declare a for-loop counter.	False
allow_generic_lambda_parameters	Allows auto as parameter type in a generic lambda	True
allow_nonfundamental_initializer	Allows auto to declare variables having a function call or initializer of non-fundamental type	True
allow_template_instance	Disables message if auto stands for a template instance.	False
allowed_contexts	Set of context predicates in which auto is allowed.	set[]
allowed_types	Set of types for which auto is allowed, given as name pattern, function, or LIR class name.	set[]
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cpp11_auto	Use of C++11 auto type specifier.

AutosarC++17_03-A7.1.6

The typedef specifier shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
typedef_specifier	The typedef specifier shall not be used.

AutosarC++17_03-A7.1.7

Each expression statement and identifier declaration shall be placed on a separate line.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_for_loop_init	Whether a multi-declaration is allowed in the for-init statement	True
allow_uninitialized_simple_type	Whether a multi-declaration of simple type is allowed when there is no initialization	False
enforcement		automated
ignore_stmts	Statements to be ignored when counting statements.	['Statement_Sequence', 'C_For_Loop']
level		required
target		implementation

Possible Messages

Name	Message
multi_declaration	Multiple declarators in single declaration
multiple_statements_per_line	Multiple statements per line.

AutosarC++17_03-A7.2.1

An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
asgn_not_an_enumerator	Assignment of value which does not correspond to an enumerator
asgn_out_of_enum_range	Assignment of value outside the range of the enumerators
conversion_creating_bad_enum_value	Expression does not correspond to an enumerator in {}

AutosarC++17_03-A7.2.2

Enumeration underlying base type shall be explicitly defined.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unbased_enum	Enumeration underlying base type shall be explicitly defined.

AutosarC++17_03-A7.2.3

Enumerations shall be declared as scoped enum classes.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unscoped_enum	Enumerations shall be declared as scoped enum classes.

AutosarC++17_03-A7.2.4

In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
incomplete_enum_init	Do not initialize enumerators other than the first, or initialize all

AutosarC++17_03-A7.4.1

The asm declaration shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_assembler	Use of assembler.

AutosarC++17_03-A7.5.1

A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_const_reference	Whether only to report returns of parameters with reference to const.	True
target		implementation

Possible Messages

Name	Message
returning_reference_to_refparam	Returning reference/pointer to reference parameter.

AutosarC++17_03-A7.5.2

Functions shall not call themselves, either directly or indirectly.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_cycle	Cycle of recursive functions
recursive_function	Function is directly recursive

AutosarC++17_03-A8.2.1

When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_trailing_return	Use trailing return-type syntax for function templates.

AutosarC++17_03-A8.4.1

Functions shall not be defined using the ellipsis notation.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_declarations	Allow declarations of functions with variable number of arguments (e.g. for existing library functions); only report definitions.	True
ignore_inherited	Do not report functions inheriting a variable number of arguments.	False
level		required
target		implementation

Possible Messages

Name	Message
ellipsis_parameter	Function definitions shall not use ellipsis

AutosarC++17_03-A8.4.2

All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
lambda_return_missing_value	Return without value in non-void lambda expression
missing_return	Non-void function needs return with value at end.
missing_return_in_lambda	Non-void lambda expression needs return with value at end.
return_missing_value	Return without value in non-void function

AutosarC++17_03-A8.5.1

In an initialization list, the order of initialization shall be following: (1) virtual base classes in depth and left to right order of the inheritance graph, (2) direct base classes in left to right order of inheritance list, (3) non-static data members in the order they were declared in the class definition.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	If provided, only messages of these types are reported.	[1719]
reported_severities	List of severities to display.	('error', 'warning', 'remark')
target		implementation
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

AutosarC++17_03-A8.5.2

Braced-initialization {}, without equals sign, shall be used for variable initialization.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
old_style_variable_init	Braced-initialization {}, without equals sign, shall be used for variable initialization.

AutosarC++17_03-A8.5.3

A variable of type auto shall not be initialized using {} or ={} braced-initialization.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
braced_auto_variable_init	A variable of type auto shall not be initialized using {} or ={} braced-initialization.

AutosarC++17_03-A8.5.4

A constructor taking parameter of type std::initializer_list shall only be defined in classes that internally store a collection of objects.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
level		advisory
target		implementation

Possible Messages

Name	Message
constructor_init_list_container	The class should be a container if a constructor taking std::initializer_list is defined.

AutosarC++17_03-A9.6.1

Bit-fields shall be either unsigned integral, or enumeration [with underlying type of unsigned integral type].

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
invalid_bitfield	Bit-fields shall be either unsigned integral, or enumeration (with underlying type of unsigned integral type).

AutosarC++17_03-A10.1.1

Class shall not be derived from more than one base class which is not an interface class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_pure_interfaces	Whether C++ interfaces are allowed, i.e. classes with only pure virtual members.	True
level		required
target		implementation

Possible Messages

Name	Message
multiple_inheritance	Use of multiple inheritance.

AutosarC++17_03-A10.2.1

Non-virtual member functions shall not be redefined in derived classes.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
redefinition_of_nonvirtual_function	Redefinition/hiding of non-virtual function.

AutosarC++17_03-A10.3.1

Virtual function declaration shall contain exactly one of the three specifiers: {1} virtual, {2} override, {3} final.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
disable_for_destructors	If set, destructors are not checked.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
invalid_combination	Use only one of {1} virtual, {2} override, {3} final.

AutosarC++17_03-A10.3.2

Each overriding virtual function shall be declared with the override or final specifier.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_final	If set to True, don't report overriding virtual functions declared with final.	True
enforcement		automated
ignore_destructors	If set to False, also report destructors. Note that not all compilers support this.	False
level		required
target		implementation

Possible Messages

Name	Message
missing_override	Override of functions is only permitted with keyword override/final.

AutosarC++17_03-A10.3.3

Virtual functions shall not be introduced in a final class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
virtual_method	Virtual functions shall not be introduced in a final class.
virtual_method_override	Virtual functions shall not be overridden without final in a final class.

AutosarC++17_03-A10.3.5

A user-defined assignment operator shall not be virtual.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
assignment_virtual	A user-defined assignment operator shall not be virtual.

AutosarC++17_03-A11.0.1

A non-POD type should be defined as class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
require_class_for_pod	Whether POD types should be classes as well	False
target		implementation

Possible Messages

Name	Message
cpp_struct	Use of struct in C++ unit.

AutosarC++17_03-A11.0.2

A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
no_function_member	Struct shall not provide any member functions or methods.
no_struct_as_base	A struct shall not be a base of another struct or class.
no_struct_inheritance	A struct shall not inherit from another struct or class.
non_public_member	Structs shall only have public data members.

AutosarC++17_03-A11.3.1

Friend declarations shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
friend_class	Do not use friend class/struct/union declarations.
friend_decl	Do not use friend declarations.

AutosarC++17_03-A12.0.1

If a class declares a copy or move operation, or a destructor, either via "=default", "=delete", or via a user-provided declaration, then all others of these five special member functions shall be declared as well.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_defaulted_destructor_only	Allow classes with a defaulted destructor and no other special member functions.	False
allow_destructor_only	Allow all destructors without copy or move constructors	False
allow_empty_destructor	Allow empty destructors without copy or move constructors.	False
allow_missing_destructor	Suppress messages about missing destructors.	False
enforcement		automated
ignore_pod_classes	Whether POD classes should be checked at all	False
level		required
target		implementation

Possible Messages

Name	Message
missing_constructor_and_asgn	Class with destructor should also declare a copy or move constructor and assignment operator.
missing_copy_asgn	Class with copy constructor is missing copy assignment operator.
missing_copy_constructor	Class with copy assignment operator is missing copy constructor.
missing_destructor	Class with copy or move constructors or assignment operators should also declare a destructor.
missing_move_asgn	Class with move constructor is missing move assignment operator.
missing_move_constructor	Class with move assignment operator is missing move constructor.

AutosarC++17_03-A12.1.1

Constructors shall explicitly initialize all virtual base classes, all direct non-virtual base classes and all non-static data members.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
init_functions	Names of functions to be inspected as well when called directly from constructor.	{'Init', 'init'}
inspect_directly_called_methods	Inspect all methods directly called from constructor.	False
level		required
only_member_initializer_list	Only inspect member initializer list and not the constructor body/methods.	False
report_missing_field_constructors	Enables detection of constructors which rely on implicit field constructor calls.	True
target		implementation

Possible Messages

Name	Message
implicit_field_init	Field is only implicitly initialized in constructor.
missing_base_class_init	Base class is not explicitly initialized in constructor.
missing_field_init	Field is not initialized in constructor.

AutosarC++17_03-A12.1.2

Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_member_initializer_list	Only inspect member initializer list and not the constructor body/methods.	True
target		implementation

Possible Messages

Name	Message
nsdmi_mixed	Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type.

AutosarC++17_03-A12.1.3

If all user-defined constructors of a class initialize data members with constant values that are the same across all constructors, then data members shall be initialized using NSDMI instead.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_constructors_of_types	List of fully qualified type names which constructors are considered literals, without regard to the actual arguments.	[]
allow_literal_like_constructors	Whether to consider constructors that take only literals as arguments a literal. If set to true, this is checked recursively.	False
enforcement		automated
level		required
min_number_common_inits	Minimum number (inclusive) of common initializations to enforce use of NSDMI.	1
target		implementation

Possible Messages

Name	Message
use_nsdmi	Use NSDMI for common constant initializations ({}).

AutosarC++17_03-A12.1.4

All constructors that are callable with a single argument of fundamental type shall be declared explicit.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
consider_only_fundamental_types	Whether this check should be limited to single arguments of fundamental type or should also be applied to user defined types.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
constructor_missing_explicit	Constructor shall be declared explicit

AutosarC++17_03-A12.4.1

Destructor of a base class shall be public virtual, public override or protected non-virtual.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
base_class_destructor	Destructor of a base class shall be public virtual, public override or protected non-virtual.

AutosarC++17_03-A12.4.2

If a public destructor of a class is non-virtual, then the class should be declared final.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
non_final	If a public destructor of a class is non-virtual, then the class should be declared final.

AutosarC++17_03-A12.6.1

All class data members that are initialized by the constructor shall be initialized using member initializers.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
init_functions	Names of functions to be inspected as well when called directly from constructor.	{'Init', 'init'}
inspect_directly_called_methods	Inspect all methods directly called from constructor.	False
level		required
only_member_initializer_list	Only inspect member initializer list and not the constructor body/methods.	True
report_missing_field_constructors	Enables detection of constructors which rely on implicit field constructor calls.	True
target		implementation

Possible Messages

Name	Message
implicit_field_init	Field is only implicitly initialized in constructor.
missing_field_init	Field is not initialized in constructor.

AutosarC++17_03-A12.8.1

A copy constructor shall only initialize its base classes and the non-static members of the class of which it is a member.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_copy_constructor	Whether to report side effects on copy constructors.	True
report_move_constructor	Whether to report side effects on move constructors.	True
target		implementation

Possible Messages

Name	Message
copy_ctor_with_side_effect	Copy Constructor has side-effect
move_ctor_with_side_effect	Move Constructor has side-effect

AutosarC++17_03-A12.8.2

User-defined copy and move assignment operators should use user-defined no-throw swap function.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
asgn_no_swap	A move/copy assignment operator shall use a no-throw swap function.
asgn_not_nothrow	Used swap function is not no-throw.

AutosarC++17_03-A12.8.3

Moved-from object shall not be read-accessed.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		partially automated
inspect_class_field_moves	Detecting moved-from accesses on class fields requires heavy computation. This can be disabled by setting this option to False, but would result in false-positives in the context of class fields.	True
level		required
read_move_function_exceptions	Names of functions that are considered to leave the moved from objects in a well-specified state and are therefore except.	['std::unique_ptr::unique_ptr', 'std::unique_ptr::operator=', 'std::shared_ptr::shared_ptr', 'std::shared_ptr::operator=', 'std::weak_ptr::weak_ptr', 'std::weak_ptr::operator=', 'std::basic_filebuf::basic_filebuf', 'std::basic_filebuf::operator=', 'std::thread::thread', 'std::thread::operator=', 'std::unique_lock::unique_lock', 'std::unique_lock::operator=', 'std::shared_lock::shared_lock', 'std::shared_lock::operator=', 'std::promise::promise', 'std::promise::operator=', 'std::future::future', 'std::future::operator=', 'std::shared_future::shared_future', 'std::shared_future::operator=', 'std::packaged_task::packaged_task', 'std::packaged_task::operator=']
read_move_type_exceptions	Names of types that are considered to be left well-specified state after a move and are therefore except.	['std::basic_ios']
target		implementation

Possible Messages

Name	Message
moved_from_read	Don't read-access a moved-from object

AutosarC++17_03-A12.8.4

Move constructor shall not initialize its class members and base classes using copy semantics.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
move_constructor	Move constructor shall not initialize using copy semantics.

AutosarC++17_03-A12.8.6

Copy and move constructors and copy assignment and move assignment operators shall be declared protected or defined "=delete" in base class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
special_asgn	Copy and move assignment operators shall be declared protected or defined "=delete" in base class.
special_ctor	Copy and move constructors shall be declared protected or defined "=delete" in base class.

AutosarC++17_03-A12.8.7

Assignment operators should be declared with the ref-qualifier &.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
asgn_missing	Assignment operators should be declared with the ref-qualifier &.

AutosarC++17_03-A13.1.1

User-defined literals shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
user_defined_literal	User-defined literals shall not be used.

AutosarC++17_03-A13.1.2

User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
user_defined_literal_naming	User-defined literals shall have a suffix matching "[a-zA-Z]+".

AutosarC++17_03-A13.1.3

User defined literals operators shall only perform conversion of passed parameters.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
literal_side_effects	User-defined literals shall not have side effects.

AutosarC++17_03-A13.2.1

An assignment operator shall return a reference to "this".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
asgn_ref_this	An assignment operator shall return a reference to "this".

AutosarC++17_03-A13.2.2

A binary arithmetic operator and a bitwise operator shall return a "prvalue".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_bitwise_shift	Whether to allow non-basic values for operator>> or operator<<.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
arith_bitwise_basic_value	Binary arithmetic or bitwise operator shall return a basic value.

AutosarC++17_03-A13.2.3

A relational operator shall return a boolean value.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
relational_bool	A relational operator shall return a boolean value.

AutosarC++17_03-A13.3.1

A function that contains "forwarding reference" as its argument shall not be overloaded.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
overloaded_fref	Functions that contain forwarding reference parameters shall not be overloaded

AutosarC++17_03-A13.5.1

If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
array_access_non_const	No const version of operator[] implemented.

AutosarC++17_03-A13.6.1

Digit sequences separators ' shall only be used as follows: (1) for decimal, every 3 digits, (2) for hexadecimal, every 2 digits, (3) for binary, every 4 digits.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
digit_separator	Possibly unreadable digit separators.

AutosarC++17_03-A15.1.1

Only instances of types derived from std::exception shall be thrown.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throwing_std_exception	Only instances of types derived from std::exception shall be thrown.

AutosarC++17_03-A15.1.2

An exception object should not have pointer type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throwing_pointer	Exception object of pointer type

AutosarC++17_03-A15.1.3

All thrown exceptions should be unique.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
throwing_duplicate	All thrown exceptions should be unique.

AutosarC++17_03-A15.2.1

Constructors that are not noexcept shall not be invoked before program startup.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
static_field_only_noexcept	Constructor called that may throw an exception in static field.
static_variable_only_noexcept	Constructor called that may throw an exception in static variable.

AutosarC++17_03-A15.3.1

Unchecked exceptions should be handled only in main or thread's main functions.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
autosar_exception_model	The default model for what counts as an checked / unchecked exception.	17_03
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
unchecked_exception	Handle unchecked exceptions only in main or thread's main function.

AutosarC++17_03-A15.3.3

There should be at least one exception handler to catch all otherwise unhandled exceptions.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
external_base_exceptions	Sequence of external/third-party exception base-classes that should be caught	[]
inspect_thread_main	Whether to also inspect thread main functions.	True
level		required
std_base_exceptions	Sequence of cpp-std exception base-classes that should be caught	[]
target		implementation

Possible Messages

Name	Message
missing_catch_all	Catch-all required around main program body
missing_catch_handler	Handler for {} needed

AutosarC++17_03-A15.3.4

Catch-all (ellipsis and std::exception) handlers shall be used only in (a) main, (b) task main functions, (c) in functions that are supposed to isolate independent components and (d) when calling third-party code that uses exceptions not according to AUTOSAR C++14 guidelines.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_extern_c_functions	Whether to allow a catch-all in `extern C` functions.	False
allow_in_functions_matching	If not None, a `re.compile()`'ed object where functions whose qualified name matches the regex are allowed to contain catch-alls.	None
allow_in_main	Whether to allow a catch-all in the main() function.	True
allow_in_thread_main	Whether to allow a catch-all in thread-main functions.	True
allow_rethrow	Whether to allow a catch-all with a re-throw.	False
autosar_exception_model	The default model for what counts as an checked / unchecked exception.	<bauhaus.ir.autosar.exceptions.autosar_exceptions.AutosarExceptionModel object at 0x7dfa83c4650>
disallow_std_exception	Whether to consider catching the literal std::exception as a catch-all.	True
enforcement		non-automated
level		required
target		implementation

Possible Messages

Name	Message
catch_all	Use of catch(...).
catch_std_exception	Catching std::exceptions is too general.

AutosarC++17_03-A15.3.5

A class type exception shall always be caught by reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_class_types	Whether all types should be caught by reference or only class types.	True
target		implementation

Possible Messages

Name	Message
catch_without_reference	A class type exception shall always be caught by reference.

AutosarC++17_03-A15.4.1

Dynamic exception-specification shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

AutosarC++17_03-A15.4.2

If a function is declared to be noexcept, noexcept(true) or noexcept(<true condition>), then it shall not exit with an exception.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
generateViolationPath	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignoreConstructorDestructor	Whether to ignore escaping exceptions from constructors and destructors.	False
ignoreThrowingFunctions	Whether to ignore noexcept specification violations on function that actually throw an exception.	False
ignoreUnknownRoutines	Whether to ignore extern or only declared routines.	False
level		required
reportNoexceptFalseViolations	Whether to report cases where the function is declared noexcept(false), but no exceptions are ever thrown (directly or indirectly).	False
target		implementation

Possible Messages

Name	Message
implicit_noexcept_spec_violation_without	Function implicitly declared noexcept(false) but no exceptions will be thrown.
noexcept_spec_violation_with	Exception violates function's noexcept-specification.
noexcept_spec_violation_without	Function declared noexcept(false) but no exceptions will be thrown.

AutosarC++17_03-A15.4.3

Function's noexcept specification shall be either identical or more restrictive across all translation units and all overriders.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
noexcept_mismatch_redecl	Function's noexcept specification shall be identical
noexcept_mismatch_override	exception specification for virtual function "{}" is incompatible with that of overridden function "{}"

AutosarC++17_03-A15.4.4

A declaration of non-throwing function shall contain noexcept specification.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
generateViolationPath	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignoreConstructorDestructor	Whether to ignore escaping exceptions from constructors and destructors.	False
ignoreThrowingFunctions	Whether to ignore noexcept specification violations on function that actually throw an exception.	True
ignoreUnknownRoutines	Whether to ignore extern or only declared routines.	True
level		required
reportNoexceptFalseViolations	Whether to report cases where the function is declared noexcept(false), but no exceptions are ever thrown (directly or indirectly).	True
target		implementation

Possible Messages

Name	Message
implicit_noexcept_spec_violation_without	Function implicitly declared noexcept(false) but no exceptions will be thrown.
noexcept_spec_missing	Explicit noexcept-specification missing.
noexcept_spec_violation_with	Exception violates function's noexcept-specification.
noexcept_spec_violation_without	Function declared noexcept(false) but no exceptions will be thrown.

AutosarC++17_03-A15.4.5

Checked exceptions that could be thrown from a function shall be specified together with the function declaration and they shall be identical in all function declarations and for all its overriders.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
autosar_exception_model	The default model for what counts as an checked / unchecked exception.	17_03
enforcement		automated
ignore_definitions_if_declaration_documented	Instead of requiring the documentation to be repeated for every declaration (including definition), with this option set, the rule only checks the non-defining declarations if at least one non-defining declaration exists.	False
level		required
match_qualified_name	Matches the fully-qualified name when comparing documented exceptions with what can actually occur. If set to 'False', this rule will accept any suffix of the qualified name of an exception class as the documentation string.	True
target		implementation
throw_marker	The command to document an exception to be thrown.	@throw

Possible Messages

Name	Message
differing_documented	Documented exceptions differ from overridden method.
document_exception	Document checked exception {} using {}.
superflous_documented	Documented exceptions {} probably never thrown.

AutosarC++17_03-A15.4.6

Unchecked exceptions should not be specified together with a function declaration.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
autosar_exception_model	The default model for what counts as an checked / unchecked exception.	17_03
enforcement		automated
level		advisory
target		implementation
throw_marker	The command to document an exception to be thrown.	@throw

Possible Messages

Name	Message
dont_document_exception	Don't document unchecked exception.

AutosarC++17_03-A15.5.1

All user-provided class destructors, deallocation functions, move constructors, move assignment operators and swap functions shall not exit with an exception. A noexcept exception specification shall be added to these functions as appropriate.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
routine_may_except	Function must not exit with an exception.
routine_not_noexcept	Function shall be explicitly declared noexcept if appropriate.

AutosarC++17_03-A15.5.2

Program shall not be abruptly terminated. In particular, an implicit or explicit invocation of std::abort(), std::quick_exit(), std::_Exit(), std::terminate() shall not be done.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
allowed_exceptions	Exceptions that are allowed to escape from destructors.	['bad_alloc', 'bad_cast', 'failure', 'runtime_error', 'system_error']
blacklist	Dictionary of header globbing to (list of) function name globbing(s) of forbidden functions.	dict(...)
constructors	Whether to consider constructors.	False
destructors	Whether to consider destructors.	True
enforcement		partially automated
generateViolationPath	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignoreConstructorDestructor	Whether to ignore escaping exceptions from constructors and destructors.	False
ignoreUnknownRoutines	Whether to ignore extern or only declared routines.	False
inspectAtExitHandlers	Whether to also inspect at_exit() handlers-functions.	True
inspectThreadMain	Whether to also inspect thread main functions.	True
level		required
reportOnlyOneExceptionPerFunction	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
target		implementation

Possible Messages

Name	Message
exceptionEscapingConstructor	Escaping exception from constructor.
exceptionEscapingDestructor	Escaping exception from destructor.
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionEscapingMain	Uncaught exception escaping from main or additional entry point
exceptionSpecificationViolation	Exception violates function's exception-specification.
forbiddenLibfuncCall	Call to forbidden function.

AutosarC++17_03-A15.5.3

The terminate() function shall not be called implicitly.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
allowed_exceptions	Exceptions that are allowed to escape from destructors.	['bad_alloc', 'bad_cast', 'failure', 'runtime_error', 'system_error']
constructors	Whether to consider constructors.	False
destructors	Whether to consider destructors.	True
enforcement		automated
generateViolationPath	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignoreConstructorDestructor	Whether to ignore escaping exceptions from constructors and destructors.	True
ignoreThrowingFunctions	Whether to ignore noexcept specification violations on function that actually throw an exception.	False
ignoreUnknownRoutines	Whether to ignore extern or only declared routines.	False
inspectAtExitHandlers	Whether to also inspect at_exit() handlers-functions.	True
inspectThreadMain	Whether to also inspect thread main functions.	True
level		required
reportNoexceptFalseViolations	Whether to report cases where the function is declared noexcept(false), but no exceptions are ever thrown (directly or indirectly).	False
reportOnlyOneExceptionPerFunction	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
required	Dict which lists required operations per resource. The mapping gives each case a description which maps to a dict for key "Required_Functions", "Resource_Parameter_Empty".	dict(...)
resources	Configuration of resources and operations on them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)
target		implementation

Possible Messages

Name	Message
exceptionEscapingConstructor	Escaping exception from constructor.
exceptionEscapingDestructor	Escaping exception from destructor.
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionEscapingMain	Uncaught exception escaping from main or additional entry point
exceptionSpecificationViolation	Exception violates function's exception-specification.
implicitNoexceptSpecViolationWithout	Function implicitly declared noexcept(false) but no exceptions will be thrown.
noexceptSpecViolationWith	Exception violates function's noexcept-specification.
noexceptSpecViolationWithout	Function declared noexcept(false) but no exceptions will be thrown.
possiblyRequiredOperation	This thread is possibly joinable on destructor call
requiredOperation	This thread is joinable on destructor call

AutosarC++17_03-A16.0.1

The pre-processor shall only be used for file inclusion and include guards.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
accept_conditional_includes	Whether to accept #ifs for conditional includes.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_macro_definition	Macros are only allowed for include guards
line_directive	The pre-processor shall only be used for file inclusion and include guards.
object_macro_definition	Macros are only allowed for include guards
pp_if	Conditional compilation is only allowed for include guards
pragma	The pre-processor shall only be used for file inclusion and include guards.
undef	The pre-processor shall only be used for file inclusion and include guards.

AutosarC++17_03-A16.2.1

The ', ", /*, //, \ characters shall not occur in a header file name or in #include directive.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
check_quotation_mark	Whether system includes should be checked for containing a ".	True
enforcement		automated
forbidden	The substrings to check for. " will be added for system-includes.	set(["//", "\\", "", /*])
level		required
target		implementation

Possible Messages

Name	Message
nonstandard_include_character	Non-standard character in #include directive

AutosarC++17_03-A16.6.1

#error directive shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
error	Use of #error

AutosarC++17_03-A16.7.1

The #pragma directive shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pragma	Use of #pragma

AutosarC++17_03-A17.0.1

Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
additional_reserved_identifiers	Names listed here are seen as violations as well.	set(['assert', 'errno', 'defined'])
allow_reserved_identifier_as_include_guard	If True, macros used for include guards are not checked for being a reserved identifier.	False
check_for_keyword	Whether macro names being keywords should be reported as well.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
macro_having_reserved_name	Definition of reserved identifier or standard library element
undef_of_reserved_name	#undef of reserved identifier or standard library element

AutosarC++17_03-A18.0.1

The C library shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
c_lib_header	Include <{}> instead of <{}>.
cpp_lib_header_with_suffix	Include <{}> instead of <{}>.

AutosarC++17_03-A18.0.2

The library functions atof, atoi and atol from library <cstdlib> shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	stdlib
symbols	Names of symbols which are forbidden.	['atof', 'atoi', 'atol']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++17_03-A18.0.3

The library <clocale> (locale.h) and the setlocale function shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	locale
symbols	Names of symbols which are forbidden.	['setlocale']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++17_03-A18.1.1

C-style arrays should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
c_array_used	C-style arrays should not be used.

AutosarC++17_03-A18.1.2

The std::vector<bool> specialization shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
type_used	The std::auto_ptr type shall not be used.

AutosarC++17_03-A18.1.3

The std::auto_ptr type shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
type_used	The std::auto_ptr shall not be used.

AutosarC++17_03-A18.1.4

A pointer pointing to an element of an array of objects shall not be passed to a smart pointer of single object type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
type_used	{} shall not refer to an array type.

AutosarC++17_03-A18.1.5

The std::unique_ptr shall not be passed to a function by const reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
const_ref_unique_ptr	const std::unique_ptr& as parameter.

AutosarC++17_03-A18.5.1

Functions malloc, calloc, realloc and free shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_user_new_delete_operator	Whether to allow the library functions inside user defined new/delete operator overloads.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
stdlib_memory_function_call	Dynamic memory management function from <stdlib.h> is called.

AutosarC++17_03-A18.5.2

Operators new and delete shall not be called explicitly.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_for_types_and_subtypes	A list of fully-qualified type names for which the new and delete operators are allowed. This might be necessary if you are working with 3rd party libraries like Qt.	[]
enforcement		partially automated
level		required
target		implementation

Possible Messages

Name	Message
new_delete_call	Operators new and delete shall not be called explicitly.

AutosarC++17_03-A18.5.3

The form of delete operator shall match the form of new operator used to allocate the memory.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. See the base class for more options.	dict{...}
target		implementation

Possible Messages

Name	Message
possible_wrong_release	Resource possibly released using wrong function [allocation used {0}]
wrong_release	Resource released using wrong function [allocation used {0}]

AutosarC++17_03-A18.5.4

If a project has sized or unsized version of operator "delete" globally defined, then both sized and unsized versions shall be defined.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
only_sized	Only the sized version of operator delete is defined.
only_unsized	Only the unsized version of operator delete is defined.

AutosarC++17_03-A18.9.1

The std::bind shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	functional
symbols	Names of symbols which are forbidden.	['std::bind']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	False

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity std::bind from <>.

AutosarC++17_03-A18.9.2

Forwarding values to other functions shall be done via: (1) std::move if the value is an rvalue reference, (2) std::forward if the value is forwarding reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
forwarding_forwarding_reference	Use std::forward if the value is a forwarding reference.
forwarding_rvalue_reference	Use std::move if the value is a rvalue reference.

AutosarC++17_03-A18.9.3

The std::move shall not be used on objects declared const or const&.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
std_move_const	Call to std::move with argument declared const/const&.

AutosarC++17_03-A23.0.1

An iterator shall not be implicitly converted to const_iterator.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
implicit_const_iterator	An iterator shall not be implicitly converted to const_iterator.

AutosarC++17_03-M0.1.1

There shall be no unreachable code.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_dead_if_branch_with_const_literal_condition	Whether bodies of if-branches with constant literal conditions should be reported.	True
target		implementation

Possible Messages

Name	Message
unreachable_code	Unreachable code

AutosarC++17_03-M0.1.2

A project shall not contain infeasible paths.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_do_while_false	Whether do ... while[0] should be reported.	True
report_while_true	Whether while[1] ... should be reported.	True
target		implementation

Possible Messages

Name	Message
dead_false_branch	Condition is always true: Else branch is dead
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Parameter condition is always true: Else branch is dead
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Parameter comparison to NULL is always true: Else branch is dead
dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Parameter comparison to NULL is always false: Then branch is dead
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Parameter condition is always false: Then branch is dead
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Condition is always false: Then branch is dead
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Variable condition is always true: Else branch is dead
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Variable condition is always false: Then branch is dead
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead

AutosarC++17_03-M0.1.3

A project shall not contain unused variables.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
ignore_types	Variables of types named here are ignored in this check. Globbing patterns are supported.	[]
level		required
only_check_unit_locals	Whether only global static variables and local variables should be checked.	False
report_global_constants	Whether unused global constants should be reported.	False
report_undefined_variables	Whether only-declared variables should be reported.	True
target		implementation
treat_initialization_as_use	Whether an explicit initialization should be considered a use of the variable.	True
treat_side_effect_constructors_as_use	Whether variables should be seen as used if they are of a class type and initialized through a call to a constructor having a side-effect, e.g. std::lock_guard	False

Possible Messages

Name	Message
unused_field	Unused field
unused_variable	Unused variable

AutosarC++17_03-M0.1.4

A project shall not contain non-volatile POD variables having only one use.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_if_address_taken	Allow using a variable only once if that use involves taking the address of the variable.	False
enforcement		automated
level		required
report_fields	Select whether fields used only once should be reported as well.	True
target		implementation

Possible Messages

Name	Message
field_referenced_only_once	{ } referenced only once
unreferenced_initialized_field	{ } initialized but not referenced
unreferenced_initialized_variable	{ } initialized but not referenced
variable_used_only_once	{ } referenced only once

AutosarC++17_03-M0.1.5

A project shall not contain unused type declarations.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unused_type	Unused type declaration

AutosarC++17_03-M0.1.8

All functions with void return type shall have external side effect(s).

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
assume_effect_when_undefined	Whether all calls to undefined (external) functions should be seen as having a side-effect.	True
enforcement		automated
exceptions	Names of functions that should be excluded from the check.	main
exclude_constructors	If True, tolerate constructors with no side-effect.	False
exclude_virtual_destructors	If True, tolerate virtual destructors with no side-effect.	False
level		required
target		implementation

Possible Messages

Name	Message
void_func_without_side_effect	Void function has no external side-effect

AutosarC++17_03-M0.1.9

There shall be no dead code.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
allow_void_var	Whether {void}var; should be allowed or reported.	True
assume_effect_for_all_calls	Whether all calls should be seen as having a side-effect.	False
assume_effect_when_undefined	Whether all calls to undefined [external] functions should be seen as having a side-effect.	True
enforcement		automated
level		required
report_dead_initializations	Whether initialisations may be reported as dead.	True
report_do_while_false	Whether do ... while(0) should be reported.	True
report_while_true	Whether while(1) ... should be reported.	True
target		implementation
tolerate_void_cast	Whether a {void} cast is accepted.	True

Possible Messages

Name	Message
conditional_unused_def	Result of assignment is not used along some path(s) (disabled)
dead_false_branch	Redundant code, condition is always true
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Redundant code, parameter condition is always true
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Redundant code, parameter comparison to NULL is always true
dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Redundant code, parameter comparison to NULL is always false
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Redundant code, parameter condition is always false
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Redundant code, condition is always false
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Redundant code, variable condition is always true
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Redundant code, variable condition is always false
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead
init_used_in_other_isr	Initialization is only used in some interrupt handler
no_effect	Non-null statement without side-effect
removable_declaration	Declaration can be removed
removable_statement	Statement can be removed
unused_def	Dead (redundant) code: result of assignment is not used
unused_init	Unused initialization
used_in_other_isr	Result of assignment is only used in some interrupt handler

AutosarC++17_03-M0.1.10

Every defined function shall be called at least once.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
ignore_copy_assignment_operators	Whether uncalled copy assignment operators are allowed. Other rules like the "rule of three" might require copy assignment operators as part of good class design, even if there are currently no callers of the copy assignment operator.	False
ignore_copy_constructors	Whether uncalled copy constructors are allowed. Other rules like the "rule of three" might require copy constructors as part of good class design, even if there are currently no callers of the copy constructor.	False
level		required
only_check_internal_or_private_methods	Whether only function with internal linkage or private functions should be checked.	False
only_check_static_or_private_methods	Whether only static functions or private functions should be checked.	False
only_check_unit_locals	Whether only global static functions should be checked.	False
target		implementation

Possible Messages

Name	Message
defined_function_not_called	Defined function is not called

AutosarC++17_03-M0.2.1

An object shall not be assigned to an overlapping object.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
assignment_to_overlapping	An object shall not be assigned to an overlapping object

AutosarC++17_03-M0.3.1

Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
abstract_interpretation_div_by_zero	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
abstract_interpretation_overflow	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
argument_checks	Configuration of [dis]allowed values in addition to what iranalysis.config provides. This is a dict key -> dict with keys 'Functions', 'Parameter', and 'Allowed'/'Disallowed'. The parameter number counting starts at 0. Arguments passed in for this parameters are checked against the value specification given with '[Dis]allowed'.	dict(...)
enforcement		automated
execute_checks_covered_in_other_rules	Whether iranalysis-based checks covered by other Misra-C++:2008 rules should be executed here as well.	False
forbidden	Dict which lists forbidden operations per resource. The mapping gives each case a description which maps to a dict for key "Forbidden_Functions", "Mode_Parameter", "Mode".	dict(...)
level		required
modes	Describes when an allocating call opens a file in read or write mode. This is a dict resource -> dict with keys "Resource_Parameter", "Check_Write_Write", and "Modes" (being a dict with key "Read" and "Write")	dict(...)
perform_global_analysis	Whether global analysis (using iranalysis) should be done, or just a local analysis.	True
report_dead_initializations	Whether initialisations may be reported as dead.	True
report_do_while_false	Whether do ... while[0] should be reported.	True
report_while_true	Whether while[1] ... should be reported.	True
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)
target		implementation

Possible Messages

Name	Message
argumentViolation	Argument not within allowed values
arithmetic_out_of_bounds	Pointer arithmetic on {1} creates pointer outside array bounds of {0}
asgn_not_an_enumerator	Assignment of value which does not correspond to an enumerator
asgn_out_of_enum_range	Assignment of value outside the range of the enumerators
cast_overflow	Cast on result of arithmetic computation may cause overflow
cast_truncate	Cast may truncate value
cast_underflow	Cast on result of arithmetic computation may cause underflow
conditional_unused_def	Result of assignment is not used along some path(s)
deadCatch	Dead exception handler
dead_false_branch	Condition is always true: Else branch is dead
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Parameter condition is always true: Else branch is dead
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Parameter comparison to NULL is always true: Else branch is dead

dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Parameter comparison to NULL is always false: Then branch is dead
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Parameter condition is always false: Then branch is dead
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Condition is always false: Then branch is dead
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Variable condition is always true: Else branch is dead
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Variable condition is always false: Then branch is dead
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead
division_by_zero	Division by zero
double_free	Dynamic memory released here was already released earlier
escaping_address	Escaping address of local variable
forbidden_operation	The way in which this resource was allocated forbids this operation
init_used_in_other_isr	Initialization is only used in some interrupt handler
memory_leak	Call allocates leaking memory
mode_conflict	Same file used for both reading and writing
modulo_by_zero	Modulo by zero
null_deref	Pointer is NULL at dereference
out_of_bounds	Access into array is out of bounds
overflow	Arithmetic computation may cause overflow
pass_as_pointer_to_const_param	Passing uninitialized variable by pointer as function parameter with pointer-to-const type
possible_argumentViolation	Argument possibly not within allowed values
possible_division_by_zero	Possible division by zero
possible_double_free	Dynamic memory released here possibly already released earlier
possible_indirect_out_of_bounds	Pointer-indirect access through {1} might be out of bounds accessing {0}
possible_memory_leak	Call allocates possibly leaking memory
possible_mode_conflict	Same file possibly used for both reading and writing
possible_modulo_by_zero	Possible modulo by zero
possible_null_deref	Pointer may be NULL at dereference
possible_out_of_bounds	Access into array might be out of bounds
possible_stack_free	{ } possibly released by call to { } is a stack object
possible_uninit	Use of possibly uninitialized variable
possible_unrelated_ptr_comparison	Comparing possibly unrelated pointers
possible_unrelated_ptr_subtraction	Subtracting possibly unrelated pointers
possible_use_after_free	Dynamic memory possibly used after it was previously released

possible_wrong_release	Resource possibly released using wrong function (allocation used {0})
possiblyEscapingAddress	Possibly escaping address of local variable (as target of {1})
possiblyForbiddenOperation	The way in which this resource was allocated possibly forbids this operation
stack_free	{ } released by call to { } is a stack object
underflow	Arithmetic computation may cause underflow
uninit	Use of uninitialized variable
unrelated_ptr_comparison	Comparing unrelated pointers
unrelated_ptr_subtraction	Subtracting unrelated pointers
unused_def	Result of assignment is not used
unused_init	Unused initialization
use_after_free	Dynamic memory used after it was previously released
used_in_other_isr	Result of assignment is only used in some interrupt handler
wrong_release	Resource released using wrong function (allocation used {0})

AutosarC++17_03-M0.3.2

If a function generates error information, then that error information shall be tested.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
inspect_template_instances	Whether calls in template instances should be reported.	False
level		required
relevant_functions	If provided, only calls to these functions are inspected.	[]
target		implementation

Possible Messages

Name	Message
discarded_error_with_entity	Integral return value (possibly error code) discarded.
discarded_error_without_entity	Integral return value (possibly error code) discarded.

AutosarC++17_03-M0.4.2

Use of floating-point arithmetic shall be documented.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
float_arithetic	Use of floating-point arithmetic

AutosarC++17_03-M2.10.1

Different identifiers shall be typographically unambiguous.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
distinguish_c_name_spaces	Whether to allow similar identifiers in different C name spaces (type tag/label/member/ordinary identifier)	True
distinguish_macro_names	Whether to allow macros to same similar names as other identifiers.	False
distinguish_type_names	Whether to allow types to have similar names as other identifiers. For example, a variable declaration "MyClass myClass;" is allowed by this option. Note: unlike the option 'distinguish_c_name_spaces', this option also distinguishes typedef names, not just struct/class tag names.	False
enforcement		automated
exclude_system_headers	Whether to exclude identifiers appearing in system headers	False
level		required
normalizations	Which pairs of characters should be seen as ambiguous	[('0', 'O'), ('1', 'l'), ('I', 'l'), ('i', 'l'), ('5', 'S'), ('2', 'Z'), ('8', 'B'), ('rn', 'm'), ('n', 'h'), ('_', '')]
target		implementation

Possible Messages

Name	Message
ambiguous	Identifiers are typographically ambiguous
casing	Identifiers only differ in casing

AutosarC++17_03-M2.10.3

A typedef name (including qualification, if any) shall be a unique identifier.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation
tolerate_macros	Whether #define and #undef using the typedef's name is allowed	False
tolerate_typedef_entity	Whether the entity forming the typedef's underlying type can have the same name.	False

Possible Messages

Name	Message
reused_typedef	Typedef name reused.

AutosarC++17_03-M2.10.6

If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
invert_findings	Whether to invert primary and secondary SLocs for violations	False
level		required
target		implementation

Possible Messages

Name	Message
reused_type	Type name reused in same scope

AutosarC++17_03-M2.13.2

Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
octal_escape_sequence	Use of octal escape sequence.
octal_literal	Use of octal literal.

AutosarC++17_03-M2.13.3

A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_only_hex_and_octal	Whether only hex and octal literals should be checked.	True
consider_conversion	Selects whether literals with an implicit conversion from signed to unsigned are reported.	False
consider_pp_expressions	Whether literals in #if conditions are checked.	False
enforcement		automated
level		required
report_all_unsuffixed_hex_and_octal	Whether hex and octal literals should always be reported when they are unsuffixed, independent of the environment	False
target		implementation

Possible Messages

Name	Message
missing_hex_suffix	Hexadecimal and octal constants require "U" suffix
missing_u_suffix	Constant of unsigned type requires "U" suffix

AutosarC++17_03-M2.13.4

Literal suffixes shall be upper case.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
lowercase_suffix	Literal suffix should be upper case

AutosarC++17_03-M3.1.2

Functions shall not be declared at block scope.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
local_function_declaration	Functions shall not be declared at block scope.

AutosarC++17_03-M3.2.1

All declarations of an object or function shall have compatible types.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_parameter_type_qualifier_differences	If enabled, do not report a violation for different top-level type qualifiers in parameters: int f(int p); int f(const int p) {} Top-level qualifiers are only relevant for the implementation of the function and have no effect on callers, so it is not necessary to replicate them in the header file.	False
check_parameter_types	Whether parameter types should be compared	False
check_undefined	Whether only-declared routines and variables should also be checked.	True
enforcement		automated
level		required
require_exact_match	Whether to check for identical or compatible types (for routine return types).	False
target		implementation

Possible Messages

Name	Message
decl_return_type_mismatch	Return type at declarations differs
decl_type_mismatch	Type at declarations differs
parameter_type_mismatch	Parameter type at definition differs from type at declaration
return_type_mismatch	Return type at definition differs from type at declaration
type_mismatch	Type at definition differs from type at declaration

AutosarC++17_03-M3.2.2

The One Definition Rule shall not be violated.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
class_struct_difference	{}
different_enumerators	{}
different_field_types	{}
different_fields	{}
general_odrViolation	{}

AutosarC++17_03-M3.2.3

A type, object or function that is used in multiple translation units shall be declared in one and only one file.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
allow_only_one_decl	Enables message if there are multiple declarations but all in same file.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_declared_in_multiple_files	Function declared in more than one file.
function_declared_multiple_times	Function declared more than once.
type_declared_in_multiple_files	Type declared in more than one file.
type_declared_multiple_times	Type declared more than once.
variable_declared_in_multiple_files	Variable declared in more than one file.
variable_declared_multiple_times	Variable declared more than once.

AutosarC++17_03-M3.2.4

An identifier with external linkage shall have exactly one definition.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
allow_undefined_specialized_template	Select whether undefined templates should be reported if specializations of them exist.	True
allow_unselected_declarations	Select whether to report declarations with no definition if they are unused.	True
allowed_undefined_functions	Regular expressions for functions which are tolerated without a definition, e.g. compiler intrinsics.	['_.*']
allowed_undefined_types	Regular expressions for types which are tolerated without a definition.	['_.*']
allowed_undefined_variables	Regular expressions for variables which are tolerated without a definition.	['_.*']
check_composite_types	Check class/struct/union types for having no definition even if they have no external linkage.	False
check_types_only_with_external_linkage	Whether the types should only be checked when they have external linkage	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
composite_defined_multiple_times	Composite type has multiple definitions.
enum_defined_multiple_times	Enumeration type has multiple definitions.
function_defined_multiple_times	Type, object or function has multiple definitions.
undefined_function	Type, object or function without definition
undefined_type	Type without definition
undefined_variable	Type, object or function without definition
variable_defined_multiple_times	Type, object or function has multiple definitions.

AutosarC++17_03-M3.3.2

If a function has internal linkage then all re-declarations shall include the static storage class specifier.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_decl_missing_static	Declaration should use static storage class specifier
function_def_missing_static	Definition should use static storage class specifier
variable_decl_missing_static	Declaration should use static storage class specifier
variable_def_missing_static	Definition should use static storage class specifier

AutosarC++17_03-M3.4.1

An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_moving_to_other_primary_file	When a variable is only used in a single file X, but currently implemented in a different file Y, this controls whether the check suggests to move it into X and make it static there.	False
check_classes	Whether to report structs/classes/unions which are only used in a single function or file	True
check_loop_counter	Whether to report for-loop counters being declared before the loop that could be declared in the for-init part instead from C99 on	False
enforcement		automated
exclude_function_locals	If True, variables that could even be function-local are not reported.	True
exclude_undefined	Whether only-declared symbols should be reported as well	True
level		required
move_global_const_into_function	Controls suggestions for global constants that could be declared locally in a function.	True
only_check_unit_locals	Whether only global static variables should be checked.	False
target		implementation

Possible Messages

Name	Message
locality_block	{ } can be declared in a more local scope.
locality_file	{ } can be declared locally in primary file.
locality_function	Global { } can be declared inside function.
locality_loop_init	{ } can be declared in the for-loop's initialization.
var_file_static	{ } can be declared static in primary file.

AutosarC++17_03-M3.9.1

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_missing_qualifiers	If True, tolerate differences in the use of explicit namespace/class qualifiers.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
parameter_type_tokens_mismatch	Type of redeclaration is not token-for-token identical
return_type_tokens_mismatch	Type of redeclaration is not token-for-token identical
variable_type_tokens_mismatch	Type of redeclaration is not token-for-token identical

AutosarC++17_03-M3.9.3

The underlying bit representations of floating-point values shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
float_bit_representation	Use of bit representation of a float value.

AutosarC++17_03-M4.5.1

Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_sizeof	Allows the use of sizeof(bool_var). The Misra rule forbids all uses of expressions of type bool except for those explicitly allowed; and does not allow sizeof. However this is likely an oversight; there is no good reason to prevent the use of sizeof(bool_var). This option also allows alignof(bool_var) in C++11.	False
allow_unevaluated_operator	Allows the use of boolean expressions in contexts where the expression is not evaluated. As of C++17, these are: typeid(bool_var), noexcept(bool_var) and decltype(bool_var). sizeof(bool_var) is controlled by the separate option 'allow_sizeof'.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bool_operand_in_bad_operator	Use of boolean operand in '{}' operator
bool_operand_outside_logical_and_relational_op	Use of boolean operand with integral promotion

AutosarC++17_03-M4.5.3

Expressions with type [plain] char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
char_operand_outside_comparison	Use of character operand in forbidden context

AutosarC++17_03-M4.10.1

NULL shall not be used as an integer value.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
null_as_int	Use of NULL as integer value

AutosarC++17_03-M4.10.2

Literal zero (0) shall not be used as the null-pointer-constant.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
require_nullptr	Require C++11 nullptr instead of macro NULL or literal zero (0).	False
target		implementation

Possible Messages

Name	Message
zero_as_null	Use of literal zero (0) as null-pointer-constant, use {} instead

AutosarC++17_03-M5.0.2

Limited dependence should be placed on C++ operator precedence rules in expressions.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_algebraic_order	Whether parens can be omitted for multiplicative operators as operands in additive operators	False
allow_redundant_parentheses_in_logical_operator	Allow redundant parentheses around unary operators when they appear as operand of '&&' or ' '. This option can be used to avoid a conflict with MisraC++-5.2.1, which requires parentheses in these instances.	True
enforcement		automated
level		required
report_unnecessary_parentheses	Controls whether unnecessary use of parentheses on the right side of assignments or around unary operators are reported.	True
target		implementation

Possible Messages

Name	Message
assignment_rhs_in_parens	No parentheses required for right-hand side of assignment
missing_parens_depends_on_precedence	Parentheses required to avoid dependence on precedence rules
unary_op_in_parens	No parentheses required for unary operator

AutosarC++17_03-M5.0.3

A value expression shall not be implicitly converted to a different underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes], [SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
target		implementation
type_category	Selection of type categories to consider for type changes inside same category.	[SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Result of cvalue expression implicitly converted to different underlying type
cast_changes_type_inside_category	Result of cvalue expression implicitly converted to different underlying type

AutosarC++17_03-M5.0.4

An implicit integral conversion shall not change the signedness of the underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_constant_conversions	If this option is enabled, the rule is relaxed to allow implicit conversions of constant integer expressions whenever the constant value fits into the target type.	True
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
report_signed_to_unsigned	Whether conversion from signed to unsigned shall be reported.	True
report_unsigned_to_signed	Whether conversion from unsigned to signed shall be reported.	True
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_from_signed_to_unsigned	Implicit integral conversion changes signedness of underlying type
cast_from_unsigned_to_signed	Implicit integral conversion changes signedness of underlying type

AutosarC++17_03-M5.0.5

There shall be no implicit floating-integral conversions.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes], [FloatingTypes]], [[FloatingTypes], [SignedTypes, UnsignedTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Implicit floating-integral conversion

AutosarC++17_03-M5.0.6

An implicit integral or floating-point conversion shall not reduce the size of the underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
report_narrowing	Whether narrowing casts should be reported.	True
report_widening	Whether widening casts should be reported.	False
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
size_source_category	Selection of type categories of the operand being cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
size_target_category	Selection of type categories appearing as target types of the cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
narrowing_cast	Implicit conversion reduces size of underlying type
widening_cast	Conversion to larger type

AutosarC++17_03-M5.0.7

There shall be no explicit floating-integral conversions of a cvalue expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, CharacterTypes], [FloatingTypes]], [[FloatingTypes], [SignedTypes, UnsignedTypes, CharacterTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Explicit floating-integral conversion of cvalue expression

AutosarC++17_03-M5.0.8

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
report_narrowing	Whether narrowing casts should be reported.	False
report_widening	Whether widening casts should be reported.	True
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
size_source_category	Selection of type categories of the operand being cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
size_target_category	Selection of type categories appearing as target types of the cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
narrowing_cast	Conversion to smaller type
widening_cast	Explicit conversion increases size of underlying type of cvalue expression

AutosarC++17_03-M5.0.9

An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
report_signed_to_unsigned	Whether conversion from signed to unsigned shall be reported.	True
report_unsigned_to_signed	Whether conversion from unsigned to signed shall be reported.	True
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_from_signed_to_unsigned	Explicit conversion changes signedness of underlying type of cvalue expression
cast_from_unsigned_to_signed	Explicit conversion changes signedness of underlying type of cvalue expression

AutosarC++17_03-M5.0.10

If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitop_small_without_cast	Bitwise operator requires cast to underlying type on result

AutosarC++17_03-M5.0.11

The plain char type shall only be used for the storage and use of character values.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
char_used_numerically	Plain char datatype used for non-character data.

AutosarC++17_03-M5.0.12

signed char and unsigned char type shall only be used for the storage and use of numeric values.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
numeric_char_used_as_character	Signed/unsigned char datatype used for character data.

AutosarC++17_03-M5.0.14

The first operand of a conditional-operator shall have type bool.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
nonbool_conditional_operator_condition	Condition must have type bool

AutosarC++17_03-M5.0.15

Array indexing shall be the only form of pointer arithmetic.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
indexing_only_on_identifiers	Report array indexing on pointers only for variables (`ptr[i]`), not for other pointer expressions (e.g. `get_ptr()[i]`). This option is meant to suppress the violations introduced by the BAUHAUS-12021 bugfix in version 6.9.6.	False
level		required
target		implementation

Possible Messages

Name	Message
array_indexing_on_pointer	Array indexing only allowed for arrays
pointer_arithmetic	Pointer arithmetic not allowed
pointer_increment_decrement	Pointer arithmetic not allowed

AutosarC++17_03-M5.0.16

A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
arithmetic_on_nonarray	Pointer arithmetic on {1} operates on non-array target {0}
arithmetic_out_of_bounds	Pointer arithmetic on {1} creates pointer outside array bounds of {0}
out_of_bounds	Access into array is out of bounds
possible_indirect_out_of_bounds	Pointer-indirect access through {1} might be out of bounds accessing {0}
possible_out_of_bounds	Access into array might be out of bounds

AutosarC++17_03-M5.0.17

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
possible_unrelated_ptr_subtraction	Subtracting possibly unrelated pointers
unrelated_ptr_subtraction	Subtracting unrelated pointers

AutosarC++17_03-M5.0.18

>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
possible_unrelated_ptr_comparison	Comparing possibly unrelated pointers
unrelated_ptr_comparison	Comparing unrelated pointers

AutosarC++17_03-M5.0.20

Non-constant operands to a binary bitwise operator shall have the same underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitop_with_different_operand_types	Non-constant operands of bitwise operator differ in underlying type
shortcut_bitop_with_different_operand_types	Non-constant operands of bitwise operator differ in underlying type

AutosarC++17_03-M5.0.21

Bitwise operators shall only be applied to operands of unsigned underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitop_on_enum	Bitwise operator on enum underlying type
bitop_on_signed	Bitwise operator on signed underlying type

AutosarC++17_03-M5.2.1

Each operand of a logical && or || shall be a postfix-expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allowed	Can be used to name additional allowed IR node types for the operands.	None
enforcement		automated
level		required
require_postfix_epression	Whether postfix or primary expressions are required as operands.	True
target		implementation

Possible Messages

Name	Message
missing_parens_for_logical_operator_operand	Operand of && or must be parenthesized.

AutosarC++17_03-M5.2.2

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cannot_cast_virtual_base	Cannot convert pointer to base class {} to pointer to derived class {} -- base class is virtual
missing_dynamic_cast_on_virtual_base	Use dynamic_cast on virtual base class

AutosarC++17_03-M5.2.3

Casts from a base class to a derived class should not be performed on polymorphic types.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cast_from_polybase_to_derived	Cast from polymorphic base class to derived class

AutosarC++17_03-M5.2.6

A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[FunctionPointerTypes], [VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes, NullPointerTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_category	Selection of type categories to consider for type changes inside same category.	[FunctionPointerTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion of function pointer to other type
cast_changes_type_inside_category	Conversion of function pointer to other function pointer type

AutosarC++17_03-M5.2.8

An object with integer type or pointer to void type shall not be converted to an object with pointer type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, VoidPointerTypes], [ObjectPointerTypes, IncompletePointerTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, $(T^*)(\text{void}^*)x$ will not be reported.	True
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion from void* or integer to pointer type

AutosarC++17_03-M5.2.9

A cast should not convert a pointer type to an integral type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories. [[[VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes], [SignedTypes, UnsignedTypes, BoolTypes, EnumTypes, CharacterTypes]]]	
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion between pointer and integral type

AutosarC++17_03-M5.2.10

The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
forbid_all_operators	If True, forbids mixing with any kind of operator; else only with arithmetic operators.	False
level		required
target		implementation

Possible Messages

Name	Message
increment_mixed_with_operator	Increment or decrement mixed with other operators

AutosarC++17_03-M5.2.11

The comma operator, `&&` operator and the `||` operator shall not be overloaded.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
invalid	Selection of disallowed operator overloads by name.	<code>['operator&&', 'operator ', 'operator,']</code>
invalid_kinds	Selection of disallowed operator overloads by operator kind.	<code>[]</code>
level		required
target		implementation

Possible Messages

Name	Message
forbidden_operator_overload	Overloaded version of comma operator or <code>&&</code> or <code> </code>

AutosarC++17_03-M5.2.12

An identifier with array type passed as a function argument shall not decay to a pointer.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
decay	Array to pointer decay

AutosarC++17_03-M5.3.1

Each operand of the `!` operator, the logical `&&` or the logical `||` operators shall have type `bool`.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_in_conditions	If True, only logical operators inside conditions are checked.	False
target		implementation

Possible Messages

Name	Message
nonbool_logical_operator_operand	Operand of logical operator shall be of type bool

AutosarC++17_03-M5.3.2

The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_literals	If True, integer literals are also disallowed as operands.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unary_minus_on_unsigned	Unary minus applied to unsigned

AutosarC++17_03-M5.3.3

The unary & operator shall not be overloaded.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
invalid	Selection of disallowed operator overloads by name.	[]
invalid_kinds	Selection of disallowed operator overloads by operator kind.	[7]
level		required
target		implementation

Possible Messages

Name	Message
forbidden_operator_overload	Overloaded version of unary &

AutosarC++17_03-M5.3.4

Evaluation of the operand to the sizeof operator shall not contain side effects.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
side_effect_in_sizeof	Operand of "sizeof" shall not contain side effects

AutosarC++17_03-M5.8.1

The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_nonconst_cases	Whether a message should be printed for non-constant cases missing a range-checking if statement	False
target		implementation
use_essential_type	Whether essential type should be computed for allowed value range.	False

Possible Messages

Name	Message
excessive_shift	Right operand of shift operator has value {}, but should be in range 0 to {}
possible_excessive_shift	Right operand of shift operator should be checked to be in range 0 to {}

AutosarC++17_03-M5.14.1

The right-hand operand of a logical && or || operator shall not contain side effects.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_volatile_access	Whether access to a volatile variable should be reported as side effect.	False
assume_effect_when_undefined	Whether all calls to undefined (external) functions should be seen as having a side-effect.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
call_with_side_effect	Call in right-hand operand of && or may have side-effect
modifies_local_var	Right-hand operand of && or modifies '{}'
side_effect	Right-hand operand of && or has side-effect

AutosarC++17_03-M5.17.1

The semantic equivalence between a binary operator and its assignment operator form shall be preserved.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_call_relation	If True, checks whether there is a call relation between binary and assignment version.	True
enforcement		automated
ignore_stream_operators	If True, allows definitions of operator<<() and operator>>() without the corresponding assignment operator. Note this will also allow operator<<=() and operator>>=() as they no longer have an equivalent binary form.	False
level		required
target		implementation

Possible Messages

Name	Message
missing_assignment_version	Missing overload for corresponding assignment version of operator
missing_binary_version	Missing overload for corresponding binary version of operator
missing_call_to_assignment_version	There is no call relation between this operator and its assignment version to ensure semantic equivalence
missing_call_to_binary_version	There is no call relation between this operator and its binary version to ensure semantic equivalence

AutosarC++17_03-M5.18.1

The comma operator shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_comma	Use of comma sequence operator.

AutosarC++17_03-M5.19.1

Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
const_expression_overflow	Overflow during evaluation of constant expression.

AutosarC++17_03-M6.2.1

Assignment operators shall not be used in sub-expressions.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
assignment_result_used	Assignment inside sub-expression.

AutosarC++17_03-M6.2.2

Floating-point expressions shall not be directly or indirectly tested for equality or inequality.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
float_equality_comparison	Testing a float for (in)equality.
float_indirect_equality_comparison	Indirectly testing a float for (in)equality.

AutosarC++17_03-M6.2.3

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_empty_macros	Whether a macro invocation before the ; is allowed if it expands to nothing.	False
allow_nonempty_macros	Whether a non-empty macro invocation before the ; is allowed.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
null_statement_not_isolated	Null statement not on a line by itself

AutosarC++17_03-M6.3.1

The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
loop_body_not_compound	Loop body shall be a statement sequence.
switch_body_not_compound	switch body shall be a statement sequence.

AutosarC++17_03-M6.4.1

An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
dangerous_else	Block or if statement required as else branch.
dangerous_then	Block required as then branch.

AutosarC++17_03-M6.4.2

All if ... else if constructs shall be terminated with an else clause.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
defensive_else	Controls if the else branch of if-cascade's should contain a suitable comment or not be empty.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
non_defensive_else	Last else in the if-cascade must not be empty or contain a comment.
unterminated_if_cascade	Last if in the if-cascade has no else.

AutosarC++17_03-M6.4.3

A switch statement shall be a well-formed switch statement.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
acceptable_end_items	This set characterizes allowed last items at the end of a switch case	('Exit_Switch', 'Throw_Expression')
acceptable_middle_items	This set characterizes allowed last items in the middle of a switch case	('Exit_Switch', 'Nondefault_Case_Label', 'Throw_Expression')
allow_as_first	Whether to allow the "default" case as the first case in a switch.	False
allow_complete_enum	Whether a switch having a case for all enumerators is tolerated without a default.	True
allow_empty_statements	Whether empty statements at the end of switch-case should be ignored.	False
allow_fallthrough_comment	Whether a comment with text "fall" and "through" is allowed to mark fallthrough.	True
allow_fallthrough_macro	Whether a macro called __fallthrough is allowed to mark fallthrough.	True
allow_if	Whether an if containing acceptable items at the end of both branches is accepted.	False
allow_noreturn_function_calls	Whether a function call to a [[noreturn]] function is considered a valid termination.	False
comment	The comment that indicates an intentional fallthrough. Can be a list of comments.	[]
disallow_multiple_compound_statements	Whether there can be more than one toplevel compound statement.	True
disallowed_nested_statements	Statement types not allowed inside compound statements within case-clause body.	None
disallowed_toplevel_statements	Statement types not allowed in case-clause body on toplevel.	('Named_Label', 'Unconditional_Branch', 'Declaration_Statement')
enforcement		automated
level		required
minimum_switch_cases	The number of cases a switch statement should at least have.	1
target		implementation

Possible Messages

Name	Message
case_label_nested	A switch label shall only be used in a switch.
default_not_last	"default" clause is not last in switch.
empty_default	Default clause should contain a statement or comment.
empty_switch	Switch has no "case" clause.
forbidden_compound_statement_in_case	At most one compound statement is allowed on toplevel in case-clauses.
forbidden_nested_statement_in_case	Statement is not allowed inside compound statement in case-clauses.
forbidden_toplevel_statement_in_case	Statement is not allowed outside a compound statement in case-clauses.
minimum_switch_cases	Switch has too little "case" clauses.
missing_default	Switch has no "default" clause.
switch_body_not_compound	A switch statement shall be a well-formed switch statement.
switch_not_well_formed	Switch has a statement before the first case label.
switch_over_bool	Switch expression is effectively Boolean.
unterminated_case	This switch-case is not terminated with {}, last statement is {}.
unterminated_empty_case	This switch-case is empty, not terminated with {}.

AutosarC++17_03-M6.4.4

A switch label shall only be used when the most closely-enclosing compound-statement is the body of a switch-statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
case_label_nested	A switch label shall only be used in a switch.

AutosarC++17_03-M6.4.5

An unconditional throw or break statement shall terminate every non-empty switch-clause.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
acceptable_end_items	This set characterizes allowed last items at the end of a switch case	('Exit_Switch', 'Throw_Expression')
acceptable_middle_items	This set characterizes allowed last items in the middle of a switch case	('Exit_Switch', 'Nondefault_Case_Label', 'Throw_Expression')
allow_empty_statements	Whether empty statements at the end of switch-case should be ignored.	False
allow_fallthrough_comment	Whether a comment with text "fall" and "through" is allowed to mark fallthrough.	True
allow_fallthrough_macro	Whether a macro called __fallthrough is allowed to mark fallthrough.	True
allow_if	Whether an if containing acceptable items at the end of both branches is accepted.	False
allow_noreturn_function_calls	Whether a function call to a [[noreturn]] function is considered a valid termination.	False
comment	The comment that indicates an intentional fallthrough. Can be a list of comments.	[]
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unterminated_case	This switch-case is not terminated with {}, last statement is {}.
unterminated_empty_case	This switch-case is empty, not terminated with {}.

AutosarC++17_03-M6.4.6

The final clause of a switch statement shall be the default clause.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_as_first	Whether to allow the "default" case as the first case in a switch.	False
allow_complete_enum	Whether a switch having a case for all enumerators is tolerated without a default.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
default_not_last	"default" clause is not last in switch.
empty_default	Default clause should contain a statement or comment.
missing_default	Switch has no "default" clause.

AutosarC++17_03-M6.4.7

The condition of a switch statement shall not have bool type.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
switch_over_bool	Switch condition shall not have bool type.

AutosarC++17_03-M6.5.2

If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
stepping_loop_uses_equality_check	Loop-counter shall not be tested with equality operator if not modified by -- or ++

AutosarC++17_03-M6.5.3

The loop-counter shall not be modified within condition or statement.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
loop_counter_modified_in_condition	Loop-counter shall not be modified within condition
modified_loop_counter	Loop-counter shall not be modified within loop body

AutosarC++17_03-M6.5.4

The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_constexpr	Allow constexpr as a constant <n>.	True
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
nonconst_loop_increment	Loop-counter shall be modified by one of: --, ++, -=n, or +=n (with constant n)

AutosarC++17_03-M6.5.5

A loop-control-variable other than the loop-counter shall not be modified within condition or expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
modified_loop_control_variable	Loop-control variable (other than counter) shall not be modified within condition or expression

AutosarC++17_03-M6.5.6

A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
nonbool_loop_control_variable	Loop-control variable (other than counter) shall have type bool

AutosarC++17_03-M6.6.1

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
goto_label_block	Label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

AutosarC++17_03-M6.6.2

The goto statement shall jump to a label declared later in the same function body.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
backwards_goto	Label referenced by a goto statement shall be declared later in same function.

AutosarC++17_03-M6.6.3

The continue statement shall only be used within a well-formed for loop.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
report_outside_for_loops	Whether to report continue statements in do..while/while loops	True
target		implementation

Possible Messages

Name	Message
continue_in_bad_loop	The continue statement shall only be used within a well-formed for loop

AutosarC++17_03-M7.1.2

A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_all_variables	Whether non-parameter variables should be checked as well.	False
enforcement		automated
ignore_forwarding_references	Whether to allow C++11 forwarding references (universal references) even if all template instances would also work with a const lvalue reference.	True
ignore_move_constructors	Whether to allow non-const rvalue references on constructors.	True
level		required
report_overrides	Whether to report violations on overrides. For virtual methods, this rule will report a violation only if the parameter can be marked const in all related methods in the inheritance hierarchy. Thus, the same violation is reported for each method in the inheritance hierarchy. You can set this option to False to report such violations only for the base methods.	True
target		implementation

Possible Messages

Name	Message
parameter_can_point_to_const	{ } can be declared as pointer/reference to const.

AutosarC++17_03-M7.3.1

The global namespace shall only contain main, namespace declarations and extern "C" declarations.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
global_symbol	Symbol not allowed in global namespace.

AutosarC++17_03-M7.3.2

The identifier main shall not be used for a function other than the global function main.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
nonglobal_function_named_main	The identifier main shall not be used for a function other than the global function main

AutosarC++17_03-M7.3.3

There shall be no unnamed namespaces in header files.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unnamed_namespace_in_header	Unnamed namespaces in header file

AutosarC++17_03-M7.3.4

Using-directives shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
using_directive	Using-directives shall not be used

AutosarC++17_03-M7.3.5

Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_other_usages	Whether to find other usages (such as function calls, or taking a fp-reference) between multiple declarations for an identifier.	False
target		implementation

Possible Messages

Name	Message
declarations_surrounding_usage	Declarations straddle a usage
declarations_surrounding_using	Declarations straddle a using-declaration

AutosarC++17_03-M7.3.6

using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
using_declaration_in_header	Using-declaration in header file
using_namespace_in_header	Using-directive in header file

AutosarC++17_03-M7.4.1

All usage of assembler shall be documented.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_assembler	Usage of assembler shall be documented

AutosarC++17_03-M7.4.2

Assembler instructions shall only be introduced using the asm declaration.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pragma_asm	Assembler instructions shall only be introduced using the asm declaration

AutosarC++17_03-M7.4.3

Assembly language shall be encapsulated and isolated.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
asm_not_encapsulated	Assembly language shall be encapsulated and isolated.

AutosarC++17_03-M7.5.1

A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
global_reference_to_local_var	Address of local variable escapes via global variable.
parameter_reference_to_local_var	Address of local variable escapes via parameter.
returning_reference_to_local_var	Returning reference/pointer to local variable.

AutosarC++17_03-M7.5.2

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
allow_longer_living_local	Whether assignment to a longer-living local variable should be accepted.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
possibly_leaking_reference_to_local_variable	Address of local variable is assigned to longer-living object.

AutosarC++17_03-M8.0.1

An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
allow_in_for_loop_init	Whether a multi-declaration is allowed in the for-init statement	False
allow_uninitialized_simple_type	Whether a multi-declaration of simple type is allowed when there is no initialization	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
multi_declaration	Multiple declarators in single declaration

AutosarC++17_03-M8.3.1

Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
redefinition_uses_different_default_argument	Default argument differs from the one in redefined method

AutosarC++17_03-M8.4.2

The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
ignore_casing	Whether to ignore casing for the check.	False
include_redefinitions	Whether declarations of redefining methods should be considered together with the redefined ones.	True
inspect_only_defined_routines	If False, re-declarations of undefined functions are compared to earlier declarations as well.	False
level		required
target		implementation

Possible Messages

Name	Message
parameter_name_mismatch	Different name used for parameter

AutosarC++17_03-M8.4.4

A function identifier shall either be used to call the function or it shall be preceded by &.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allowed	Qualified names of functions of which it is allowed to take the address implicitly, e.g. C++ I/O manipulators	['std::endl', 'std::flush', 'std::boolalpha', 'std::noboolalpha', 'std::showbase', 'std::noshowbase', 'std::showpoint', 'std::noshowpoint', 'std::showpos', 'std::noshowpos', 'std::skipws', 'std::noskipws', 'std::uppercase', 'std::nouppercase', 'std::unitbuf', 'std::nounitbuf', 'std::internal', 'std::left', 'std::right', 'std::dec', 'std::hex', 'std::oct', 'std::fixed', 'std::scientific', 'std::hexfloat', 'std::defaultfloat', 'std::ws', 'std::ends', 'std::resetiosflag', 'std::setiosflag', 'std::setbase', 'std::setfill', 'std::setprecision', 'std::setw', 'std::get_money', 'std::put_money', 'std::get_time', 'std::put_time', 'std::quoted']
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
implicit_routine_address	Taking address of function without &

AutosarC++17_03-M8.5.1

All variables shall have a defined value before they are used.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
init_functions	Names of functions to be inspected as well when called directly from constructor.	('Init', 'init')
inspect_directly_called_methods	Inspect all methods directly called from constructor.	False
level		required
only_member_initializer_list	Only inspect member initializer list and not the constructor body/methods.	False
perform_global_analysis	Whether global analysis (using iranalysis) should be done, or just a local analysis.	True
report_missing_base_constructors	Enables detection of constructors which rely on implicit base constructor calls.	False
report_missing_field_constructors	Enables detection of constructors which rely on implicit field constructor calls.	False
target		implementation

Possible Messages

Name	Message
implicit_field_init	Field is only implicitly initialized in constructor.
missing_base_class_init	Base class is not explicitly initialized in constructor.
missing_field_init	Field is not initialized in constructor.
pass_as_pointer_to_const_param	Passing uninitialized variable by pointer as function parameter with pointer-to-const type
possible_uninit	Use of possibly uninitialized variable
uninit	Use of uninitialized variable

AutosarC++17_03-M8.5.2

Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_any_zero_initialization	Whether {0} is allowed in nested aggregates as well.	False
allow_toplevel_zero_initialization	Whether {0} is allowed as top-level initialization for the complete aggregate	True
allowed_classes_without_braces	For the listed class types, don't require braces. This is useful for classes that transparently wrap a single contained value. For example, with this option "std::array<int, 3> a = {1,2,3};" is OK even though there could be two sets of braces (one for "std::array", the other and for its inner "int[3]" array).	['std::array']
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
incomplete_aggregate_initialization	Provide explicit initializer for each part of initialized entity
missing_aggregate_initialization_braces	Use braces to indicate structure of initialized entity

AutosarC++17_03-M9.3.1

const member functions shall not return non-const pointers or references to class-data.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
include_copies_of_pointer_fields	Whether returning a pointer field should be reported as well.	True
inspect_only_const_methods	Whether all methods or only const methods should be checked.	True
level		required
only_report_references	Whether pointer and reference to field should be reported, or just references.	False
smart_pointer_names	Names that are considered smart pointers.	['unique_ptr', 'shared_ptr', 'weak_ptr']
target		implementation

Possible Messages

Name	Message
returning_nonconst_member_reference	Returning non-const reference/pointer to class data.

AutosarC++17_03-M9.3.3

If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_candidates_for_const	If False, avoid reporting methods that can be made const.	True
report_candidates_for_static	If False, avoid reporting methods that can be made static.	True
target		implementation
test_operators_for_static	If True, check whether a method can be made static is also applied to operator methods	False

Possible Messages

Name	Message
method_can_be_const	Method can be declared const.
method_can_be_static	Method can be declared static.

AutosarC++17_03-M9.6.1

When the absolute positioning of bits representing a bit-field is required, then the behaviour and packing of bit-fields shall be documented.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitfield	Usage of bit-fields shall be documented

AutosarC++17_03-M10.1.1

Classes should not be derived from virtual bases.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
virtual_inheritance	Classes should not be derived from virtual bases.

AutosarC++17_03-M10.1.2

A base class shall only be declared virtual if it is used in a diamond hierarchy.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
virtual_inheritance_outside_diamond	A base class shall only be declared virtual if it is used in a diamond hierarchy.

AutosarC++17_03-M10.1.3

An accessible base class shall not be both virtual and non-virtual in the same hierarchy.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
base_class_being_virtual_and_nonvirtual	Has base class which is both virtual and non-virtual.

AutosarC++17_03-M10.2.1

All accessible entity names within a multiple inheritance hierarchy should be unique.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
ambiguous_member	All accessible entity names within a multiple inheritance hierarchy should be unique
use_of_ambiguous_name	{ } is ambiguous

AutosarC++17_03-M10.3.3

A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pure_redefinition	Pure redefinition of non-pure virtual function.

AutosarC++17_03-M11.0.1

Member data in non-POD class types shall be private.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_protected_members	If True, protected members are tolerated.	False
allowed	Specifies allowed fields as pairs [class name pattern, field name pattern]. Example: [re.compile('.*'), re.compile('x')] to allow x in all classes.	[]
enforcement		automated
ignore_const_members	If True, non-private const members are tolerated.	False
ignore_pod	Whether fields in POD classes should be reported.	True
ignore_structs	Whether fields in structs should be reported.	False
ignore_templates	Whether fields in generic templates should be reported.	True
level		required
target		implementation

Possible Messages

Name	Message
protected_field	Member data in non-POD class types shall be private.
public_field	Member data in non-POD class types shall be private.

AutosarC++17_03-M12.1.1

An object's dynamic type shall not be used from the body of its constructor or destructor.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
constructor_using_dynamic_cast	Dynamic cast used in constructor/destructor.
constructor_using_typeid	Typeid on polymorphic class used in constructor/destructor.
constructor_using_virtual_call	Virtual call used in constructor/destructor.

AutosarC++17_03-M14.5.2

A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_copy_constructor_for_template	Class has template constructor but no copy constructor

AutosarC++17_03-M14.5.3

A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_copy_asgn_for_template	Class has template assignment operator but no copy assignment operator

AutosarC++17_03-M14.6.1

In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unqualified_dependent_member_access	Use qualifiers or this-> to select name that may be found in that dependent base

AutosarC++17_03-M14.7.3

All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_in_type_declaration_file	Also allow specializations in the same file of a user-defined type for which the specialization is declared.	False
enforcement		automated
level		required
relax_if_template_only_declared	Allow specializations that are not declared in the header file if the template itself is only declared but not defined in the header.	False
target		implementation

Possible Messages

Name	Message
template_specialization_in_different_file	Specialization not declared in same file as primary template

AutosarC++17_03-M14.8.1

Overloaded function templates shall not be explicitly specialized.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
specialization_of_overloaded_template	Overloaded function templates shall not be explicitly specialized

AutosarC++17_03-M15.0.3

Control shall not be transferred into a try or catch block using a goto or a switch statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
goto_into_try	Goto jumps into try or catch block
switch_into_try	Switch statement jumps into try or catch block

AutosarC++17_03-M15.1.1

The assignment-expression of a throw statement shall not itself cause an exception to be thrown.

Input: IR
Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throw_expression_raises_exception	Expression of throw may itself raise an exception

AutosarC++17_03-M15.1.2

NULL shall not be thrown explicitly.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throwing_null	NULL shall not be thrown explicitly

AutosarC++17_03-M15.1.3

An empty throw (throw;) shall only be used in the compound-statement of a catch handler.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
rethrow_outsideCatch	Rethrow outside any catch block

AutosarC++17_03-M15.3.1

Exceptions shall be raised only after start-up and before termination of the program.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_only_one_exception_per_function	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
report_only_uncaught	Whether the check shall report all throws or just those not caught.	False
target		implementation

Possible Messages

Name	Message
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionRaisedInInitialization	Exception raised in initialization or finalization

AutosarC++17_03-M15.3.3

Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
handlerUsesField	Handler of a function-try-block shall not reference non-static members from this class or its bases

AutosarC++17_03-M15.3.4

Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
inspect_at_exit_handlers	Whether to also inspect at_exit() handlers-functions.	True
inspect_thread_main	Whether to also inspect thread main functions.	True
level		required
report_only_one_exception_per_function	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
target		implementation

Possible Messages

Name	Message
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionEscapingMain	Uncaught exception escaping from main or additional entry point

AutosarC++17_03-M15.3.6

Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
wrongCatchOrder	Catch handlers in wrong order.

AutosarC++17_03-M15.3.7

Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
catchAllNotLast	Catch-all shall occur as last handler.

AutosarC++17_03-M16.0.1

#include directives in a file shall only be preceded by other preprocessor directives or comments.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_extern_blocks	Whether to allow includes in `extern` blocks and not report a violation.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
include_after_code	#include should only be preceded by comments or preprocessor directives

AutosarC++17_03-M16.0.2

Macros shall only be #define'd or #undef'd in the global namespace.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
local_macro	#define or #undef not in global namespace

AutosarC++17_03-M16.0.5

Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
directives	Preprocessing directives to check for in macro arguments.	(`#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, `#endif`, `#pragma`, `#warning`, `#error`, `#line`, `#include`, `#include_next`, `#ident`, `#region`, `#endregion`, `#asm`, `#endasm`, `#define`, `#undef`)
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pp_directive_as_macro_arg	Preprocessing directive used in macro argument.

AutosarC++17_03-M16.0.6

In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unsafe_macro_param	Macro parameter not enclosed in parentheses

AutosarC++17_03-M16.0.7

Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
undefined_macro_used	Zero used for undefined preprocessing identifier

AutosarC++17_03-M16.0.8

If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
malformed_pp_directive	Malformed or unknown preprocessing directive

AutosarC++17_03-M16.1.1

The defined preprocessor operator shall only be used in one of the two standard forms.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
nonstandard_defined	Non-standard use of defined operator

AutosarC++17_03-M16.1.2

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_endif	The #endif for this directive is missing
missing_if	The #if for this directive is missing

AutosarC++17_03-M16.2.3

Include guards shall be provided.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
macro_name_restrictions	Python iterable of functions with parameters (file, define, macro) to perform additional checks on the macro used as include guard in the given file. Should return None if macro is accepted, else a message to print for the violation. The parameters are: PIR File node for the header, token for the macro #define or #pragma once, and the name of the macro or the string 'pragma once'.	None
target		implementation

Possible Messages

Name	Message
include_guard_missing	Missing include guard.
include_guard_partial	Include guard does not cover complete file.
include_guard_with_wrong_define	Bad include guard: #define uses wrong macro, should be {}.
include_guard_without_define	Incomplete include guard: missing #define {}.

AutosarC++17_03-M16.3.1

There shall be at most one occurrence of the # or ## operators in a single macro definition.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_only_concatenated_stringized	If True, only macros where a parameter is stringized and the result used for token concatenation are reported.	False
target		implementation

Possible Messages

Name	Message
concatenated_stringized	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator
more_than_one_macro_hash	Only one # or ## allowed in macro definition

AutosarC++17_03-M16.3.2

The # and ## operators should not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
hash_in_macro	The # and ## operators should not be used.

AutosarC++17_03-M17.0.2

The names of standard library macros and objects shall not be reused.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_locals	Whether parameters and local variables should also be checked	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
reused_macro_object_libname	The names of standard library macros and objects shall not be reused.

AutosarC++17_03-M17.0.3

The names of standard library functions shall not be overridden.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_function_declarations	Whether nondefining function declarations with library names are allowed	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
reused_routine_libname	The names of standard library functions shall not be overridden.

AutosarC++17_03-M17.0.5

The setjmp macro and the longjmp function shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	setjmp
symbols	Names of symbols which are forbidden.	['setjmp', 'longjmp']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++17_03-M18.0.3

The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	stdlib
symbols	Names of symbols which are forbidden.	['abort', 'exit', 'getenv', 'system']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib{.h} -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++17_03-M18.0.4

The time handling functions of library <ctime> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	time
symbols	Names of symbols which are forbidden.	['clock', 'difftime', 'mktime', 'time', 'asctime', 'ctime', 'gmtime', 'localtime', 'strftime']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib{.h} -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++17_03-M18.0.5

The unbounded functions of library <cstring> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	string
symbols	Names of symbols which are forbidden.	['strcpy', 'strcmp', 'strcat', 'strchr', 'strspn', 'strcspn', 'strpbrk', 'strrchr', 'strstr', 'strtok', 'strlen']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++17_03-M18.2.1

The macro offsetof shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	stddef
symbols	Names of symbols which are forbidden.	['offsetof']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++17_03-M18.7.1

The signal handling facilities of <csignal> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header (used for symbols like NULL and size_t which are defined in multiple headers)	['NULL', 'size_t']
enforcement		automated
header	Name of the header file which should not be used.	signal
level		required
target		implementation

Possible Messages

Name	Message
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.

AutosarC++17_03-M19.3.1

The error indicator errno shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	['errno', 'stdlib', 'stddef']
symbols	Names of symbols which are forbidden.	['errno']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <cerrno>.

AutosarC++17_03-M27.0.1

The stream input/output library <cstdio> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header (used for symbols like NULL and size_t which are defined in multiple headers)	['NULL', 'size_t']
enforcement		automated
header	Name of the header file which should not be used.	stdio
level		required
target		implementation

Possible Messages

Name	Message
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.

Rules in Group AutosarC++17_10

AutosarC++17_10-A0.1.1

A project shall not contain instances of non-volatile variables being given values that are never subsequently used.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_dead_initializations	Whether initialisations may be reported as dead.	True
target		implementation

Possible Messages

Name	Message
conditional_unused_def	Result of assignment is not used along some path(s)
init_used_in_other_isr	Initialization is only used in some interrupt handler
unused_def	Result of assignment is not used
unused_init	Unused initialization
used_in_other_isr	Result of assignment is only used in some interrupt handler

AutosarC++17_10-A0.1.2

The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed_functions	Calls to these functions are ignored.	frozenset(['strncat', 'memmove', 'memset', 'strcat', 'strcpy', 'memcpy', 'strncpy'])
enforcement		automated
inspect_template_instances	Whether calls in template instances should be reported.	False
level		required
target		implementation

Possible Messages

Name	Message
discarded_return_with_entity	Return value of function discarded.
discarded_return_without_entity	Return value of function discarded.

AutosarC++17_10-A0.1.3

Every defined function shall be called at least once.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_copy_assignment_operators	Whether uncalled copy assignment operators are allowed. Other rules like the "rule of three" might require copy assignment operators as part of good class design, even if there are currently no callers of the copy assignment operator.	False
ignore_copy_constructors	Whether uncalled copy constructors are allowed. Other rules like the "rule of three" might require copy constructors as part of good class design, even if there are currently no callers of the copy constructor.	False
level		required
only_check_internal_or_private_methods	Whether only function with internal linkage or private functions should be checked.	True
only_check_static_or_private_methods	Whether only static functions or private functions should be checked.	False
only_check_unit_locals	Whether only global static functions should be checked.	False
target		implementation

Possible Messages

Name	Message
defined_function_not_called	Defined function is not called

AutosarC++17_10-A0.1.4

There shall be no unused named parameters in non-virtual functions.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
inspect_nonvirtual_functions	Whether parameters of non-virtual functions should be checked.	True
inspect_virtual_functions	Whether virtual functions should be checked.	False
level		required
report_unnamed_parameters	Whether unnamed unused parameters should be reported as well.	False
target		implementation

Possible Messages

Name	Message
unused_parameter	Unused named parameter of non-virtual function

AutosarC++17_10-A0.1.5

There shall be no unused named parameters in the set of parameters for a virtual function and all the functions that override it.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
inspect_nonvirtual_functions	Whether non-virtual functions should be checked.	False
inspect_virtual_functions	Whether parameters of virtual functions should be checked. Violations will be reported in the base class when none of the derived classes make use of the parameter.	True
level		required
report_unnamed_parameters	Whether unnamed unused parameters should be reported as well.	False
target		implementation

Possible Messages

Name	Message
unused_parameter	Unused named parameter of virtual function

AutosarC++17_10-A0.4.2

Type long double shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
forbidden_types	The ir.Logical Types not to be used e.g. `C_Long_Double_Type`.	['C_Long_Double_Type']
level		required
target		implementation

Possible Messages

Name	Message
type_used	The type shall not be used.

AutosarC++17_10-A1.1.1

All code shall conform to ISO/IEC 14882:2014 - Programming Language C++ and shall not use deprecated features.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	The set of messages regarding syntax and constraint violations.	set([2464, 2465, 1444, 2567, 2381, 2221, 1909, 1215])
reported_severities	List of severities to display.	('error', 'warning')
show_include_path	If True, violations regarding nested includes print the #include path.	False
standards	List of allowed frontend command-line options to select language standard.	['--strict', '-A']
target		implementation
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

Possible Messages

Name	Message
access_control	Number of access control declarations is higher than translation limit (which is {})
at_quick_exit_functions	Number of functions registered with at_quick_exit() is higher than translation limit (which is {})
atexit_functions	Number of functions registered with atexit() is higher than translation limit (which is {})
base_classes	Number of direct and indirect base classes is higher than translation limit (which is {})
call_arguments	Number of arguments is higher than translation limit (which is {})
catch_handlers	Number of catch handlers is higher than translation limit (which is {})
chars_in_line	Number of characters in logical line is higher than translation limit (which is {})
chars_in_string	Number of characters in string is higher than translation limit (which is {})
cpp_members	Number of members is higher than translation limit (which is {})
data_members	Number of data members is higher than translation limit (which is {})
direct_bases	Number of direct base classes is higher than translation limit (which is {})
enumerators	Number of enumerators is higher than translation limit (which is {})
expr_in_core_constant	Number of full expressions evaluated within a core constant expression is higher than translation limit (which is {})
external_identifiers	Number of external identifiers in this unit is higher than translation limit (which is {})
final_virtual_functions	Number of final overriding virtual functions is higher than translation limit (which is {})
friends	Number of friend declarations is higher than translation limit (which is {})
function_parameters	Number of parameters is higher than translation limit (which is {})
macro_arguments	Number of macro arguments is higher than translation limit (which is {})
macro_parameters	Number of macro parameters is higher than translation limit (which is {})
macros	Number of macro definitions in this unit is higher than translation limit (which is {})
member_initializers	Number of member initializers is higher than translation limit (which is {})
nested_blocks	Number of nested blocks is higher than translation limit (which is {})
nested_composites	Nesting level of structs and unions is higher than translation limit (which is {})
nested_declarators	Number of pointer/array/function declarators is higher than translation limit (which is {})

nested_externals	Number of nested external specifications is higher than translation limit (which is {})
nested_includes	Nesting level of #includes is higher than translation limit (which is {})
nested_preprocessor_ifs	Nesting level of #if is higher than translation limit (which is {})
nested_template_instantiations	Number of nested template instantiations is higher than translation limit (which is {})
object_size	Number of bytes in object is higher than translation limit (which is {})
parenthesized_declarators	Nesting level of parenthesized declarators is higher than translation limit (which is {})
parenthesized_expressions	Nesting level of parenthesized subexpressions is higher than translation limit (which is {})
placeholders	Number of placeholders is higher than translation limit (which is {})
recursive_constexpr	Number of recursive constexpr function invocations is higher than translation limit (which is {})
scope_qualifications	Number of scope qualifications is higher than translation limit (which is {})
significant_chars_external	Number of significant initial characters for external identifier is higher than translation limit (which is {})
significant_chars_internal	Number of significant initial characters for internal identifier is higher than translation limit (which is {})
static_members	Number of static members is higher than translation limit (which is {})
std_compile	You have to compile with option {}
switch_cases	Number of cases is higher than translation limit (which is {})
template_params	Number of template parameters is higher than translation limit (which is {})
throw_spec	Number of throw specifications is higher than translation limit (which is {})
variables_in_block	Number of variables in block is higher than translation limit (which is {})
virtual_bases	Number of virtual base classes is higher than translation limit (which is {})

AutosarC++17_10-A2.2.1

Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_comments	Whether to allow characters not in the basic source character set in comments.	False
allow_in_unicode_strings	Whether to allow characters not in the basic source character set in unicode string literals.	False
allow_in_wide_strings	Whether to allow characters not in the basic source character set in wide string literals.	True
basic_characters_list	The basic character list as per rule.	\ \\ \ \\\ abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_\\ {}\\[]#\\()\\<\\>%\\:\\;.\\?*\\+\\-\\^\\&\\ \\~\\!=\\,\\\"\\`\\\\@\\
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
forbidden_characters	Forbidden characters used.

AutosarC++17_10-A2.5.1

Trigraphs shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_comments	Allow the use of trigraphs in comments	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
trigraph_use	Use of trigraph.

AutosarC++17_10-A2.6.1

Digraphs should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
digraph_use	Digraph used.

AutosarC++17_10-A2.8.1

The character \ shall not occur as a last character of a C++ comment.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
line_splicing_in_cpp_comment	Line-splicing shall not be used in // comments.

AutosarC++17_10-A2.8.2

Sections of code shall not be "commented out".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
files_to_check	Files to be checked, e.g. Primary_File or File.	File
level		required
normal_text	Tokens that indicate non-code comments. This is the first step of checking a comment. If any of these words exist in the comment text, it is regarded as not containing code.	(' and ', ' or ', ' but ', ' now ', ' to ', ' is ', ' are ', ' only ', ' be ', ' has ', ' the ', ' with ', ' because ', ' when ', ' oder ', ' und ', ' // ', '###', 'AXIVION', '++++', '----', '====')
normal_text_regex	Regular expression to match normal non-code text. This is the second step of checking a comment. If this regex matches the comment text, the comment is regarded as not containing code.	\b[\\w\\d_]+\\s+[\\w\\d_]+\\s+[\\w\\d_]+\\b
suspicious	Tokens that indicate code in comments. This is the final step of checking a comment. If the first two steps did not exclude the comment from further inspection, the comment text is searched for any of these suspicious tokens. The comment is regarded as containing code, if any suspicious token is found.	(' = ', ' == ', ' >= ', ' <= ', '[', ']', ':', '->', '->*', ' ::*', 'if', 'while', 'for', 'if (', 'while (', 'for (', '#pragma', '#else', '#endif', '#if', '#include', '++', '--')
target		implementation

Possible Messages

Name	Message
code_in_c_comment	Sections of code shall not be commented out using C-style comments.
code_in_cpp_comment	Sections of code should not be commented out using C++ comments.

AutosarC++17_10-A2.8.3

All declarations of "user-defined" types, static and non-static data members, functions and methods shall be preceded by documentation using "///" comments and "@tag" tags.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_inherited	If True, a definition does not need documentation, if a corresponding declaration is documented.	False
allow_missing_documentation_on_private	If True, a class-member definition does not need documentation, if it is `private`.	False
allow_missing_documentation_on_protected	If True, a class-member definition does not need documentation, if it is `protected`.	False
doxygen_start	Start of a valid Doxygen comment.	{'/**', '///'}
enforcement		automated
ignore_defaulted	If True, defaulted function declarations are not checked for comments.	False
ignore_deleted	If True, deleted function declarations are not checked for comments.	False
ignore_redefinitions	If True, method redefinitions are not checked as they can 'inherit' the comment from the redefined method.	False
ignore_tool_comments	An optional compiled regular expression. Comments where this regex finds a matching substring are ignored in the search for a doxygen comment (e.g. control-comments of other tools).	None
level		required
node_types	IR node types to check for preceding Doxygen comment.	{'Routine_Definition', 'Routine_Declaration', 'Named_Type_Definition', 'Field_Definition'}
target		implementation

Possible Messages

Name	Message
missing_doxygen_comment_before_def	No Doxygen comment before declaration.

AutosarC++17_10-A2.8.4

C-style comments shall not be used.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
exception	Start of a valid comment, even if it has an invalid prefix.	None
invalid	Start of an invalid comment.	/*
level		required
target		implementation

Possible Messages

Name	Message
cpp_comment_style	Use of invalid comment style in C++ unit.

AutosarC++17_10-A2.9.1

A header file name shall be identical to a type name declared in it if it declares a type.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
file_type	Type of header file to examine.	User_Include_File
junk_characters	Characters that should not be considered in the filename to type similarity.	_-::
level		required
named_node_types	Physical IR node types to inspect for logical names to compare against the file name	Named_Type_Interface
similarity_threshold	Ratio of how similar the names need to be.	0.9
target		implementation

Possible Messages

Name	Message
source_file_name_not_type	The header should be named as a type it declares.

AutosarC++17_10-A2.11.1

Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_classes	Whether identifiers in local classes can hide surrounding ones	True
allow_in_lambdas	Whether identifiers in lambdas can hide surrounding ones	True
allow_in_local_classes	Whether identifiers in local classes can hide surrounding ones	False
allow_in_namespaces	Whether identifiers in namespaces can hide surrounding ones	True
allow_parameter_in_function_declaration	Whether identifiers of parameters in function declarations may hide surrounding ones, given the function declaration uses a different identifier for the same parameter that does not hide a surrounding one.	False
check_inheritance_hiding	Whether to check for hiding by inheritance.	False
check_nested_classes	Whether symbols from inner classes should be reported as hiding symbols of outer classes.	False
distinguish_name_spaces	Whether C name spaces should be distinguished.	False
enforcement		automated
hiding_entities	LIR node types to consider symbols that hide other symbols.	{'Variable', 'Typedef_Type', 'Field', 'Parameter'}
level		required
maxlen	Number of significant characters {or None}	None
target		implementation
tolerate_constructor_parameters_hiding_fields	Allow constructor parameters to hide fields	False
tolerate_function_prototype_argument_hiding	Whether to tolerate arguments of a function prototype, that itself is a function argument, that have the same identifier as a argument of the outer function. If 'True' is given, the rule will still report cases, where argument identifiers match up and the function prototype refers to arguments of the outer function (e.g. by `decltype()`).	False
tolerate_macro_local_identifier	Whether an identifier being local to a macro is allowed to hide identifiers from outside the macro	False
unchecked_types	Deprecated legacy option, kept to avoid breaking old configurations.	{'Routine_Call', 'Stack_Object_Definition'}

Possible Messages

Name	Message
hiding	{ } hides { }

AutosarC++17_10-A2.11.2

A "using" name shall be a unique identifier within a namespace.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
reused_using	Using name reused.

AutosarC++17_10-A2.11.3

A "user-defined" type name shall be a unique identifier within a namespace.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
check_types_only_with_external_linkage	Whether the types should only be checked when they have external linkage	False
enforcement		automated
level		required
target		implementation
tolerate_macros	Whether #define and #undef using the typedef's name is allowed	False
tolerate_typedef	Whether a typedef to the tag may have the same name.	False
tolerate_typedef_entity	Whether the entity forming the typedef's underlying type can have the same name.	False

Possible Messages

Name	Message
composite_defined_multiple_times	Composite type has multiple definitions.
enum_defined_multiple_times	Enumeration type has multiple definitions.
reused_tag	Tag name reused.
reused_typedef	Typedef name reused.

AutosarC++17_10-A2.11.4

The identifier name of a non-member object or function with static storage duration should not be reused.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_static_no_linkage	Whether to allow same identifiers for static objects without linkage (e.g. function local static objects)	False
enforcement		automated
func_filter	Restricts which functions are considered.	None
level		required
target		implementation
use_unqualified_names	If set to True, ns1::Type is considered equal to ns2::Type.	False
var_filter	Restricts which variables are considered.	None

Possible Messages

Name	Message
reused_global_function_name	Name of object or function with static storage duration reused.
reused_global_variable_name	Name of object or function with static storage duration reused.

AutosarC++17_10-A2.11.5

The identifier name of a non-member object or function with static storage duration should not be reused.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_static_no_linkage	Whether to allow same identifiers for static objects without linkage (e.g. function local static objects)	True
enforcement		automated
func_filter	Restricts which functions are considered.	None
level		required
target		implementation
use_unqualified_names	If set to True, ns1::Type is considered equal to ns2::Type.	True
var_filter	Restricts which variables are considered.	None

Possible Messages

Name	Message
reused_global_function_name	Name of object or function with static storage duration reused.
reused_global_variable_name	Name of object or function with static storage duration reused.

AutosarC++17_10-A2.14.1

Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed	List of allowed characters after backslash.	"\"?\\abfnrtv
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
nonstandard_escape_sequence	Use of non-standard escape sequence.

AutosarC++17_10-A2.14.2

Narrow and wide string literals shall not be concatenated.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
cpp11_mode	Use rules as defined in the C++11 standard.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
mixed_string_concatenation	Concatenation of mixed string encodings
narrow_wide_concat	Concatenation of narrow and wide string literal

AutosarC++17_10-A2.14.3

Type wchar_t shall not be used.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
forbidden_types	The ir.Logical Types not to be used e.g. `C_Long_Double_Type`.	['CPP_WChar_Type']
level		required
target		implementation

Possible Messages

Name	Message
type_used	The type shall not be used.

AutosarC++17_10-A3.1.1

It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
accept_const_fields	Whether const-qualified static fields in header files should be tolerated.	True
accept_const_variables	Whether global const variables in header files should be tolerated.	True
enforcement		automated
find_non_external_function_declaration	Whether function declarations with non-external linkage should be reported.	True
level		required
target		implementation

Possible Messages

Name	Message
function_definition_in_header	Definition in header file.
static_field_def_in_header	Definition in header file.
variable_definition_in_header	Definition in header file.

AutosarC++17_10-A3.1.2

Header files, that are defined locally in the project, shall have a file name' extension of one of: ".h", ".hpp" or "..hxx".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed_extensions	Which file extensions are allowed.	set[{'h', '.hxx', 'hpp'}]
enforcement		automated
file_type	The files to check the extensions of.	User_Include_File
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
illegal_file_extension	Use only certain extensions for file names.

AutosarC++17_10-A3.1.3

Implementation files, that are defined locally in the project, should have a file name extension of ".cpp".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed_extensions	Which file extensions are allowed.	set(['.cpp'])
enforcement		automated
file_type	The files to check the extensions of.	Primary_File
level		advisory
target		architecture/design/implementation

Possible Messages

Name	Message
illegal_file_extension	Use only certain extensions for file names.

AutosarC++17_10-A3.1.4

When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_extern	Whether to consider only extern declared arrays	True
report_definitions	Whether definitions of array variables should also be reported	True
target		implementation

Possible Messages

Name	Message
unbounded_array	Array declaration with unknown size.

AutosarC++17_10-A3.3.1

Objects or functions with external linkage shall be declared in a header file.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_function_declaration_in_header	Object or function with external linkage shall be declared in a header file
missing_variable_declaration_in_header	Object or function with external linkage shall be declared in a header file

AutosarC++17_10-A3.3.2

Non-POD type objects with static storage duration shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
static_nonpod_variable	Do not use non-POD type objects with static storage duration.

AutosarC++17_10-A3.9.1

Typedefs that indicate size and signedness should be used in place of the basic numerical types.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_floating_point	Whether floating point types without should not be reported.	True
enforcement		automated
ignore_inherited	If true, missing typedefs in inherited methods are not reported.	False
level		required
target		implementation
treat_plain_char_as_numerical	Whether the char type without any signedness should be reported.	True

Possible Messages

Name	Message
missing_integer_typedef	Use of base type outside typedef.
wrong_integer_typedef	Use of badly named typedef for base type.

AutosarC++17_10-A4.5.1

Expressions with type enum shall not be used as operands to built-in operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_use_in_operator_calls	Whether enum arguments in calls to forbidden overloaded operators should be reported.	True
target		implementation

Possible Messages

Name	Message
enum_operand_outside_comparison	Use of enum operand in arithmetic or similar context

AutosarC++17_10-A4.7.1

An integer expression shall not lead to data loss.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
abstract_interpretation_overflow	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cast_overflow	Cast on result of arithmetic computation may cause overflow
cast_truncate	Cast may truncate value
cast_underflow	Cast on result of arithmetic computation may cause underflow
overflow	Arithmetic computation may cause overflow
underflow	Arithmetic computation may cause underflow

AutosarC++17_10-A4.10.1

Only nullptr literal shall be used as the null-pointer-constant.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
require_nullptr	Require C++11 nullptr instead of macro NULL or literal zero {0}.	True
target		architecture/design/implementation

Possible Messages

Name	Message
null_constant	Only nullptr literal shall be used as the null-pointer-constant.
zero_as_null	Use of literal zero {0} as null-pointer-constant, use {} instead

AutosarC++17_10-A5.0.1

The value of an expression shall be the same under any order of evaluation that the standard permits.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_exceptions	Ignore possible exceptions in the computation of side effects. (this option only has an effect if report_calls is enabled)	False
level		required
report_calls	If True, unsequenced function calls are reported.	True
target		implementation

Possible Messages

Name	Message
unsequenced_call_volatile	Function calls unsequenced with volatile access
unsequenced_calls	Unsequenced function calls
unsequenced_read_write	Unsequenced read and write accesses
unsequenced_volatiles	Unsequenced volatile accesses
unsequenced_writes	Unsequenced writes

AutosarC++17_10-A5.0.2

The condition of an if-statement and the condition of an iteration-statement shall have type bool.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_in_conditions	If True, only logical operators inside conditions are checked.	True
report_do_while_false	Whether do ... while[0] should be reported.	True
report_while_true	Whether while[1] ... should be reported.	True
target		implementation

Possible Messages

Name	Message
nonbool_if_condition	Condition must have type bool
nonbool_logical_operator_operand	Sub-condition must have type bool
nonbool_loop_condition	Condition must have type bool

AutosarC++17_10-A5.0.3

The declaration of objects should contain no more than two levels of pointer indirection.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
max_levels	Maximum number of allowed pointer-indirection levels.	2
target		implementation

Possible Messages

Name	Message
too_many_pointer_levels	More than {} level{} of pointer indirection

AutosarC++17_10-A5.1.1

Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_nonconst_variable_initialization	If set to `True`, allow the initialization of a non-const variable with a string literal in addition to the initialization of a const variable. A reassignment of such a variable is still a violation.	False
allowed	Literal values that are ok.	{0.0, 1.0, 2.0}
allowed_contexts	Optional set of PIR classes or functions {[node] -> bool} for allowed contexts, e.g. Case_Label.	set[<function allow_in_type_initialization at 0x7dfa84459b0>]
allowed_logging_contexts	List of fully qualified function types that are considered logging contexts [using operator<>]. If this is non-empty, `std::throw()` is considered a valid logging context as well.	['std::basic_ostream']
allowed_string_contexts	Optional set of PIR classes or functions {[node] -> bool} for allowed contexts.	set[<function allow_in_type_initialization at 0x7dfa84459b0>]
allowed_strings	Literal values that are ok.	['', ' ']
enforcement		partially automated
exceptions	Optional predicate to filter out cases that should be allowed as an exception. The predicate takes a literal node and returns True if the given node is such an exception.	None
exclude_pp_literals	If True, literals in conditions of #if are ignored.	True
exclude_single_uses	If True, report only literals used more than once.	False
level		required
target		implementation

Possible Messages

Name	Message
magic_number	Use of magic literal.
magic_number_without_token	Use of magic literal.
magic_string	Use of magic string literal.
possible_magic_number	Potential use of magic literal.

AutosarC++17_10-A5.1.2

Variables shall not be implicitly captured in a lambda expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
implicit_lambda_capture	Variables shall not be implicitly captured in a lambda expression.

AutosarC++17_10-A5.1.5

If a lambda expression is used in the same scope in which it has been defined, the lambda should capture objects by reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
level		advisory
target		implementation

Possible Messages

Name	Message
local_lambda	An only locally used lambda should capture by reference.

AutosarC++17_10-A5.1.7

The underlying type of lambda expression shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
underlying_lambda_type	The underlying type of lambda expression shall not be used.

AutosarC++17_10-A5.1.8

Lambda expressions should not be defined inside another lambda expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
nested_lambda	Lambda expressions should not be defined inside another lambda expression

AutosarC++17_10-A5.2.1

dynamic_cast should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
dynamic_cast	dynamic_cast should not be used.

AutosarC++17_10-A5.2.2

Traditional C-style casts shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_void_cast	If True, [void] is always allowed, else only for return value of calls.	False
allow_void_cast_on_call	If True, [void] is allowed, for return value of calls.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
c_cast	Use of C-style cast in C++ unit.

AutosarC++17_10-A5.2.3

A cast shall not remove any const or volatile qualification from the type of a pointer or reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
check_const	Whether casts affecting const-qualifiers should be reported	True
check_volatile	Whether casts affecting volatile-qualifiers should be reported	True
enforcement		automated
level		required
only_top_level	Whether the check only applies to the top-level pointee or recursively to deeper pointer levels as well	False
target		implementation

Possible Messages

Name	Message
cast_removes_const	Cast removes const qualification
cast_removes_volatile	Cast removes volatile qualification
implicit_cast_removes_const	Implicit cast removes const qualification
implicit_cast_removes_volatile	Implicit cast removes volatile qualification

AutosarC++17_10-A5.2.4

reinterpret_cast shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
reinterpret_cast	reinterpret_cast should not be used.

AutosarC++17_10-A5.3.1

Evaluation of the operand to the typeid operator shall not contain side effects.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
level		required
target		implementation

Possible Messages

Name	Message
side_effect_in_typeid	Operand of "typeid" shall not contain side effects

AutosarC++17_10-A5.5.1

The right hand operand of the integer division or remainder operators shall not be equal to zero.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
abstract_interpretation_div_by_zero	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
division_by_zero	Division by zero
modulo_by_zero	Modulo by zero
possible_division_by_zero	Possible division by zero
possible_modulo_by_zero	Possible modulo by zero

AutosarC++17_10-A5.10.1

A pointer to member virtual function shall only be tested for equality with null-pointer-constant.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
member_virtual_function_comparison	Comparison of a member virtual function with non-nullptr.

AutosarC++17_10-A5.16.1

The ternary conditional operator shall not be used as a sub-expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_nested_conditional_operator	Use of nested conditional operator.

AutosarC++17_10-A6.4.1

Every switch statement shall have at least one case-clause.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
minimum_switch_cases	The number of cases a switch statement should at least have.	2
target		implementation

Possible Messages

Name	Message
empty_switch	Switch has no "case" clause.
minimum_switch_cases	Switch has too little "case" clauses.

AutosarC++17_10-A6.5.1

A for-loop that loops through all elements of the container and does not use its loop-counter shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
exclude_simple_for_loops	If set to `True` this rule will not report unused loop counters in the simple cases where the C for-loop only references loop-counters in its condition but no other variables.	True
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
could_be_range_based	For loop could be a range-based for loop.
unused_loop_counter	For loop does not use its loop counter.

AutosarC++17_10-A6.5.2

A for loop shall contain a single loop-counter which shall not have floating type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
loop_float_counter	Loop-counter of for loop shall not have floating type
loop_missing_counter	For loop has no loop-counter
loop_multiple_counters	For loop shall have only a single loop-counter

AutosarC++17_10-A6.5.3

Do statements should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
do_while_loop	Do statements should not be used.

AutosarC++17_10-A6.6.1

The goto statement shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
goto	Do not use goto.

AutosarC++17_10-A7.1.1

Constexpr or const specifiers shall be used for immutable data declaration.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_pointer_variables	Whether variables of pointer type should be ignored.	False
level		required
only_check_unit_locals	Whether only local variables and global static variables should be checked.	False
only_immutable_data	Whether only declarations with immutable data should be checked.	True
target		implementation

Possible Messages

Name	Message
variable_missing_const	An immutable variable shall be const/constexpr qualified.

AutosarC++17_10-A7.1.3

CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_nonpointer_typedefs	Whether to allow the qualifier on the rhs of a type, if it is a non-pointer typedef.	False
allowed_typedefs	Set of names for typedefs/usings that are allowed (e.g. int32_t).	set(['int32_t', 'uint_least64_t', 'intptr_t', 'uintmax_t', 'int_fast16_t', 'intmax_t', 'int_fast8_t', 'int64_t', 'size_t', 'int_fast64_t', 'time_t', 'uint8_t', 'lldiv_t', 'int_least8_t', 'div_t', 'uint_least16_t', 'clock_t', 'uint_least32_t', 'int_least64_t', 'int_least16_t', 'int_least32_t', 'uint_least8_t', 'uintptr_t', 'max_align_t', 'int8_t', 'fpos_t', 'ldiv_t', 'uint_fast32_t', 'uint_fast64_t', 'nullptr_t', 'int_fast32_t', 'uint_fast16_t', 'uint32_t', 'ptrdiff_t', 'int16_t', 'uint64_t', 'uint16_t', 'uint_fast8_t'])
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
lhs_cv_qualifier	CV qualifier on the lhs of a type.

AutosarC++17_10-A7.1.4

The register keyword shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
register_keyword	The register keyword shall not be used

AutosarC++17_10-A7.1.5

The auto specifier shall not be used apart from following cases: (1) to declare that a variable has the same type as return type of a function call, (2) to declare that a variable has the same type as initializer of non-fundamental type, (3) to declare parameters of a generic lambda expression, (4) to declare a function template using trailing return type syntax.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_for_loop_counter	Disables message if auto is used to declare a for-loop counter.	False
allow_generic_lambda_parameters	Allows auto as parameter type in a generic lambda	True
allow_nonfundamental_initializer	Allows auto to declare variables having a function call or initializer of non-fundamental type	True
allow_template_instance	Disables message if auto stands for a template instance.	False
allowed_contexts	Set of context predicates in which auto is allowed.	set[[])
allowed_types	Set of types for which auto is allowed, given as name pattern, function, or LIR class name.	set[[])
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cpp11_auto	Use of C++11 auto type specifier.

AutosarC++17_10-A7.1.6

The typedef specifier shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
typedefSpecifier	The typedef specifier shall not be used.

AutosarC++17_10-A7.1.7

Each expression statement and identifier declaration shall be placed on a separate line.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_for_loop_init	Whether a multi-declaration is allowed in the for-init statement	True
allow_uninitialized_simple_type	Whether a multi-declaration of simple type is allowed when there is no initialization	False
enforcement		automated
ignore_stmts	Statements to be ignored when counting statements.	['Statement_Sequence', 'C_For_Loop']
level		required
target		implementation

Possible Messages

Name	Message
multi_declaration	Multiple declarators in single declaration
multiple_statements_per_line	Multiple statements per line.

AutosarC++17_10-A7.2.1

An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
asgn_not_an_enumerator	Assignment of value which does not correspond to an enumerator
asgn_out_of_enum_range	Assignment of value outside the range of the enumerators
conversion_creating_bad_enum_value	Expression does not correspond to an enumerator in {}

AutosarC++17_10-A7.2.2

Enumeration underlying base type shall be explicitly defined.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unbased_enum	Enumeration underlying base type shall be explicitly defined.

AutosarC++17_10-A7.2.3

Enumerations shall be declared as scoped enum classes.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unscoped_enum	Enumerations shall be declared as scoped enum classes.

AutosarC++17_10-A7.2.4

In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
incomplete_enum_init	Do not initialize enumerators other than the first, or initialize all

AutosarC++17_10-A7.4.1

The asm declaration shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_assembler	Use of assembler.

AutosarC++17_10-A7.5.1

A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_const_reference	Whether only to report returns of parameters with reference to const.	True
target		implementation

Possible Messages

Name	Message
returning_reference_to_refparam	Returning reference/pointer to reference parameter.

AutosarC++17_10-A7.5.2

Functions shall not call themselves, either directly or indirectly.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_cycle	Cycle of recursive functions
recursive_function	Function is directly recursive

AutosarC++17_10-A8.2.1

When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_trailing_return	Use trailing return-type syntax for function templates.

AutosarC++17_10-A8.4.1

Functions shall not be defined using the ellipsis notation.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_declarations	Allow declarations of functions with variable number of arguments (e.g. for existing library functions); only report definitions.	True
ignore_inherited	Do not report functions inheriting a variable number of arguments.	False
level		required
target		implementation

Possible Messages

Name	Message
ellipsis_parameter	Function definitions shall not use ellipsis

AutosarC++17_10-A8.4.2

All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
lambda_return_missing_value	Return without value in non-void lambda expression
missing_return	Non-void function needs return with value at end.
missing_return_in_lambda	Non-void lambda expression needs return with value at end.
return_missing_value	Return without value in non-void function

AutosarC++17_10-A8.4.4

Multiple output values from a function should be returned as a struct or tuple.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_modifying_calls	Whether a function may call a function on a parameter that modifies it, without regarding it as a violation here.	True
allow_move_parameters	Whether a move parameter (a rvalue reference) may be written to without triggering a violation. Note: move-constructors and move-assignments are allowed in either case.	False
allow_this_modification	Whether a function may call a member-function on a parameter that modifies this, that is the internal parameter state ('allow_modifying_calls' must be False if this is set to False)	True
enforcement		automated
exclude_constructors	Whether not to report parameters on constructors	True
exclude_operators	Whether not to report parameters on operator functions. Note: if set to True, this also sets 'exclude_shift_operators=True'.	False
exclude_shift_operators	Whether not to report parameters on shift operator functions (operator<<(), operator<<=(), operator>>(), and operator>>=).	False
level		advisory
only_report_multiple_output_parameters	Whether to only report output-parameters, if a function has more than one or also returns a value.	True
record_field_modification_threshold	A percentage of how many record fields may be modified, before the record pointer is considered an output parameter. 0 : every field modification counts as an output parameter 100: the record pointer is only regarded an output parameter if all field are modified	0
target		design

Possible Messages

Name	Message
output_parameter	Use return value instead of output parameter.

AutosarC++17_10-A8.5.1

In an initialization list, the order of initialization shall be following: (1) virtual base classes in depth and left to right order of the inheritance graph, (2) direct base classes in left to right order of inheritance list, [3] non-static data members in the order they were declared in the class definition.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	If provided, only messages of these types are reported.	[1719]
reported_severities	List of severities to display.	('error', 'warning', 'remark')
target		implementation
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

AutosarC++17_10-A8.5.2

Braced-initialization {}, without equals sign, shall be used for variable initialization.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
old_style_variable_init	Braced-initialization {}, without equals sign, shall be used for variable initialization.

AutosarC++17_10-A8.5.3

A variable of type auto shall not be initialized using {} or ={} braced-initialization.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
braced_auto_variable_init	A variable of type auto shall not be initialized using {} or ={} braced-initialization.

AutosarC++17_10-A8.5.4

A constructor taking parameter of type std::initializer_list shall only be defined in classes that internally store a collection of objects.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
level		advisory
target		implementation

Possible Messages

Name	Message
constructor_init_list_container	The class should be a container if a constructor taking std::initializer_list is defined.

AutosarC++17_10-A9.5.1

Unions shall not be used.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_tagged_unions	Whether tagged unions (nested in a struct that has an enum discriminator) should be allowed	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
union	Unions shall not be used

AutosarC++17_10-A9.6.1

Bit-fields shall be either unsigned integral, or enumeration (with underlying type of unsigned integral type).

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
invalid_bitfield	Bit-fields shall be either unsigned integral, or enumeration (with underlying type of unsigned integral type).

AutosarC++17_10-A10.1.1

Class shall not be derived from more than one base class which is not an interface class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_pure_interfaces	Whether C++ interfaces are allowed, i.e. classes with only pure virtual members.	True
level		required
target		implementation

Possible Messages

Name	Message
multiple_inheritance	Use of multiple inheritance.

AutosarC++17_10-A10.2.1

Non-virtual member functions shall not be redefined in derived classes.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
redefinition_of_nonvirtual_function	Redefinition/hiding of non-virtual function.

AutosarC++17_10-A10.3.1

Virtual function declaration shall contain exactly one of the three specifiers: {1} virtual, {2} override, {3} final.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
disable_for_destructors	If set, destructors are not checked.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
invalid_combination	Use only one of {1} virtual, {2} override, {3} final.

AutosarC++17_10-A10.3.2

Each overriding virtual function shall be declared with the override or final specifier.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_final	If set to True, don't report overriding virtual functions declared with final.	True
enforcement		automated
ignore_destructors	If set to False, also report destructors. Note that not all compilers support this.	False
level		required
target		implementation

Possible Messages

Name	Message
missing_override	Override of functions is only permitted with keyword override/final.

AutosarC++17_10-A10.3.3

Virtual functions shall not be introduced in a final class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
virtual_method	Virtual functions shall not be introduced in a final class.
virtual_method_override	Virtual functions shall not be overridden without final in a final class.

AutosarC++17_10-A10.3.5

A user-defined assignment operator shall not be virtual.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
assignment_virtual	A user-defined assignment operator shall not be virtual.

AutosarC++17_10-A11.0.1

A non-POD type should be defined as class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
require_class_for_pod	Whether POD types should be classes as well	False
target		implementation

Possible Messages

Name	Message
cpp_struct	Use of struct in C++ unit.

AutosarC++17_10-A11.0.2

A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
no_function_member	Struct shall not provide any member functions or methods.
no_struct_as_base	A struct shall not be a base of another struct or class.
no_struct_inheritance	A struct shall not inherit from another struct or class.
non_public_member	Structs shall only have public data members.

AutosarC++17_10-A11.3.1

Friend declarations shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
friend_class	Do not use friend class/struct/union declarations.
friend_decl	Do not use friend declarations.

AutosarC++17_10-A12.0.1

If a class declares a copy or move operation, or a destructor, either via "=default", "=delete", or via a user-provided declaration, then all others of these five special member functions shall be declared as well.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_defaulted_destructor_only	Allow classes with a defaulted destructor and no other special member functions.	False
allow_destructor_only	Allow all destructors without copy or move constructors	False
allow_empty_destructor	Allow empty destructors without copy or move constructors.	False
allow_missing_destructor	Suppress messages about missing destructors.	False
enforcement		automated
ignore_pod_classes	Whether POD classes should be checked at all	False
level		required
target		implementation

Possible Messages

Name	Message
missing_constructor_and_asgn	Class with destructor should also declare a copy or move constructor and assignment operator.
missing_copy_asgn	Class with copy constructor is missing copy assignment operator.
missing_copy_constructor	Class with copy assignment operator is missing copy constructor.
missing_destructor	Class with copy or move constructors or assignment operators should also declare a destructor.
missing_move_asgn	Class with move constructor is missing move assignment operator.
missing_move_constructor	Class with move assignment operator is missing move constructor.

AutosarC++17_10-A12.1.1

Constructors shall explicitly initialize all virtual base classes, all direct non-virtual base classes and all non-static data members.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
init_functions	Names of functions to be inspected as well when called directly from constructor.	{'Init', 'init'}
inspect_directly_called_methods	Inspect all methods directly called from constructor.	False
level		required
only_member_initializer_list	Only inspect member initializer list and not the constructor body/methods.	False
report_missing_field_constructors	Enables detection of constructors which rely on implicit field constructor calls.	True
target		implementation

Possible Messages

Name	Message
implicit_field_init	Field is only implicitly initialized in constructor.
missing_base_class_init	Base class is not explicitly initialized in constructor.
missing_field_init	Field is not initialized in constructor.

AutosarC++17_10-A12.1.2

Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_member_initializer_list	Only inspect member initializer list and not the constructor body/methods.	True
target		implementation

Possible Messages

Name	Message
nsdmi_mixed	Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type.

AutosarC++17_10-A12.1.3

If all user-defined constructors of a class initialize data members with constant values that are the same across all constructors, then data members shall be initialized using NSDMI instead.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_constructors_of_types	List of fully qualified type names which constructors are considered literals, without regard to the actual arguments.	[]
allow_literal_like_constructors	Whether to consider constructors that take only literals as arguments a literal. If set to true, this is checked recursively.	False
enforcement		automated
level		required
min_number_common_inits	Minimum number (inclusive) of common initializations to enforce use of NSDMI.	1
target		implementation

Possible Messages

Name	Message
use_nsdmi	Use NSDMI for common constant initializations ({}).

AutosarC++17_10-A12.1.4

All constructors that are callable with a single argument of fundamental type shall be declared explicit.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
consider_only_fundamental_types	Whether this check should be limited to single arguments of fundamental type or should also be applied to user defined types.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
constructor_missing_explicit	Constructor shall be declared explicit

AutosarC++17_10-A12.1.5

Common class initialization for non-constant members shall be done by a delegating constructor.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_copy_move_constructor	Whether to allow direct field initialization without a delegating constructor call for copy and move constructors.	False
enforcement		partially automated
level		required
target		implementation

Possible Messages

Name	Message
delegating_constructor	Use delegating constructor for common initialization.

AutosarC++17_10-A12.1.6

Derived classes that do not need further explicit initialization and require all the constructors from the base class shall use inheriting constructors.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_inheriting	Use inheriting constructors if possible.

AutosarC++17_10-A12.4.1

Destructor of a base class shall be public virtual, public override or protected non-virtual.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
base_class_destructor	Destructor of a base class shall be public virtual, public override or protected non-virtual.

AutosarC++17_10-A12.4.2

If a public destructor of a class is non-virtual, then the class should be declared final.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
non_final	If a public destructor of a class is non-virtual, then the class should be declared final.

AutosarC++17_10-A12.6.1

All class data members that are initialized by the constructor shall be initialized using member initializers.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
init_functions	Names of functions to be inspected as well when called directly from constructor.	{'Init', 'init'}
inspect_directly_called_methods	Inspect all methods directly called from constructor.	False
level		required
only_member_initializer_list	Only inspect member initializer list and not the constructor body/methods.	True
report_missing_field_constructors	Enables detection of constructors which rely on implicit field constructor calls.	True
target		implementation

Possible Messages

Name	Message
implicit_field_init	Field is only implicitly initialized in constructor.
missing_field_init	Field is not initialized in constructor.

AutosarC++17_10-A12.8.1

A copy constructor shall only initialize its base classes and the non-static members of the class of which it is a member.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_copy_constructor	Whether to report side effects on copy constructors.	True
report_move_constructor	Whether to report side effects on move constructors.	True
target		implementation

Possible Messages

Name	Message
copy_ctor_with_side_effect	Copy Constructor has side-effect
move_ctor_with_side_effect	Move Constructor has side-effect

AutosarC++17_10-A12.8.2

User-defined copy and move assignment operators should use user-defined no-throw swap function.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
asgn_no_swap	A move/copy assignment operator shall use a no-throw swap function.
asgn_not_nothrow	Used swap function is not no-throw.

AutosarC++17_10-A12.8.3

Moved-from object shall not be read-accessed.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		partially automated
inspect_class_field_moves	Detecting moved-from accesses on class fields requires heavy computation. This can be disabled by setting this option to False, but would result in false-positives in the context of class fields.	True
level		required
read_move_function_exceptions	Names of functions that are considered to leave the moved from objects in a well-specified state and are therefore except.	['std::unique_ptr::unique_ptr', 'std::unique_ptr::operator=', 'std::shared_ptr::shared_ptr', 'std::shared_ptr::operator=', 'std::weak_ptr::weak_ptr', 'std::weak_ptr::operator=', 'std::basic_filebuf::basic_filebuf', 'std::basic_filebuf::operator=', 'std::thread::thread', 'std::thread::operator=', 'std::unique_lock::unique_lock', 'std::unique_lock::operator=', 'std::shared_lock::shared_lock', 'std::shared_lock::operator=', 'std::promise::promise', 'std::promise::operator=', 'std::future::future', 'std::future::operator=', 'std::shared_future::shared_future', 'std::shared_future::operator=', 'std::packaged_task::packaged_task', 'std::packaged_task::operator=']
read_move_type_exceptions	Names of types that are considered to be left well-specified state after a move and are therefore except.	['std::basic_ios']
target		implementation

Possible Messages

Name	Message
moved_from_read	Don't read-access a moved-from object

AutosarC++17_10-A12.8.4

Move constructor shall not initialize its class members and base classes using copy semantics.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
move_constructor	Move constructor shall not initialize using copy semantics.

AutosarC++17_10-A12.8.6

Copy and move constructors and copy assignment and move assignment operators shall be declared protected or defined "=delete" in base class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
special_asgn	Copy and move assignment operators shall be declared protected or defined "=delete" in base class.
special_ctor	Copy and move constructors shall be declared protected or defined "=delete" in base class.

AutosarC++17_10-A12.8.7

Assignment operators should be declared with the ref-qualifier &.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
asgn_missing	Assignment operators should be declared with the ref-qualifier &.

AutosarC++17_10-A13.1.1

User-defined literals shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
user_defined_literal	User-defined literals shall not be used.

AutosarC++17_10-A13.1.2

User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
user_defined_literal_naming	User-defined literals shall have a suffix matching "[a-zA-Z]+".

AutosarC++17_10-A13.1.3

User defined literals operators shall only perform conversion of passed parameters.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
literal_side_effects	User-defined literals shall not have side effects.

AutosarC++17_10-A13.2.1

An assignment operator shall return a reference to "this".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
asgn_ref_this	An assignment operator shall return a reference to "this".

AutosarC++17_10-A13.2.2

A binary arithmetic operator and a bitwise operator shall return a "prvalue".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_bitwise_shift	Whether to allow non-basic values for operator>> or operator<<.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
arith_bitwise_basic_value	Binary arithmetic or bitwise operator shall return a basic value.

AutosarC++17_10-A13.2.3

A relational operator shall return a boolean value.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
relational_bool	A relational operator shall return a boolean value.

AutosarC++17_10-A13.3.1

A function that contains "forwarding reference" as its argument shall not be overloaded.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
overloaded_fref	Functions that contain forwarding reference parameters shall not be overloaded

AutosarC++17_10-A13.5.1

If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
array_access_non_const	No const version of operator[] implemented.

AutosarC++17_10-A13.5.2

All user-defined conversion operators shall be defined explicit.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
user_defined_conversion_explicit	All user-defined conversion operators shall be defined explicit.

AutosarC++17_10-A13.6.1

Digit sequences separators ' shall only be used as follows: (1) for decimal, every 3 digits, (2) for hexadecimal, every 2 digits, (3) for binary, every 4 digits.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
digit_separator	Possibly unreadable digit separators.

AutosarC++17_10-A15.1.1

Only instances of types derived from std::exception shall be thrown.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throwing_std_exception	Only instances of types derived from std::exception shall be thrown.

AutosarC++17_10-A15.1.2

An exception object should not have pointer type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throwing_pointer	Exception object of pointer type

AutosarC++17_10-A15.1.3

All thrown exceptions should be unique.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
throwing_duplicate	All thrown exceptions should be unique.

AutosarC++17_10-A15.2.1

Constructors that are not noexcept shall not be invoked before program startup.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
static_field_only_noexcept	Constructor called that may throw an exception in static field.
static_variable_only_noexcept	Constructor called that may throw an exception in static variable.

AutosarC++17_10-A15.3.1

Unchecked exceptions should be handled only in main or thread's main functions.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
autosar_exception_model	The default model for what counts as an checked / unchecked exception.	17_03
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
unchecked_exception	Handle unchecked exceptions only in main or thread's main function.

AutosarC++17_10-A15.3.3

There should be at least one exception handler to catch all otherwise unhandled exceptions.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
external_base_exceptions	Sequence of external/third-party exception base-classes that should be caught	[]
inspect_thread_main	Whether to also inspect thread main functions.	True
level		required
std_base_exceptions	Sequence of cpp-std exception base-classes that should be caught	[]
target		implementation

Possible Messages

Name	Message
missing_catch_all	Catch-all required around main program body
missing_catch_handler	Handler for {} needed

AutosarC++17_10-A15.3.4

Catch-all (ellipsis and std::exception) handlers shall be used only in [a] main, [b] task main functions, [c] in functions that are supposed to isolate independent components and [d] when calling third-party code that uses exceptions not according to AUTOSAR C++14 guidelines.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_extern_c_functions	Whether to allow a catch-all in `extern C` functions.	False
allow_in_functions_matching	If not None, a `re.compile()`ed object where functions whose qualified name matches the regex are allowed to contain catch-alls.	None
allow_in_main	Whether to allow a catch-all in the main() function.	True
allow_in_thread_main	Whether to allow a catch-all in thread-main functions.	True
allow_rethrow	Whether to allow a catch-all with a re-throw.	False
autosar_exception_model	The default model for what counts as an checked / unchecked exception.	<bauhaus.ir.autosar.exceptions.autosar_exceptions.AutosarExceptionModel object at 0x7dfa83c4650>
disallow_std_exception	Whether to consider catching the literal std::exception as a catch-all.	True
enforcement		non-automated
level		required
target		implementation

Possible Messages

Name	Message
catch_all	Use of catch(...).
catch_std_exception	Catching std::exceptions is too general.

AutosarC++17_10-A15.3.5

A class type exception shall always be caught by reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_class_types	Whether all types should be caught by reference or only class types.	True
target		implementation

Possible Messages

Name	Message
catch_without_reference	A class type exception shall always be caught by reference.

AutosarC++17_10-A15.4.1

Dynamic exception-specification shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

AutosarC++17_10-A15.4.2

If a function is declared to be noexcept, noexcept(true) or noexcept(<true condition>), then it shall not exit with an exception.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
generate_violation_path	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignore_constructor_destructor	Whether to ignore escaping exceptions from constructors and destructors.	False
ignore_throwing_functions	Whether to ignore noexcept specification violations on function that actually throw an exception.	False
ignore_unkown_routines	Whether to ignore extern or only declared routines.	False
level		required
report_noexcept_falseViolations	Whether to report cases where the function is declared noexcept(false), but no exceptions are ever thrown (directly or indirectly).	False
target		implementation

Possible Messages

Name	Message
implicit_noexcept_spec_violation_without	Function implicitly declared noexcept(false) but no exceptions will be thrown.
noexcept_spec_violation_with	Exception violates function's noexcept-specification.
noexcept_spec_violation_without	Function declared noexcept(false) but no exceptions will be thrown.

AutosarC++17_10-A15.4.3

Function's noexcept specification shall be either identical or more restrictive across all translation units and all overriders.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
noexcept_mismatch_redecl	Function's noexcept specification shall be identical
noexcept_mismatch_override	exception specification for virtual function "{}" is incompatible with that of overridden function "{}"

AutosarC++17_10-A15.4.4

A declaration of non-throwing function shall contain noexcept specification.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
generate_violation_path	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignore_constructor_destructor	Whether to ignore escaping exceptions from constructors and destructors.	False
ignore_throwing_functions	Whether to ignore noexcept specification violations on function that actually throw an exception.	True
ignore_unkown_routines	Whether to ignore extern or only declared routines.	True
level		required
report_noexcept_falseViolations	Whether to report cases where the function is declared noexcept(false), but no exceptions are ever thrown (directly or indirectly).	True
target		implementation

Possible Messages

Name	Message
implicit_noexcept_spec_violation_without	Function implicitly declared noexcept(false) but no exceptions will be thrown.
noexcept_spec_missing	Explicit noexcept-specification missing.
noexcept_spec_violation_with	Exception violates function's noexcept-specification.
noexcept_spec_violation_without	Function declared noexcept(false) but no exceptions will be thrown.

AutosarC++17_10-A15.4.5

Checked exceptions that could be thrown from a function shall be specified together with the function declaration and they shall be identical in all function declarations and for all its overriders.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
autosar_exception_model	The default model for what counts as an checked / unchecked exception.	17_03
enforcement		automated
ignore_definitions_if_declaration_documented	Instead of requiring the documentation to be repeated for every declaration (including definition), with this option set, the rule only checks the non-defining declarations if at least one non-defining declaration exists.	False
level		required
match_qualified_name	Matches the fully-qualified name when comparing documented exceptions with what can actually occur. If set to 'False', this rule will accept any suffix of the qualified name of an exception class as the documentation string.	True
target		implementation
throw_marker	The command to document an exception to be thrown.	@throw

Possible Messages

Name	Message
differing_documented	Documented exceptions differ from overridden method.
document_exception	Document checked exception {} using {}.
superflous_documented	Documented exceptions {} probably never thrown.

AutosarC++17_10-A15.4.6

Unchecked exceptions should not be specified together with a function declaration.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
autosar_exception_model	The default model for what counts as an checked / unchecked exception.	17_03
enforcement		automated
level		advisory
target		implementation
throw_marker	The command to document an exception to be thrown.	@throw

Possible Messages

Name	Message
dont_document_exception	Don't document unchecked exception.

AutosarC++17_10-A15.5.1

All user-provided class destructors, deallocation functions, move constructors, move assignment operators and swap functions shall not exit with an exception. A noexcept exception specification shall be added to these functions as appropriate.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
routine_may_except	Function must not exit with an exception.
routine_not_noexcept	Function shall be explicitly declared noexcept if appropriate.

AutosarC++17_10-A15.5.2

Program shall not be abruptly terminated. In particular, an implicit or explicit invocation of std::abort(), std::quick_exit(), std::_Exit(), std::terminate() shall not be done.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
allowed_exceptions	Exceptions that are allowed to escape from destructors.	['bad_alloc', 'bad_cast', 'failure', 'runtime_error', 'system_error']
blacklist	Dictionary of header globbing to (list of) function name globbing(s) of forbidden functions.	dict(...)
constructors	Whether to consider constructors.	False
destructors	Whether to consider destructors.	True
enforcement		partially automated
generate_violation_path	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignore_constructor_destructor	Whether to ignore escaping exceptions from constructors and destructors.	False
ignore_unkown_routines	Whether to ignore extern or only declared routines.	False
inspect_at_exit_handlers	Whether to also inspect at_exit() handlers-functions.	True
inspect_thread_main	Whether to also inspect thread main functions.	True
level		required
report_only_one_exception_per_function	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
target		implementation

Possible Messages

Name	Message
exceptionEscapingConstructor	Escaping exception from constructor.
exceptionEscapingDestructor	Escaping exception from destructor.
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionEscapingMain	Uncaught exception escaping from main or additional entry point
exceptionSpecificationViolation	Exception violates function's exception-specification.
forbiddenLibfuncCall	Call to forbidden function.

AutosarC++17_10-A15.5.3

The terminate() function shall not be called implicitly.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
allowed_exceptions	Exceptions that are allowed to escape from destructors.	['bad_alloc', 'bad_cast', 'failure', 'runtime_error', 'system_error']
constructors	Whether to consider constructors.	False
destructors	Whether to consider destructors.	True
enforcement		automated
generateViolationPath	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignoreConstructorDestructor	Whether to ignore escaping exceptions from constructors and destructors.	True
ignoreThrowingFunctions	Whether to ignore noexcept specification violations on function that actually throw an exception.	False
ignoreUnknownRoutines	Whether to ignore extern or only declared routines.	False
inspectAtExitHandlers	Whether to also inspect at_exit() handlers-functions.	True
inspectThreadMain	Whether to also inspect thread main functions.	True
level		required
reportNoexceptFalseViolations	Whether to report cases where the function is declared noexcept(false), but no exceptions are ever thrown (directly or indirectly).	False
reportOnlyOneExceptionPerFunction	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
required	Dict which lists required operations per resource. The mapping gives each case a description which maps to a dict for key "Required_Functions", "Resource_Parameter_Empty".	dict(...)
resources	Configuration of resources and operations on them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)
target		implementation

Possible Messages

Name	Message
exceptionEscapingConstructor	Escaping exception from constructor.
exceptionEscapingDestructor	Escaping exception from destructor.
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionEscapingMain	Uncaught exception escaping from main or additional entry point
exceptionSpecificationViolation	Exception violates function's exception-specification.
implicitNoexceptSpecViolationWithout	Function implicitly declared noexcept(false) but no exceptions will be thrown.
noexceptSpecViolationWith	Exception violates function's noexcept-specification.
noexceptSpecViolationWithout	Function declared noexcept(false) but no exceptions will be thrown.
possiblyRequiredOperation	This thread is possibly joinable on destructor call
requiredOperation	This thread is joinable on destructor call

AutosarC++17_10-A16.0.1

The pre-processor shall only be used for file inclusion and include guards.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
accept_conditional_includes	Whether to accept #ifs for conditional includes.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_macro_definition	Macros are only allowed for include guards
line_directive	The pre-processor shall only be used for file inclusion and include guards.
object_macro_definition	Macros are only allowed for include guards
pp_if	Conditional compilation is only allowed for include guards
pragma	The pre-processor shall only be used for file inclusion and include guards.
undef	The pre-processor shall only be used for file inclusion and include guards.

AutosarC++17_10-A16.2.1

The ', ", /*, //, \ characters shall not occur in a header file name or in #include directive.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
check_quotation_mark	Whether system includes should be checked for containing a ".	True
enforcement		automated
forbidden	The substrings to check for. " will be added for system-includes.	set(['//', '\\\\', """", /*])
level		required
target		implementation

Possible Messages

Name	Message
nonstandard_include_character	Non-standard character in #include directive

AutosarC++17_10-A16.6.1

#error directive shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
error	Use of #error

AutosarC++17_10-A16.7.1

The #pragma directive shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pragma	Use of #pragma

AutosarC++17_10-A17.0.1

Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
additional_reserved_identifiers	Names listed here are seen as violations as well.	set(['assert', 'errno', 'defined'])
allow_reserved_identifier_as_include_guard	If True, macros used for include guards are not checked for being a reserved identifier.	False
check_for_keyword	Whether macro names being keywords should be reported as well.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
macro_having_reserved_name	Definition of reserved identifier or standard library element
undef_of_reserved_name	#undef of reserved identifier or standard library element

AutosarC++17_10-A18.0.1

The C library shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
c_lib_header	Include <{}> instead of <{}>.
cpp_lib_header_with_suffix	Include <{}> instead of <{}>.

AutosarC++17_10-A18.0.2

The library functions atof, atoi and atol from library <cstdlib> shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	stdlib
symbols	Names of symbols which are forbidden.	['atof', 'atoi', 'atol']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++17_10-A18.0.3

The library <clocale> (locale.h) and the setlocale function shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	locale
symbols	Names of symbols which are forbidden.	['setlocale']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++17_10-A18.1.1

C-style arrays should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
c_array_used	C-style arrays should not be used.

AutosarC++17_10-A18.1.2

The std::vector<bool> specialization shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
type_used	The std::auto_ptr type shall not be used.

AutosarC++17_10-A18.1.3

The std::auto_ptr type shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
type_used	The std::auto_ptr shall not be used.

AutosarC++17_10-A18.1.4

A pointer pointing to an element of an array of objects shall not be passed to a smart pointer of single object type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
type_used	{} shall not refer to an array type.

AutosarC++17_10-A18.1.5

The std::unique_ptr shall not be passed to a function by const reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
const_ref_unique_ptr	const std::unique_ptr& as parameter.

AutosarC++17_10-A18.1.6

All std::hash specializations for user-defined types shall have a noexcept function call operator.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
specialization_of_hash_except	std::hash specialization shall have noexcept call operator.

AutosarC++17_10-A18.5.1

Functions malloc, calloc, realloc and free shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_user_new_delete_operator	Whether to allow the library functions inside user defined new/delete operator overloads.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
stdlib_memory_function_call	Dynamic memory management function from <stdlib.h> is called.

AutosarC++17_10-A18.5.2

Operators new and delete shall not be called explicitly.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_for_types_and_subtypes	A list of fully-qualified type names for which the new and delete operators are allowed. This might be necessary if you are working with 3rd party libraries like Qt.	[]
enforcement		partially automated
level		required
target		implementation

Possible Messages

Name	Message
new_delete_call	Operators new and delete shall not be called explicitly.

AutosarC++17_10-A18.5.3

The form of delete operator shall match the form of new operator used to allocate the memory.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. See the base class for more options.	dict{...}
target		implementation

Possible Messages

Name	Message
possible_wrong_release	Resource possibly released using wrong function [allocation used {0}]
wrong_release	Resource released using wrong function [allocation used {0}]

AutosarC++17_10-A18.5.4

If a project has sized or unsized version of operator "delete" globally defined, then both sized and unsized versions shall be defined.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
only_sized	Only the sized version of operator delete is defined.
only_unsized	Only the unsized version of operator delete is defined.

AutosarC++17_10-A18.5.8

Objects that do not outlive a function shall have automatic storage duration.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
allow_smart_ptr_from_function_return	Whether to allow a smart pointer returned from a function as a local variable in a function. Calling the smart pointer constructors or the helper functions std::make_{shared,unique,..} will remain a violation.	True
enforcement		partially automated
level		required
target		implementation

Possible Messages

Name	Message
could_be_scoped	Local objects shall be allocated on the stack.

AutosarC++17_10-A18.9.1

The std::bind shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	functional
symbols	Names of symbols which are forbidden.	['std::bind']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	False

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity std::bind from <{}>.

AutosarC++17_10-A18.9.2

Forwarding values to other functions shall be done via: (1) std::move if the value is an rvalue reference, (2) std::forward if the value is forwarding reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
forwarding_forwarding_reference	Use std::forward if the value is a forwarding reference.
forwarding_rvalue_reference	Use std::move if the value is a rvalue reference.

AutosarC++17_10-A18.9.3

The std::move shall not be used on objects declared const or const&.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
std_move_const	Call to std::move with argument declared const/const&.

AutosarC++17_10-A23.0.1

An iterator shall not be implicitly converted to const_iterator.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
implicit_const_iterator	An iterator shall not be implicitly converted to const_iterator.

AutosarC++17_10-M0.1.1

There shall be no unreachable code.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_dead_if_branch_with_const_literal_condition	Whether bodies of if-branches with constant literal conditions should be reported.	True
target		implementation

Possible Messages

Name	Message
unreachable_code	Unreachable code

AutosarC++17_10-M0.1.2

A project shall not contain infeasible paths.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_do_while_false	Whether do ... while[0] should be reported.	True
report_while_true	Whether while[1] ... should be reported.	True
target		implementation

Possible Messages

Name	Message
dead_false_branch	Condition is always true: Else branch is dead
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Parameter condition is always true: Else branch is dead
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Parameter comparison to NULL is always true: Else branch is dead
dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Parameter comparison to NULL is always false: Then branch is dead
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Parameter condition is always false: Then branch is dead
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Condition is always false: Then branch is dead
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Variable condition is always true: Else branch is dead
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Variable condition is always false: Then branch is dead
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead

AutosarC++17_10-M0.1.3

A project shall not contain unused variables.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
ignore_types	Variables of types named here are ignored in this check. Globbing patterns are supported.	[]
level		required
only_check_unit_locals	Whether only global static variables and local variables should be checked.	False
report_global_constants	Whether unused global constants should be reported.	False
report_undefined_variables	Whether only-declared variables should be reported.	True
target		implementation
treat_initialization_as_use	Whether an explicit initialization should be considered a use of the variable.	True
treat_side_effect_constructors_as_use	Whether variables should be seen as used if they are of a class type and initialized through a call to a constructor having a side-effect, e.g. std::lock_guard	False

Possible Messages

Name	Message
unused_field	Unused field
unused_variable	Unused variable

AutosarC++17_10-M0.1.4

A project shall not contain non-volatile POD variables having only one use.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_if_address_taken	Allow using a variable only once if that use involves taking the address of the variable.	False
enforcement		automated
level		required
report_fields	Select whether fields used only once should be reported as well.	True
target		implementation

Possible Messages

Name	Message
field_referenced_only_once	{ } referenced only once
unreferenced_initialized_field	{ } initialized but not referenced
unreferenced_initialized_variable	{ } initialized but not referenced
variable_used_only_once	{ } referenced only once

AutosarC++17_10-M0.1.5

A project shall not contain unused type declarations.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unused_type	Unused type declaration

AutosarC++17_10-M0.1.8

All functions with void return type shall have external side effect(s).

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
assume_effect_when_undefined	Whether all calls to undefined (external) functions should be seen as having a side-effect.	True
enforcement		automated
exceptions	Names of functions that should be excluded from the check.	main
exclude_constructors	If True, tolerate constructors with no side-effect.	False
exclude_virtual_destructors	If True, tolerate virtual destructors with no side-effect.	False
level		required
target		implementation

Possible Messages

Name	Message
void_func_without_side_effect	Void function has no external side-effect

AutosarC++17_10-M0.1.9

There shall be no dead code.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
allow_void_var	Whether {void}var; should be allowed or reported.	True
assume_effect_for_all_calls	Whether all calls should be seen as having a side-effect.	False
assume_effect_when_undefined	Whether all calls to undefined (external) functions should be seen as having a side-effect.	True
enforcement		automated
level		required
report_dead_initializations	Whether initialisations may be reported as dead.	True
report_do_while_false	Whether do ... while(0) should be reported.	True
report_while_true	Whether while(1) ... should be reported.	True
target		implementation
tolerate_void_cast	Whether a {void} cast is accepted.	True

Possible Messages

Name	Message
conditional_unused_def	Result of assignment is not used along some path(s) (disabled)
dead_false_branch	Redundant code, condition is always true
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Redundant code, parameter condition is always true
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Redundant code, parameter comparison to NULL is always true
dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Redundant code, parameter comparison to NULL is always false
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Redundant code, parameter condition is always false
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Redundant code, condition is always false
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Redundant code, variable condition is always true
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Redundant code, variable condition is always false
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead
init_used_in_other_isr	Initialization is only used in some interrupt handler
no_effect	Non-null statement without side-effect
removable_declaration	Declaration can be removed
removable_statement	Statement can be removed
unused_def	Dead (redundant) code: result of assignment is not used
unused_init	Unused initialization
used_in_other_isr	Result of assignment is only used in some interrupt handler

AutosarC++17_10-M0.1.10

Every defined function shall be called at least once.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
ignore_copy_assignment_operators	Whether uncalled copy assignment operators are allowed. Other rules like the "rule of three" might require copy assignment operators as part of good class design, even if there are currently no callers of the copy assignment operator.	False
ignore_copy_constructors	Whether uncalled copy constructors are allowed. Other rules like the "rule of three" might require copy constructors as part of good class design, even if there are currently no callers of the copy constructor.	False
level		required
only_check_internal_or_private_methods	Whether only function with internal linkage or private functions should be checked.	False
only_check_static_or_private_methods	Whether only static functions or private functions should be checked.	False
only_check_unit_locals	Whether only global static functions should be checked.	False
target		implementation

Possible Messages

Name	Message
defined_function_not_called	Defined function is not called

AutosarC++17_10-M0.2.1

An object shall not be assigned to an overlapping object.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
assignment_to_overlapping	An object shall not be assigned to an overlapping object

AutosarC++17_10-M0.3.1

Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
abstract_interpretation_div_by_zero	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
abstract_interpretation_overflow	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
argument_checks	Configuration of [dis]allowed values in addition to what iranalysis.config provides. This is a dict key -> dict with keys 'Functions', 'Parameter', and 'Allowed'/'Disallowed'. The parameter number counting starts at 0. Arguments passed in for this parameters are checked against the value specification given with '[Dis]allowed'.	dict(...)
enforcement		automated
execute_checks_covered_in_other_rules	Whether iranalysis-based checks covered by other Misra-C++:2008 rules should be executed here as well.	False
forbidden	Dict which lists forbidden operations per resource. The mapping gives each case a description which maps to a dict for key "Forbidden_Functions", "Mode_Parameter", "Mode".	dict(...)
level		required
modes	Describes when an allocating call opens a file in read or write mode. This is a dict resource -> dict with keys "Resource_Parameter", "Check_Write_Write", and "Modes" (being a dict with key "Read" and "Write")	dict(...)
perform_global_analysis	Whether global analysis (using iranalysis) should be done, or just a local analysis.	True
report_dead_initializations	Whether initialisations may be reported as dead.	True
report_do_while_false	Whether do ... while[0] should be reported.	True
report_while_true	Whether while[1] ... should be reported.	True
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)
target		implementation

Possible Messages

Name	Message
argumentViolation	Argument not within allowed values
arithmetic_out_of_bounds	Pointer arithmetic on {1} creates pointer outside array bounds of {0}
asgn_not_an_enumerator	Assignment of value which does not correspond to an enumerator
asgn_out_of_enum_range	Assignment of value outside the range of the enumerators
cast_overflow	Cast on result of arithmetic computation may cause overflow
cast_truncate	Cast may truncate value
cast_underflow	Cast on result of arithmetic computation may cause underflow
conditional_unused_def	Result of assignment is not used along some path(s)
deadCatch	Dead exception handler
dead_false_branch	Condition is always true: Else branch is dead
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Parameter condition is always true: Else branch is dead
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Parameter comparison to NULL is always true: Else branch is dead

dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Parameter comparison to NULL is always false: Then branch is dead
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Parameter condition is always false: Then branch is dead
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Condition is always false: Then branch is dead
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Variable condition is always true: Else branch is dead
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Variable condition is always false: Then branch is dead
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead
division_by_zero	Division by zero
double_free	Dynamic memory released here was already released earlier
escaping_address	Escaping address of local variable
forbidden_operation	The way in which this resource was allocated forbids this operation
init_used_in_other_isr	Initialization is only used in some interrupt handler
memory_leak	Call allocates leaking memory
mode_conflict	Same file used for both reading and writing
modulo_by_zero	Modulo by zero
null_deref	Pointer is NULL at dereference
out_of_bounds	Access into array is out of bounds
overflow	Arithmetic computation may cause overflow
pass_as_pointer_to_const_param	Passing uninitialized variable by pointer as function parameter with pointer-to-const type
possible_argumentViolation	Argument possibly not within allowed values
possible_division_by_zero	Possible division by zero
possible_double_free	Dynamic memory released here possibly already released earlier
possible_indirect_out_of_bounds	Pointer-indirect access through {1} might be out of bounds accessing {0}
possible_memory_leak	Call allocates possibly leaking memory
possible_mode_conflict	Same file possibly used for both reading and writing
possible_modulo_by_zero	Possible modulo by zero
possible_null_deref	Pointer may be NULL at dereference
possible_out_of_bounds	Access into array might be out of bounds
possible_stack_free	{ } possibly released by call to { } is a stack object
possible_uninit	Use of possibly uninitialized variable
possible_unrelated_ptr_comparison	Comparing possibly unrelated pointers
possible_unrelated_ptr_subtraction	Subtracting possibly unrelated pointers
possible_use_after_free	Dynamic memory possibly used after it was previously released

possible_wrong_release	Resource possibly released using wrong function (allocation used {0})
possiblyEscapingAddress	Possibly escaping address of local variable (as target of {1})
possiblyForbiddenOperation	The way in which this resource was allocated possibly forbids this operation
stack_free	{ } released by call to { } is a stack object
underflow	Arithmetic computation may cause underflow
uninit	Use of uninitialized variable
unrelated_ptr_comparison	Comparing unrelated pointers
unrelated_ptr_subtraction	Subtracting unrelated pointers
unused_def	Result of assignment is not used
unused_init	Unused initialization
use_after_free	Dynamic memory used after it was previously released
used_in_other_isr	Result of assignment is only used in some interrupt handler
wrong_release	Resource released using wrong function (allocation used {0})

AutosarC++17_10-M0.3.2

If a function generates error information, then that error information shall be tested.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
inspect_template_instances	Whether calls in template instances should be reported.	False
level		required
relevant_functions	If provided, only calls to these functions are inspected.	[]
target		implementation

Possible Messages

Name	Message
discarded_error_with_entity	Integral return value (possibly error code) discarded.
discarded_error_without_entity	Integral return value (possibly error code) discarded.

AutosarC++17_10-M0.4.2

Use of floating-point arithmetic shall be documented.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
float_arithmetric	Use of floating-point arithmetic

AutosarC++17_10-M2.10.1

Different identifiers shall be typographically unambiguous.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
distinguish_c_name_spaces	Whether to allow similar identifiers in different C name spaces (type tag/label/member/ordinary identifier)	True
distinguish_macro_names	Whether to allow macros to same similar names as other identifiers.	False
distinguish_type_names	Whether to allow types to have similar names as other identifiers. For example, a variable declaration "MyClass myClass;" is allowed by this option. Note: unlike the option 'distinguish_c_name_spaces', this option also distinguishes typedef names, not just struct/class tag names.	False
enforcement		automated
exclude_system_headers	Whether to exclude identifiers appearing in system headers	False
level		required
normalizations	Which pairs of characters should be seen as ambiguous	[('0', 'O'), ('1', 'l'), ('I', 'l'), ('i', 'l'), ('5', 'S'), ('2', 'Z'), ('8', 'B'), ('rn', 'm'), ('n', 'h'), ('_', '')]
target		implementation

Possible Messages

Name	Message
ambiguous	Identifiers are typographically ambiguous
casing	Identifiers only differ in casing

AutosarC++17_10-M2.10.3

A typedef name (including qualification, if any) shall be a unique identifier.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation
tolerate_macros	Whether #define and #undef using the typedef's name is allowed	False
tolerate_typedef_entity	Whether the entity forming the typedef's underlying type can have the same name.	False

Possible Messages

Name	Message
reused_typedef	Typedef name reused.

AutosarC++17_10-M2.10.6

If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
invert_findings	Whether to invert primary and secondary SLocs for violations	False
level		required
target		implementation

Possible Messages

Name	Message
reused_type	Type name reused in same scope

AutosarC++17_10-M2.13.2

Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
octal_escape_sequence	Use of octal escape sequence.
octal_literal	Use of octal literal.

AutosarC++17_10-M2.13.3

A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_only_hex_and_octal	Whether only hex and octal literals should be checked.	True
consider_conversion	Selects whether literals with an implicit conversion from signed to unsigned are reported.	False
consider_pp_expressions	Whether literals in #if conditions are checked.	False
enforcement		automated
level		required
report_all_unsuffixed_hex_and_octal	Whether hex and octal literals should always be reported when they are unsuffixed, independent of the environment	False
target		implementation

Possible Messages

Name	Message
missing_hex_suffix	Hexadecimal and octal constants require "U" suffix
missing_u_suffix	Constant of unsigned type requires "U" suffix

AutosarC++17_10-M2.13.4

Literal suffixes shall be upper case.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
lowercase_suffix	Literal suffix should be upper case

AutosarC++17_10-M3.1.2

Functions shall not be declared at block scope.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
local_function_declaration	Functions shall not be declared at block scope.

AutosarC++17_10-M3.2.1

All declarations of an object or function shall have compatible types.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_parameter_type_qualifier_differences	If enabled, do not report a violation for different top-level type qualifiers in parameters: int f(int p); int f(const int p) {} Top-level qualifiers are only relevant for the implementation of the function and have no effect on callers, so it is not necessary to replicate them in the header file.	False
check_parameter_types	Whether parameter types should be compared	False
check_undefined	Whether only-declared routines and variables should also be checked.	True
enforcement		automated
level		required
require_exact_match	Whether to check for identical or compatible types (for routine return types).	False
target		implementation

Possible Messages

Name	Message
decl_return_type_mismatch	Return type at declarations differs
decl_type_mismatch	Type at declarations differs
parameter_type_mismatch	Parameter type at definition differs from type at declaration
return_type_mismatch	Return type at definition differs from type at declaration
type_mismatch	Type at definition differs from type at declaration

AutosarC++17_10-M3.2.2

The One Definition Rule shall not be violated.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
class_struct_difference	{}
different_enumerators	{}
different_field_types	{}
different_fields	{}
general_odrViolation	{}

AutosarC++17_10-M3.2.3

A type, object or function that is used in multiple translation units shall be declared in one and only one file.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
allow_only_one_decl	Enables message if there are multiple declarations but all in same file.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_declared_in_multiple_files	Function declared in more than one file.
function_declared_multiple_times	Function declared more than once.
type_declared_in_multiple_files	Type declared in more than one file.
type_declared_multiple_times	Type declared more than once.
variable_declared_in_multiple_files	Variable declared in more than one file.
variable_declared_multiple_times	Variable declared more than once.

AutosarC++17_10-M3.2.4

An identifier with external linkage shall have exactly one definition.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
allow_undefined_specialized_template	Select whether undefined templates should be reported if specializations of them exist.	True
allow_unselected_declarations	Select whether to report declarations with no definition if they are unused.	True
allowed_undefined_functions	Regular expressions for functions which are tolerated without a definition, e.g. compiler intrinsics.	['_.*']
allowed_undefined_types	Regular expressions for types which are tolerated without a definition.	['_.*']
allowed_undefined_variables	Regular expressions for variables which are tolerated without a definition.	['_.*']
check_composite_types	Check class/struct/union types for having no definition even if they have no external linkage.	False
check_types_only_with_external_linkage	Whether the types should only be checked when they have external linkage	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
composite_defined_multiple_times	Composite type has multiple definitions.
enum_defined_multiple_times	Enumeration type has multiple definitions.
function_defined_multiple_times	Type, object or function has multiple definitions.
undefined_function	Type, object or function without definition
undefined_type	Type without definition
undefined_variable	Type, object or function without definition
variable_defined_multiple_times	Type, object or function has multiple definitions.

AutosarC++17_10-M3.3.2

If a function has internal linkage then all re-declarations shall include the static storage class specifier.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_decl_missing_static	Declaration should use static storage class specifier
function_def_missing_static	Definition should use static storage class specifier
variable_decl_missing_static	Declaration should use static storage class specifier
variable_def_missing_static	Definition should use static storage class specifier

AutosarC++17_10-M3.4.1

An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_moving_to_other_primary_file	When a variable is only used in a single file X, but currently implemented in a different file Y, this controls whether the check suggests to move it into X and make it static there.	False
check_classes	Whether to report structs/classes/unions which are only used in a single function or file	True
check_loop_counter	Whether to report for-loop counters being declared before the loop that could be declared in the for-init part instead from C99 on	False
enforcement		automated
exclude_function_locals	If True, variables that could even be function-local are not reported.	True
exclude_undefined	Whether only-declared symbols should be reported as well	True
level		required
move_global_const_into_function	Controls suggestions for global constants that could be declared locally in a function.	True
only_check_unit_locals	Whether only global static variables should be checked.	False
target		implementation

Possible Messages

Name	Message
locality_block	{ } can be declared in a more local scope.
locality_file	{ } can be declared locally in primary file.
locality_function	Global { } can be declared inside function.
locality_loop_init	{ } can be declared in the for-loop's initialization.
var_file_static	{ } can be declared static in primary file.

AutosarC++17_10-M3.9.1

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_missing_qualifiers	If True, tolerate differences in the use of explicit namespace/class qualifiers.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
parameter_type_tokens_mismatch	Type of redeclaration is not token-for-token identical
return_type_tokens_mismatch	Type of redeclaration is not token-for-token identical
variable_type_tokens_mismatch	Type of redeclaration is not token-for-token identical

AutosarC++17_10-M3.9.3

The underlying bit representations of floating-point values shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
float_bit_representation	Use of bit representation of a float value.

AutosarC++17_10-M4.5.1

Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_sizeof	Allows the use of sizeof(bool_var). The Misra rule forbids all uses of expressions of type bool except for those explicitly allowed; and does not allow sizeof. However this is likely an oversight; there is no good reason to prevent the use of sizeof(bool_var). This option also allows alignof(bool_var) in C++11.	False
allow_unevaluated_operator	Allows the use of boolean expressions in contexts where the expression is not evaluated. As of C++17, these are: typeid(bool_var), noexcept(bool_var) and decltype(bool_var). sizeof(bool_var) is controlled by the separate option 'allow_sizeof'.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bool_operand_in_bad_operator	Use of boolean operand in '{}' operator
bool_operand_outside_logical_and_relational_op	Use of boolean operand with integral promotion

AutosarC++17_10-M4.5.3

Expressions with type [plain] char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
char_operand_outside_comparison	Use of character operand in forbidden context

AutosarC++17_10-M4.10.1

NULL shall not be used as an integer value.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
null_as_int	Use of NULL as integer value

AutosarC++17_10-M4.10.2

Literal zero (0) shall not be used as the null-pointer-constant.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
require_nullptr	Require C++11 nullptr instead of macro NULL or literal zero (0).	False
target		implementation

Possible Messages

Name	Message
zero_as_null	Use of literal zero (0) as null-pointer-constant, use {} instead

AutosarC++17_10-M5.0.2

Limited dependence should be placed on C++ operator precedence rules in expressions.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_algebraic_order	Whether parens can be omitted for multiplicative operators as operands in additive operators	False
allow_redundant_parentheses_in_logical_operator	Allow redundant parentheses around unary operators when they appear as operand of '&&' or ' '. This option can be used to avoid a conflict with MisraC++-5.2.1, which requires parentheses in these instances.	True
enforcement		automated
level		required
report_unnecessary_parentheses	Controls whether unnecessary use of parentheses on the right side of assignments or around unary operators are reported.	True
target		implementation

Possible Messages

Name	Message
assignment_rhs_in_parens	No parentheses required for right-hand side of assignment
missing_parens_depends_on_precedence	Parentheses required to avoid dependence on precedence rules
unary_op_in_parens	No parentheses required for unary operator

AutosarC++17_10-M5.0.3

A value expression shall not be implicitly converted to a different underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes], [SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
target		implementation
type_category	Selection of type categories to consider for type changes inside same category.	[SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Result of cvalue expression implicitly converted to different underlying type
cast_changes_type_inside_category	Result of cvalue expression implicitly converted to different underlying type

AutosarC++17_10-M5.0.4

An implicit integral conversion shall not change the signedness of the underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_constant_conversions	If this option is enabled, the rule is relaxed to allow implicit conversions of constant integer expressions whenever the constant value fits into the target type.	True
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
report_signed_to_unsigned	Whether conversion from signed to unsigned shall be reported.	True
report_unsigned_to_signed	Whether conversion from unsigned to signed shall be reported.	True
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_from_signed_to_unsigned	Implicit integral conversion changes signedness of underlying type
cast_from_unsigned_to_signed	Implicit integral conversion changes signedness of underlying type

AutosarC++17_10-M5.0.5

There shall be no implicit floating-integral conversions.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes], [FloatingTypes]], [[FloatingTypes], [SignedTypes, UnsignedTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Implicit floating-integral conversion

AutosarC++17_10-M5.0.6

An implicit integral or floating-point conversion shall not reduce the size of the underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
report_narrowing	Whether narrowing casts should be reported.	True
report_widening	Whether widening casts should be reported.	False
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
size_source_category	Selection of type categories of the operand being cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
size_target_category	Selection of type categories appearing as target types of the cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
narrowing_cast	Implicit conversion reduces size of underlying type
widening_cast	Conversion to larger type

AutosarC++17_10-M5.0.7

There shall be no explicit floating-integral conversions of a cvalue expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories. [[[SignedTypes, UnsignedTypes, CharacterTypes], [FloatingTypes]], [[FloatingTypes], [SignedTypes, UnsignedTypes, CharacterTypes]]]	
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Explicit floating-integral conversion of cvalue expression

AutosarC++17_10-M5.0.8

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
report_narrowing	Whether narrowing casts should be reported.	False
report_widening	Whether widening casts should be reported.	True
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
size_source_category	Selection of type categories of the operand being cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
size_target_category	Selection of type categories appearing as target types of the cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
narrowing_cast	Conversion to smaller type
widening_cast	Explicit conversion increases size of underlying type of cvalue expression

AutosarC++17_10-M5.0.9

An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
report_signed_to_unsigned	Whether conversion from signed to unsigned shall be reported.	True
report_unsigned_to_signed	Whether conversion from unsigned to signed shall be reported.	True
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_from_signed_to_unsigned	Explicit conversion changes signedness of underlying type of cvalue expression
cast_from_unsigned_to_signed	Explicit conversion changes signedness of underlying type of cvalue expression

AutosarC++17_10-M5.0.10

If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitop_small_without_cast	Bitwise operator requires cast to underlying type on result

AutosarC++17_10-M5.0.11

The plain char type shall only be used for the storage and use of character values.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
char_used_numerically	Plain char datatype used for non-character data.

AutosarC++17_10-M5.0.12

signed char and unsigned char type shall only be used for the storage and use of numeric values.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
numeric_char_used_as_character	Signed/unsigned char datatype used for character data.

AutosarC++17_10-M5.0.14

The first operand of a conditional-operator shall have type bool.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
nonbool_conditional_operator_condition	Condition must have type bool

AutosarC++17_10-M5.0.15

Array indexing shall be the only form of pointer arithmetic.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
indexing_only_on_identifiers	Report array indexing on pointers only for variables (`ptr[i]`), not for other pointer expressions (e.g. `get_ptr()[i]`). This option is meant to suppress the violations introduced by the BAUHAUS-12021 bugfix in version 6.9.6.	False
level		required
target		implementation

Possible Messages

Name	Message
array_indexing_on_pointer	Array indexing only allowed for arrays
pointer_arithmetic	Pointer arithmetic not allowed
pointer_increment_decrement	Pointer arithmetic not allowed

AutosarC++17_10-M5.0.16

A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
arithmetic_on_nonarray	Pointer arithmetic on {1} operates on non-array target {0}
arithmetic_out_of_bounds	Pointer arithmetic on {1} creates pointer outside array bounds of {0}
out_of_bounds	Access into array is out of bounds
possible_indirect_out_of_bounds	Pointer-indirect access through {1} might be out of bounds accessing {0}
possible_out_of_bounds	Access into array might be out of bounds

AutosarC++17_10-M5.0.17

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
possible_unrelated_ptr_subtraction	Subtracting possibly unrelated pointers
unrelated_ptr_subtraction	Subtracting unrelated pointers

AutosarC++17_10-M5.0.18

>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
possible_unrelated_ptr_comparison	Comparing possibly unrelated pointers
unrelated_ptr_comparison	Comparing unrelated pointers

AutosarC++17_10-M5.0.20

Non-constant operands to a binary bitwise operator shall have the same underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitop_with_different_operand_types	Non-constant operands of bitwise operator differ in underlying type
shortcut_bitop_with_different_operand_types	Non-constant operands of bitwise operator differ in underlying type

AutosarC++17_10-M5.0.21

Bitwise operators shall only be applied to operands of unsigned underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitop_on_enum	Bitwise operator on enum underlying type
bitop_on_signed	Bitwise operator on signed underlying type

AutosarC++17_10-M5.2.1

Each operand of a logical && or || shall be a postfix-expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allowed	Can be used to name additional allowed IR node types for the operands.	None
enforcement		automated
level		required
require_postfix_epression	Whether postfix or primary expressions are required as operands.	True
target		implementation

Possible Messages

Name	Message
missing_parens_for_logical_operator_operand	Operand of && or must be parenthesized.

AutosarC++17_10-M5.2.2

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cannot_cast_virtual_base	Cannot convert pointer to base class {} to pointer to derived class {} -- base class is virtual
missing_dynamic_cast_on_virtual_base	Use dynamic_cast on virtual base class

AutosarC++17_10-M5.2.3

Casts from a base class to a derived class should not be performed on polymorphic types.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cast_from_polybase_to_derived	Cast from polymorphic base class to derived class

AutosarC++17_10-M5.2.6

A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[FunctionPointerTypes], [VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes, NullPointerTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_category	Selection of type categories to consider for type changes inside same category.	[FunctionPointerTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion of function pointer to other type
cast_changes_type_inside_category	Conversion of function pointer to other function pointer type

AutosarC++17_10-M5.2.8

An object with integer type or pointer to void type shall not be converted to an object with pointer type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, VoidPointerTypes], [ObjectPointerTypes, IncompletePointerTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, $(T^*)(\text{void}^*)x$ will not be reported.	True
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion from void* or integer to pointer type

AutosarC++17_10-M5.2.9

A cast should not convert a pointer type to an integral type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories. [[[VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes], [SignedTypes, UnsignedTypes, BoolTypes, EnumTypes, CharacterTypes]]]	
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion between pointer and integral type

AutosarC++17_10-M5.2.10

The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
forbid_all_operators	If True, forbids mixing with any kind of operator; else only with arithmetic operators.	False
level		required
target		implementation

Possible Messages

Name	Message
increment_mixed_with_operator	Increment or decrement mixed with other operators

AutosarC++17_10-M5.2.11

The comma operator, `&&` operator and the `||` operator shall not be overloaded.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
invalid	Selection of disallowed operator overloads by name.	<code>['operator&&', 'operator ', 'operator,']</code>
invalid_kinds	Selection of disallowed operator overloads by operator kind.	<code>[]</code>
level		required
target		implementation

Possible Messages

Name	Message
forbidden_operator_overload	Overloaded version of comma operator or <code>&&</code> or <code> </code>

AutosarC++17_10-M5.2.12

An identifier with array type passed as a function argument shall not decay to a pointer.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
decay	Array to pointer decay

AutosarC++17_10-M5.3.1

Each operand of the `!` operator, the logical `&&` or the logical `||` operators shall have type `bool`.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_in_conditions	If True, only logical operators inside conditions are checked.	False
target		implementation

Possible Messages

Name	Message
nonbool_logical_operator_operand	Operand of logical operator shall be of type bool

AutosarC++17_10-M5.3.2

The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_literals	If True, integer literals are also disallowed as operands.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unary_minus_on_unsigned	Unary minus applied to unsigned

AutosarC++17_10-M5.3.3

The unary & operator shall not be overloaded.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
invalid	Selection of disallowed operator overloads by name.	[]
invalid_kinds	Selection of disallowed operator overloads by operator kind.	[7]
level		required
target		implementation

Possible Messages

Name	Message
forbidden_operator_overload	Overloaded version of unary &

AutosarC++17_10-M5.3.4

Evaluation of the operand to the sizeof operator shall not contain side effects.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
side_effect_in_sizeof	Operand of "sizeof" shall not contain side effects

AutosarC++17_10-M5.8.1

The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_nonconst_cases	Whether a message should be printed for non-constant cases missing a range-checking if statement	False
target		implementation
use_essential_type	Whether essential type should be computed for allowed value range.	False

Possible Messages

Name	Message
excessive_shift	Right operand of shift operator has value {}, but should be in range 0 to {}
possible_excessive_shift	Right operand of shift operator should be checked to be in range 0 to {}

AutosarC++17_10-M5.14.1

The right-hand operand of a logical && or || operator shall not contain side effects.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_volatile_access	Whether access to a volatile variable should be reported as side effect.	False
assume_effect_when_undefined	Whether all calls to undefined (external) functions should be seen as having a side-effect.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
call_with_side_effect	Call in right-hand operand of && or may have side-effect
modifies_local_var	Right-hand operand of && or modifies '{}'
side_effect	Right-hand operand of && or has side-effect

AutosarC++17_10-M5.17.1

The semantic equivalence between a binary operator and its assignment operator form shall be preserved.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_call_relation	If True, checks whether there is a call relation between binary and assignment version.	True
enforcement		automated
ignore_stream_operators	If True, allows definitions of operator<<() and operator>>() without the corresponding assignment operator. Note this will also allow operator<<=() and operator>>=() as they no longer have an equivalent binary form.	False
level		required
target		implementation

Possible Messages

Name	Message
missing_assignment_version	Missing overload for corresponding assignment version of operator
missing_binary_version	Missing overload for corresponding binary version of operator
missing_call_to_assignment_version	There is no call relation between this operator and its assignment version to ensure semantic equivalence
missing_call_to_binary_version	There is no call relation between this operator and its binary version to ensure semantic equivalence

AutosarC++17_10-M5.18.1

The comma operator shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_comma	Use of comma sequence operator.

AutosarC++17_10-M5.19.1

Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
const_expression_overflow	Overflow during evaluation of constant expression.

AutosarC++17_10-M6.2.1

Assignment operators shall not be used in sub-expressions.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
assignment_result_used	Assignment inside sub-expression.

AutosarC++17_10-M6.2.2

Floating-point expressions shall not be directly or indirectly tested for equality or inequality.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
float_equality_comparison	Testing a float for (in)equality.
float_indirect_equality_comparison	Indirectly testing a float for (in)equality.

AutosarC++17_10-M6.2.3

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_empty_macros	Whether a macro invocation before the ; is allowed if it expands to nothing.	False
allow_nonempty_macros	Whether a non-empty macro invocation before the ; is allowed.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
null_statement_not_isolated	Null statement not on a line by itself

AutosarC++17_10-M6.3.1

The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
loop_body_not_compound	Loop body shall be a statement sequence.
switch_body_not_compound	switch body shall be a statement sequence.

AutosarC++17_10-M6.4.1

An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
dangerous_else	Block or if statement required as else branch.
dangerous_then	Block required as then branch.

AutosarC++17_10-M6.4.2

All if ... else if constructs shall be terminated with an else clause.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
defensive_else	Controls if the else branch of if-cascade's should contain a suitable comment or not be empty.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
non_defensive_else	Last else in the if-cascade must not be empty or contain a comment.
unterminated_if_cascade	Last if in the if-cascade has no else.

AutosarC++17_10-M6.4.3

A switch statement shall be a well-formed switch statement.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
acceptable_end_items	This set characterizes allowed last items at the end of a switch case	('Exit_Switch', 'Throw_Expression')
acceptable_middle_items	This set characterizes allowed last items in the middle of a switch case	('Exit_Switch', 'Nondefault_Case_Label', 'Throw_Expression')
allow_as_first	Whether to allow the "default" case as the first case in a switch.	False
allow_complete_enum	Whether a switch having a case for all enumerators is tolerated without a default.	True
allow_empty_statements	Whether empty statements at the end of switch-case should be ignored.	False
allow_fallthrough_comment	Whether a comment with text "fall" and "through" is allowed to mark fallthrough.	True
allow_fallthrough_macro	Whether a macro called __fallthrough is allowed to mark fallthrough.	True
allow_if	Whether an if containing acceptable items at the end of both branches is accepted.	False
allow_noreturn_function_calls	Whether a function call to a [[noreturn]] function is considered a valid termination.	False
comment	The comment that indicates an intentional fallthrough. Can be a list of comments.	[]
disallow_multiple_compound_statements	Whether there can be more than one toplevel compound statement.	True
disallowed_nested_statements	Statement types not allowed inside compound statements within case-clause body.	None
disallowed_toplevel_statements	Statement types not allowed in case-clause body on toplevel.	('Named_Label', 'Unconditional_Branch', 'Declaration_Statement')
enforcement		automated
level		required
minimum_switch_cases	The number of cases a switch statement should at least have.	1
target		implementation

Possible Messages

Name	Message
case_label_nested	A switch label shall only be used in a switch.
default_not_last	"default" clause is not last in switch.
empty_default	Default clause should contain a statement or comment.
empty_switch	Switch has no "case" clause.
forbidden_compound_statement_in_case	At most one compound statement is allowed on toplevel in case-clauses.
forbidden_nested_statement_in_case	Statement is not allowed inside compound statement in case-clauses.
forbidden_toplevel_statement_in_case	Statement is not allowed outside a compound statement in case-clauses.
minimum_switch_cases	Switch has too little "case" clauses.
missing_default	Switch has no "default" clause.
switch_body_not_compound	A switch statement shall be a well-formed switch statement.
switch_not_well_formed	Switch has a statement before the first case label.
switch_over_bool	Switch expression is effectively Boolean.
unterminated_case	This switch-case is not terminated with {}, last statement is {}.
unterminated_empty_case	This switch-case is empty, not terminated with {}.

AutosarC++17_10-M6.4.4

A switch label shall only be used when the most closely-enclosing compound-statement is the body of a switch-statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
case_label_nested	A switch label shall only be used in a switch.

AutosarC++17_10-M6.4.5

An unconditional throw or break statement shall terminate every non-empty switch-clause.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
acceptable_end_items	This set characterizes allowed last items at the end of a switch case	('Exit_Switch', 'Throw_Expression')
acceptable_middle_items	This set characterizes allowed last items in the middle of a switch case	('Exit_Switch', 'Nondefault_Case_Label', 'Throw_Expression')
allow_empty_statements	Whether empty statements at the end of switch-case should be ignored.	False
allow_fallthrough_comment	Whether a comment with text "fall" and "through" is allowed to mark fallthrough.	True
allow_fallthrough_macro	Whether a macro called __fallthrough is allowed to mark fallthrough.	True
allow_if	Whether an if containing acceptable items at the end of both branches is accepted.	False
allow_noreturn_function_calls	Whether a function call to a [[noreturn]] function is considered a valid termination.	False
comment	The comment that indicates an intentional fallthrough. Can be a list of comments.	[]
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unterminated_case	This switch-case is not terminated with {}, last statement is {}.
unterminated_empty_case	This switch-case is empty, not terminated with {}.

AutosarC++17_10-M6.4.6

The final clause of a switch statement shall be the default clause.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_as_first	Whether to allow the "default" case as the first case in a switch.	False
allow_complete_enum	Whether a switch having a case for all enumerators is tolerated without a default.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
default_not_last	"default" clause is not last in switch.
empty_default	Default clause should contain a statement or comment.
missing_default	Switch has no "default" clause.

AutosarC++17_10-M6.4.7

The condition of a switch statement shall not have bool type.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
switch_over_bool	Switch condition shall not have bool type.

AutosarC++17_10-M6.5.2

If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
stepping_loop_uses_equality_check	Loop-counter shall not be tested with equality operator if not modified by -- or ++

AutosarC++17_10-M6.5.3

The loop-counter shall not be modified within condition or statement.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
loop_counter_modified_in_condition	Loop-counter shall not be modified within condition
modified_loop_counter	Loop-counter shall not be modified within loop body

AutosarC++17_10-M6.5.4

The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_constexpr	Allow constexpr as a constant <n>.	True
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
nonconst_loop_increment	Loop-counter shall be modified by one of: --, ++, -=n, or +=n (with constant n)

AutosarC++17_10-M6.5.5

A loop-control-variable other than the loop-counter shall not be modified within condition or expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
modified_loop_control_variable	Loop-control variable (other than counter) shall not be modified within condition or expression

AutosarC++17_10-M6.5.6

A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
nonbool_loop_control_variable	Loop-control variable (other than counter) shall have type bool

AutosarC++17_10-M6.6.1

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
goto_label_block	Label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

AutosarC++17_10-M6.6.2

The goto statement shall jump to a label declared later in the same function body.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
backwards_goto	Label referenced by a goto statement shall be declared later in same function.

AutosarC++17_10-M6.6.3

The continue statement shall only be used within a well-formed for loop.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
report_outside_for_loops	Whether to report continue statements in do..while/while loops	True
target		implementation

Possible Messages

Name	Message
continue_in_bad_loop	The continue statement shall only be used within a well-formed for loop

AutosarC++17_10-M7.1.2

A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_all_variables	Whether non-parameter variables should be checked as well.	False
enforcement		automated
ignore_forwarding_references	Whether to allow C++11 forwarding references (universal references) even if all template instances would also work with a const lvalue reference.	True
ignore_move_constructors	Whether to allow non-const rvalue references on constructors.	True
level		required
report_overrides	Whether to report violations on overrides. For virtual methods, this rule will report a violation only if the parameter can be marked const in all related methods in the inheritance hierarchy. Thus, the same violation is reported for each method in the inheritance hierarchy. You can set this option to False to report such violations only for the base methods.	True
target		implementation

Possible Messages

Name	Message
parameter_can_point_to_const	{ } can be declared as pointer/reference to const.

AutosarC++17_10-M7.3.1

The global namespace shall only contain main, namespace declarations and extern "C" declarations.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
global_symbol	Symbol not allowed in global namespace.

AutosarC++17_10-M7.3.2

The identifier main shall not be used for a function other than the global function main.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
nonglobal_function_named_main	The identifier main shall not be used for a function other than the global function main

AutosarC++17_10-M7.3.3

There shall be no unnamed namespaces in header files.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unnamed_namespace_in_header	Unnamed namespaces in header file

AutosarC++17_10-M7.3.4

Using-directives shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
using_directive	Using-directives shall not be used

AutosarC++17_10-M7.3.5

Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_other_usages	Whether to find other usages (such as function calls, or taking a fp-reference) between multiple declarations for an identifier.	False
target		implementation

Possible Messages

Name	Message
declarations_surrounding_usage	Declarations straddle a usage
declarations_surrounding_using	Declarations straddle a using-declaration

AutosarC++17_10-M7.3.6

using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
using_declaration_in_header	Using-declaration in header file
using_namespace_in_header	Using-directive in header file

AutosarC++17_10-M7.4.1

All usage of assembler shall be documented.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_assembler	Usage of assembler shall be documented

AutosarC++17_10-M7.4.2

Assembler instructions shall only be introduced using the asm declaration.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pragma_asm	Assembler instructions shall only be introduced using the asm declaration

AutosarC++17_10-M7.4.3

Assembly language shall be encapsulated and isolated.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
asm_not_encapsulated	Assembly language shall be encapsulated and isolated.

AutosarC++17_10-M7.5.1

A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
global_reference_to_local_var	Address of local variable escapes via global variable.
parameter_reference_to_local_var	Address of local variable escapes via parameter.
returning_reference_to_local_var	Returning reference/pointer to local variable.

AutosarC++17_10-M7.5.2

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
allow_longer_living_local	Whether assignment to a longer-living local variable should be accepted.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
possibly_leaking_reference_to_local_variable	Address of local variable is assigned to longer-living object.

AutosarC++17_10-M8.0.1

An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
allow_in_for_loop_init	Whether a multi-declaration is allowed in the for-init statement	False
allow_uninitialized_simple_type	Whether a multi-declaration of simple type is allowed when there is no initialization	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
multi_declaration	Multiple declarators in single declaration

AutosarC++17_10-M8.3.1

Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
redefinition_uses_different_default_argument	Default argument differs from the one in redefined method

AutosarC++17_10-M8.4.2

The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
ignore_casing	Whether to ignore casing for the check.	False
include_redefinitions	Whether declarations of redefining methods should be considered together with the redefined ones.	True
inspect_only_defined_routines	If False, re-declarations of undefined functions are compared to earlier declarations as well.	False
level		required
target		implementation

Possible Messages

Name	Message
parameter_name_mismatch	Different name used for parameter

AutosarC++17_10-M8.4.4

A function identifier shall either be used to call the function or it shall be preceded by &.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allowed	Qualified names of functions of which it is allowed to take the address implicitly, e.g. C++ I/O manipulators	['std::endl', 'std::flush', 'std::boolalpha', 'std::noboolalpha', 'std::showbase', 'std::noshowbase', 'std::showpoint', 'std::noshowpoint', 'std::showpos', 'std::noshowpos', 'std::skipws', 'std::noskipws', 'std::uppercase', 'std::nouppercase', 'std::unitbuf', 'std::nounitbuf', 'std::internal', 'std::left', 'std::right', 'std::dec', 'std::hex', 'std::oct', 'std::fixed', 'std::scientific', 'std::hexfloat', 'std::defaultfloat', 'std::ws', 'std::ends', 'std::resetiosflag', 'std::setiosflag', 'std::setbase', 'std::setfill', 'std::setprecision', 'std::setw', 'std::get_money', 'std::put_money', 'std::get_time', 'std::put_time', 'std::quoted']
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
implicit_routine_address	Taking address of function without &

AutosarC++17_10-M8.5.1

All variables shall have a defined value before they are used.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
init_functions	Names of functions to be inspected as well when called directly from constructor.	('Init', 'init')
inspect_directly_called_methods	Inspect all methods directly called from constructor.	False
level		required
only_member_initializer_list	Only inspect member initializer list and not the constructor body/methods.	False
perform_global_analysis	Whether global analysis (using iranalysis) should be done, or just a local analysis.	True
report_missing_base_constructors	Enables detection of constructors which rely on implicit base constructor calls.	False
report_missing_field_constructors	Enables detection of constructors which rely on implicit field constructor calls.	False
target		implementation

Possible Messages

Name	Message
implicit_field_init	Field is only implicitly initialized in constructor.
missing_base_class_init	Base class is not explicitly initialized in constructor.
missing_field_init	Field is not initialized in constructor.
pass_as_pointer_to_const_param	Passing uninitialized variable by pointer as function parameter with pointer-to-const type
possible_uninit	Use of possibly uninitialized variable
uninit	Use of uninitialized variable

AutosarC++17_10-M8.5.2

Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_any_zero_initialization	Whether {0} is allowed in nested aggregates as well.	False
allow_toplevel_zero_initialization	Whether {0} is allowed as top-level initialization for the complete aggregate	True
allowed_classes_without_braces	For the listed class types, don't require braces. This is useful for classes that transparently wrap a single contained value. For example, with this option "std::array<int, 3> a = {1,2,3};" is OK even though there could be two sets of braces (one for "std::array", the other and for its inner "int[3]" array).	['std::array']
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
incomplete_aggregate_initialization	Provide explicit initializer for each part of initialized entity
missing_aggregate_initialization_braces	Use braces to indicate structure of initialized entity

AutosarC++17_10-M9.3.1

const member functions shall not return non-const pointers or references to class-data.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
include_copies_of_pointer_fields	Whether returning a pointer field should be reported as well.	True
inspect_only_const_methods	Whether all methods or only const methods should be checked.	True
level		required
only_report_references	Whether pointer and reference to field should be reported, or just references.	False
smart_pointer_names	Names that are considered smart pointers.	['unique_ptr', 'shared_ptr', 'weak_ptr']
target		implementation

Possible Messages

Name	Message
returning_nonconst_member_reference	Returning non-const reference/pointer to class data.

AutosarC++17_10-M9.3.3

If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_candidates_for_const	If False, avoid reporting methods that can be made const.	True
report_candidates_for_static	If False, avoid reporting methods that can be made static.	True
target		implementation
test_operators_for_static	If True, check whether a method can be made static is also applied to operator methods	False

Possible Messages

Name	Message
method_can_be_const	Method can be declared const.
method_can_be_static	Method can be declared static.

AutosarC++17_10-M9.6.1

When the absolute positioning of bits representing a bit-field is required, then the behaviour and packing of bit-fields shall be documented.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitfield	Usage of bit-fields shall be documented

AutosarC++17_10-M10.1.1

Classes should not be derived from virtual bases.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
virtual_inheritance	Classes should not be derived from virtual bases.

AutosarC++17_10-M10.1.2

A base class shall only be declared virtual if it is used in a diamond hierarchy.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
virtual_inheritance_outside_diamond	A base class shall only be declared virtual if it is used in a diamond hierarchy.

AutosarC++17_10-M10.1.3

An accessible base class shall not be both virtual and non-virtual in the same hierarchy.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
base_class_being_virtual_and_nonvirtual	Has base class which is both virtual and non-virtual.

AutosarC++17_10-M10.2.1

All accessible entity names within a multiple inheritance hierarchy should be unique.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
ambiguous_member	All accessible entity names within a multiple inheritance hierarchy should be unique
use_of_ambiguous_name	{ } is ambiguous

AutosarC++17_10-M10.3.3

A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pure_redefinition	Pure redefinition of non-pure virtual function.

AutosarC++17_10-M11.0.1

Member data in non-POD class types shall be private.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_protected_members	If True, protected members are tolerated.	False
allowed	Specifies allowed fields as pairs [class name pattern, field name pattern]. Example: [re.compile('.*'), re.compile('x')] to allow x in all classes.	[]
enforcement		automated
ignore_const_members	If True, non-private const members are tolerated.	False
ignore_pod	Whether fields in POD classes should be reported.	True
ignore_structs	Whether fields in structs should be reported.	False
ignore_templates	Whether fields in generic templates should be reported.	True
level		required
target		implementation

Possible Messages

Name	Message
protected_field	Member data in non-POD class types shall be private.
public_field	Member data in non-POD class types shall be private.

AutosarC++17_10-M12.1.1

An object's dynamic type shall not be used from the body of its constructor or destructor.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
constructor_using_dynamic_cast	Dynamic cast used in constructor/destructor.
constructor_using_typeid	Typeid on polymorphic class used in constructor/destructor.
constructor_using_virtual_call	Virtual call used in constructor/destructor.

AutosarC++17_10-M14.5.2

A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_copy_constructor_for_template	Class has template constructor but no copy constructor

AutosarC++17_10-M14.5.3

A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_copy_asgn_for_template	Class has template assignment operator but no copy assignment operator

AutosarC++17_10-M14.6.1

In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unqualified_dependent_member_access	Use qualifiers or this-> to select name that may be found in that dependent base

AutosarC++17_10-M14.7.3

All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_in_type_declaration_file	Also allow specializations in the same file of a user-defined type for which the specialization is declared.	False
enforcement		automated
level		required
relax_if_template_only_declared	Allow specializations that are not declared in the header file if the template itself is only declared but not defined in the header.	False
target		implementation

Possible Messages

Name	Message
template_specialization_in_different_file	Specialization not declared in same file as primary template

AutosarC++17_10-M14.8.1

Overloaded function templates shall not be explicitly specialized.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
specialization_of_overloaded_template	Overloaded function templates shall not be explicitly specialized

AutosarC++17_10-M15.0.3

Control shall not be transferred into a try or catch block using a goto or a switch statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
goto_into_try	Goto jumps into try or catch block
switch_into_try	Switch statement jumps into try or catch block

AutosarC++17_10-M15.1.1

The assignment-expression of a throw statement shall not itself cause an exception to be thrown.

Input: IR
Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throw_expression_raises_exception	Expression of throw may itself raise an exception

AutosarC++17_10-M15.1.2

NULL shall not be thrown explicitly.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throwing_null	NULL shall not be thrown explicitly

AutosarC++17_10-M15.1.3

An empty throw (throw;) shall only be used in the compound-statement of a catch handler.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
rethrow_outsideCatch	Rethrow outside any catch block

AutosarC++17_10-M15.3.1

Exceptions shall be raised only after start-up and before termination of the program.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_only_one_exception_per_function	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
report_only_uncaught	Whether the check shall report all throws or just those not caught.	False
target		implementation

Possible Messages

Name	Message
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionRaisedInInitialization	Exception raised in initialization or finalization

AutosarC++17_10-M15.3.3

Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
handlerUsesField	Handler of a function-try-block shall not reference non-static members from this class or its bases

AutosarC++17_10-M15.3.4

Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
inspect_at_exit_handlers	Whether to also inspect at_exit() handlers-functions.	True
inspect_thread_main	Whether to also inspect thread main functions.	True
level		required
report_only_one_exception_per_function	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
target		implementation

Possible Messages

Name	Message
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionEscapingMain	Uncaught exception escaping from main or additional entry point

AutosarC++17_10-M15.3.6

Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
wrongCatchOrder	Catch handlers in wrong order.

AutosarC++17_10-M15.3.7

Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
catchAllNotLast	Catch-all shall occur as last handler.

AutosarC++17_10-M16.0.1

#include directives in a file shall only be preceded by other preprocessor directives or comments.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_extern_blocks	Whether to allow includes in `extern` blocks and not report a violation.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
include_after_code	#include should only be preceded by comments or preprocessor directives

AutosarC++17_10-M16.0.2

Macros shall only be #define'd or #undef'd in the global namespace.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
local_macro	#define or #undef not in global namespace

AutosarC++17_10-M16.0.5

Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
directives	Preprocessing directives to check for in macro arguments.	(`#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, `#endif`, `#pragma`, `#warning`, `#error`, `#line`, `#include`, `#include_next`, `#ident`, `#region`, `#endregion`, `#asm`, `#endasm`, `#define`, `#undef`)
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pp_directive_as_macro_arg	Preprocessing directive used in macro argument.

AutosarC++17_10-M16.0.6

In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unsafe_macro_param	Macro parameter not enclosed in parentheses

AutosarC++17_10-M16.0.7

Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
undefined_macro_used	Zero used for undefined preprocessing identifier

AutosarC++17_10-M16.0.8

If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
malformed_pp_directive	Malformed or unknown preprocessing directive

AutosarC++17_10-M16.1.1

The defined preprocessor operator shall only be used in one of the two standard forms.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
nonstandard_defined	Non-standard use of defined operator

AutosarC++17_10-M16.1.2

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_endif	The #endif for this directive is missing
missing_if	The #if for this directive is missing

AutosarC++17_10-M16.2.3

Include guards shall be provided.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
macro_name_restrictions	Python iterable of functions with parameters (file, define, macro) to perform additional checks on the macro used as include guard in the given file. Should return None if macro is accepted, else a message to print for the violation. The parameters are: PIR File node for the header, token for the macro #define or #pragma once, and the name of the macro or the string 'pragma once'.	None
target		implementation

Possible Messages

Name	Message
include_guard_missing	Missing include guard.
include_guard_partial	Include guard does not cover complete file.
include_guard_with_wrong_define	Bad include guard: #define uses wrong macro, should be {}.
include_guard_without_define	Incomplete include guard: missing #define {}.

AutosarC++17_10-M16.3.1

There shall be at most one occurrence of the # or ## operators in a single macro definition.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_only_concatenated_stringized	If True, only macros where a parameter is stringized and the result used for token concatenation are reported.	False
target		implementation

Possible Messages

Name	Message
concatenated_stringized	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator
more_than_one_macro_hash	Only one # or ## allowed in macro definition

AutosarC++17_10-M16.3.2

The # and ## operators should not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
hash_in_macro	The # and ## operators should not be used.

AutosarC++17_10-M17.0.2

The names of standard library macros and objects shall not be reused.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_locals	Whether parameters and local variables should also be checked	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
reused_macro_object_libname	The names of standard library macros and objects shall not be reused.

AutosarC++17_10-M17.0.3

The names of standard library functions shall not be overridden.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_function_declarations	Whether nondefining function declarations with library names are allowed	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
reused_routine_libname	The names of standard library functions shall not be overridden.

AutosarC++17_10-M17.0.5

The setjmp macro and the longjmp function shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	setjmp
symbols	Names of symbols which are forbidden.	['setjmp', 'longjmp']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++17_10-M18.0.3

The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	stdlib
symbols	Names of symbols which are forbidden.	['abort', 'exit', 'getenv', 'system']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib{.h} -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++17_10-M18.0.4

The time handling functions of library <ctime> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	time
symbols	Names of symbols which are forbidden.	['clock', 'difftime', 'mktime', 'time', 'asctime', 'ctime', 'gmtime', 'localtime', 'strftime']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib{.h} -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++17_10-M18.0.5

The unbounded functions of library <cstring> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	string
symbols	Names of symbols which are forbidden.	['strcpy', 'strcmp', 'strcat', 'strchr', 'strspn', 'strcspn', 'strpbrk', 'strrchr', 'strstr', 'strtok', 'strlen']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h} -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++17_10-M18.2.1

The macro offsetof shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	stddef
symbols	Names of symbols which are forbidden.	['offsetof']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h} -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++17_10-M18.7.1

The signal handling facilities of <csignal> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header (used for symbols like NULL and size_t which are defined in multiple headers)	['NULL', 'size_t']
enforcement		automated
header	Name of the header file which should not be used.	signal
level		required
target		implementation

Possible Messages

Name	Message
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.

AutosarC++17_10-M19.3.1

The error indicator errno shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	['errno', 'stdlib', 'stddef']
symbols	Names of symbols which are forbidden.	['errno']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <cerrno>.

AutosarC++17_10-M27.0.1

The stream input/output library <cstdio> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header (used for symbols like NULL and size_t which are defined in multiple headers)	['NULL', 'size_t']
enforcement		automated
header	Name of the header file which should not be used.	stdio
level		required
target		implementation

Possible Messages

Name	Message
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.

Rules in Group AutosarC++18_03

AutosarC++18_03-A0.1.1

A project shall not contain instances of non-volatile variables being given values that are never subsequently used.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_dead_initializations	Whether initialisations may be reported as dead.	True
target		implementation

Possible Messages

Name	Message
conditional_unused_def	Result of assignment is not used along some path(s)
init_used_in_other_isr	Initialization is only used in some interrupt handler
unused_def	Result of assignment is not used
unused_init	Unused initialization
used_in_other_isr	Result of assignment is only used in some interrupt handler

AutosarC++18_03-A0.1.2

The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed_functions	Calls to these functions are ignored.	frozenset(['strncat', 'memmove', 'memset', 'strcat', 'strcpy', 'memcpy', 'strncpy'])
enforcement		automated
inspect_template_instances	Whether calls in template instances should be reported.	False
level		required
target		implementation

Possible Messages

Name	Message
discarded_return_with_entity	Return value of function discarded.
discarded_return_without_entity	Return value of function discarded.

AutosarC++18_03-A0.1.3

Every defined function shall be called at least once.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_copy_assignment_operators	Whether uncalled copy assignment operators are allowed. Other rules like the "rule of three" might require copy assignment operators as part of good class design, even if there are currently no callers of the copy assignment operator.	False
ignore_copy_constructors	Whether uncalled copy constructors are allowed. Other rules like the "rule of three" might require copy constructors as part of good class design, even if there are currently no callers of the copy constructor.	False
level		required
only_check_internal_or_private_methods	Whether only function with internal linkage or private functions should be checked.	True
only_check_static_or_private_methods	Whether only static functions or private functions should be checked.	False
only_check_unit_locals	Whether only global static functions should be checked.	False
target		implementation

Possible Messages

Name	Message
defined_function_not_called	Defined function is not called

AutosarC++18_03-A0.1.4

There shall be no unused named parameters in non-virtual functions.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
inspect_nonvirtual_functions	Whether parameters of non-virtual functions should be checked.	True
inspect_virtual_functions	Whether virtual functions should be checked.	False
level		required
report_unnamed_parameters	Whether unnamed unused parameters should be reported as well.	False
target		implementation

Possible Messages

Name	Message
unused_parameter	Unused named parameter of non-virtual function

AutosarC++18_03-A0.1.5

There shall be no unused named parameters in the set of parameters for a virtual function and all the functions that override it.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
inspect_nonvirtual_functions	Whether non-virtual functions should be checked.	False
inspect_virtual_functions	Whether parameters of virtual functions should be checked. Violations will be reported in the base class when none of the derived classes make use of the parameter.	True
level		required
report_unnamed_parameters	Whether unnamed unused parameters should be reported as well.	False
target		implementation

Possible Messages

Name	Message
unused_parameter	Unused named parameter of virtual function

AutosarC++18_03-A0.1.6

A project shall not contain unused type declarations.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unused_type	Unused type declaration

AutosarC++18_03-A0.4.2

Type long double shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
forbidden_types	The ir.Logical Types not to be used e.g. `C_Long_Double_Type`.	['C_Long_Double_Type']
level		required
target		implementation

Possible Messages

Name	Message
type_used	The type shall not be used.

AutosarC++18_03-A1.1.1

All code shall conform to ISO/IEC 14882:2014 - Programming Language C++ and shall not use deprecated features.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	The set of messages regarding syntax and constraint violations.	set([2464, 2465, 1444, 2567, 2381, 2221, 1909, 1215])
reported_severities	List of severities to display.	('error', 'warning')
show_include_path	If True, violations regarding nested includes print the #include path.	False
standards	List of allowed frontend command-line options to select language standard.	['--strict', '-A']
target		implementation
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

Possible Messages

Name	Message
access_control	Number of access control declarations is higher than translation limit (which is {})
at_quick_exit_functions	Number of functions registered with at_quick_exit() is higher than translation limit (which is {})
atexit_functions	Number of functions registered with atexit() is higher than translation limit (which is {})
base_classes	Number of direct and indirect base classes is higher than translation limit (which is {})
call_arguments	Number of arguments is higher than translation limit (which is {})
catch_handlers	Number of catch handlers is higher than translation limit (which is {})
chars_in_line	Number of characters in logical line is higher than translation limit (which is {})
chars_in_string	Number of characters in string is higher than translation limit (which is {})
cpp_members	Number of members is higher than translation limit (which is {})

data_members	Number of data members is higher than translation limit (which is {})
direct_bases	Number of direct base classes is higher than translation limit (which is {})
enumerators	Number of enumerators is higher than translation limit (which is {})
expr_in_core_constant	Number of full expressions evaluated within a core constant expression is higher than translation limit (which is {})
external_identifiers	Number of external identifiers in this unit is higher than translation limit (which is {})
final_virtual_functions	Number of final overriding virtual functions is higher than translation limit (which is {})
friends	Number of friend declarations is higher than translation limit (which is {})
function_parameters	Number of parameters is higher than translation limit (which is {})
macro_arguments	Number of macro arguments is higher than translation limit (which is {})
macro_parameters	Number of macro parameters is higher than translation limit (which is {})
macros	Number of macro definitions in this unit is higher than translation limit (which is {})
member_initializers	Number of member initializers is higher than translation limit (which is {})
nested_blocks	Number of nested blocks is higher than translation limit (which is {})
nested_composites	Nesting level of structs and unions is higher than translation limit (which is {})
nested_declarators	Number of pointer/array/function declarators is higher than translation limit (which is {})
nested_externals	Number of nested external specifications is higher than translation limit (which is {})
nested_includes	Nesting level of #includes is higher than translation limit (which is {})
nested_preprocessor_ifs	Nesting level of #if is higher than translation limit (which is {})
nested_template_instantiations	Number of nested template instantiations is higher than translation limit (which is {})
object_size	Number of bytes in object is higher than translation limit (which is {})
parenthesized_declarators	Nesting level of parenthesized declarators is higher than translation limit (which is {})
parenthesized_expressions	Nesting level of parenthesized subexpressions is higher than translation limit (which is {})
placeholders	Number of placeholders is higher than translation limit (which is {})
recursive_constexpr	Number of recursive constexpr function invocations is higher than translation limit (which is {})
scope_qualifications	Number of scope qualifications is higher than translation limit (which is {})
significant_chars_external	Number of significant initial characters for external identifier is higher than translation limit (which is {})
significant_chars_internal	Number of significant initial characters for internal identifier is higher than translation limit (which is {})
static_members	Number of static members is higher than translation limit (which is {})
std_compile	You have to compile with option {}
switch_cases	Number of cases is higher than translation limit (which is {})
template_params	Number of template parameters is higher than translation limit (which is {})
throw_spec	Number of throw specifications is higher than translation limit (which is {})
variables_in_block	Number of variables in block is higher than translation limit (which is {})
virtual_bases	Number of virtual base classes is higher than translation limit (which is {})

AutosarC++18_03-A1.4.3

All code should compile free of compiler warnings.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	If provided, only messages of these types are reported.	None
reported_severities	List of severities to display.	('error', 'warning')
target		implementation
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	False

AutosarC++18_03-A2.3.1

Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_comments	Whether to allow characters not in the basic source character set in comments.	False
allow_in_unicode_strings	Whether to allow characters not in the basic source character set in unicode string literals.	True
allow_in_wide_strings	Whether to allow characters not in the basic source character set in wide string literals.	True
basic_characters_list	The basic character list as per rule.	\\\\\\\ abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ#~!@%^&_=+=`~`@
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
forbidden_characters	Forbidden characters used.

AutosarC++18_03-A2.5.1

Trigraphs shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_comments	Allow the use of trigraphs in comments	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
trigraph_use	Use of trigraph.

AutosarC++18_03-A2.5.2

Digraphs should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
digraph_use	Digraph used.

AutosarC++18_03-A2.7.1

The character \ shall not occur as a last character of a C++ comment.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
line_splicing_in_cpp_comment	Line-splicing shall not be used in // comments.

AutosarC++18_03-A2.7.2

Sections of code shall not be "commented out".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
files_to_check	Files to be checked, e.g. Primary_File or File.	File
level		required
normal_text	Tokens that indicate non-code comments. This is the first step of checking a comment. If any of these words exist in the comment text, it is regarded as not containing code.	{'and', 'or', 'but', 'now', 'to', 'is', 'are', 'only', 'be', 'has', 'the', 'with', 'because', 'when', 'oder', 'und', '//', '#', 'AXIVION', '++++', '---', '===='}
normal_text_regex	Regular expression to match normal non-code text. This is the second step of checking a comment. If this regex matches the comment text, the comment is regarded as not containing code.	\b[\w\d_]+\s+[\w\d_]+\s+[\w\d_]+\b
suspicious	Tokens that indicate code in comments. This is the final step of checking a comment. If the first two steps did not exclude the comment from further inspection, the comment text is searched for any of these suspicious tokens. The comment is regarded as containing code, if any suspicious token is found.	{'=', '==', '>=', '<=', '[', ']', ':', '->', '->*', '/*', 'if', 'while', 'for', 'if (', 'while (', 'for (', '#pragma', '#else', '#endif', '#if', '#include', '++', '--')}
target		implementation

Possible Messages

Name	Message
code_in_c_comment	Sections of code shall not be commented out using C-style comments.
code_in_cpp_comment	Sections of code should not be commented out using C++ comments.

AutosarC++18_03-A2.7.3

All declarations of "user-defined" types, static and non-static data members, functions and methods shall be preceded by documentation.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_inherited	If True, a definition does not need documentation, if a corresponding declaration is documented.	False
allow_missing_documentation_on_private	If True, a class-member definition does not need documentation, if it is `private`.	False
allow_missing_documentation_on_protected	If True, a class-member definition does not need documentation, if it is `protected`.	False
doxygen_start	Start of a valid Doxygen comment.	{'/**', '///'}
enforcement		automated
ignore_defaulted	If True, defaulted function declarations are not checked for comments.	False
ignore_deleted	If True, deleted function declarations are not checked for comments.	False
ignore_redefinitions	If True, method redefinitions are not checked as they can 'inherit' the comment from the redefined method.	False
ignore_tool_comments	An optional compiled regular expression. Comments where this regex finds a matching substring are ignored in the search for a doxygen comment (e.g. control-comments of other tools).	None
level		required
node_types	IR node types to check for preceding Doxygen comment.	{'Routine_Definition', 'Routine_Declaration', 'Named_Type_Definition', 'Field_Definition'}
target		implementation

Possible Messages

Name	Message
missing_doxygen_comment_before_def	No Doxygen comment before declaration.

AutosarC++18_03-A2.8.1

A header file name should reflect the logical entity for which it provides declarations.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
file_type	Type of source_file file to examine.	User_Include_File
junk_characters	Characters that should not be considered in the filename to type similarity.	_-::
level		required
named_node_types	Physical IR node types to inspect for logical names to compare against the file name	{'Definition', 'No_Def_Declaration'}
similarity_threshold	Ratio of how similar the names need to be.	0.9
target		architecture/design/implementation

Possible Messages

Name	Message
source_file_name_not_type	The header should be named as a type it declares.

AutosarC++18_03-A2.8.2

An implementation file name should reflect the logical entity for which it provides definitions.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
file_type	Type of source_file file to examine.	Primary_File
junk_characters	Characters that should not be considered in the filename to type similarity.	_-::
level		advisory
named_node_types	Physical IR node types to inspect for logical names to compare against the file name	('Definition', 'No_Def_Declaration')
similarity_threshold	Ratio of how similar the names need to be.	0.9
target		architecture / design / implementation

Possible Messages

Name	Message
source_file_name_not_type	The file should be named as a type it declares.

AutosarC++18_03-A2.10.1

Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_classes	Whether identifiers in local classes can hide surrounding ones	True
allow_in_lambdas	Whether identifiers in lambdas can hide surrounding ones	True
allow_in_local_classes	Whether identifiers in local classes can hide surrounding ones	False
allow_in_namespaces	Whether identifiers in namespaces can hide surrounding ones	True
allow_parameter_in_function_declaration	Whether identifiers of parameters in function declarations may hide surrounding ones, given the function declaration uses a different identifier for the same parameter that does not hide a surrounding one.	False
check_inheritance_hiding	Whether to check for hiding by inheritance.	False
check_nested_classes	Whether symbols from inner classes should be reported as hiding symbols of outer classes.	False
distinguish_name_spaces	Whether C name spaces should be distinguished.	False
enforcement		automated
hiding_entities	LIR node types to consider symbols that hide other symbols.	{'Variable', 'Typedef_Type', 'Field', 'Parameter'}
level		required
maxlen	Number of significant characters (or None)	None
target		architecture/design/implementation
tolerate_constructor_parameters_hiding_fields	Allow constructor parameters to hide fields	False
tolerate_function_prototype_argument_hiding	Whether to tolerate arguments of a function prototype, that itself is a function argument, that have the same identifier as a argument of the outer function. If 'True' is given, the rule will still report cases, where argument identifiers match up and the function prototype refers to arguments of the outer function (e.g. by `decltype()`).	False
tolerate_macro_local_identifier	Whether an identifier being local to a macro is allowed to hide identifiers from outside the macro	False
unchecked_types	Deprecated legacy option, kept to avoid breaking old configurations.	{'Routine_Call', 'Stack_Object_Definition'}

Possible Messages

Name	Message
hiding	{ } hides { }

AutosarC++18_03-A2.10.4

The identifier name of a non-member object or function with static storage duration should not be reused.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_static_no_linkage	Whether to allow same identifiers for static objects without linkage (e.g. function local static objects)	False
enforcement		automated
func_filter	Restricts which functions are considered.	None
level		required
target		implementation
use_unqualified_names	If set to True, ns1::Type is considered equal to ns2::Type.	False
var_filter	Restricts which variables are considered.	None

Possible Messages

Name	Message
reused_global_function_name	Name of object or function with static storage duration reused.
reused_global_variable_name	Name of object or function with static storage duration reused.

AutosarC++18_03-A2.10.5

The identifier name of a non-member object or function with static storage duration should not be reused.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_static_no_linkage	Whether to allow same identifiers for static objects without linkage (e.g. function local static objects)	True
enforcement		automated
func_filter	Restricts which functions are considered.	None
level		required
target		design/architecture
use_unqualified_names	If set to True, ns1::Type is considered equal to ns2::Type.	True
var_filter	Restricts which variables are considered.	None

Possible Messages

Name	Message
reused_global_function_name	Name of object or function with static storage duration reused.
reused_global_variable_name	Name of object or function with static storage duration reused.

AutosarC++18_03-A2.10.6

A class or enumeration name shall not be hidden by a variable, function or enumerator declaration in the same scope.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
invert_findings	Whether to invert primary and secondary SLocs for violations	True
level		required
target		implementation

Possible Messages

Name	Message
reused_type	Type should not be hidden by a name in the same scope.

AutosarC++18_03-A2.11.1

Volatile keyword shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		design / implementation

Possible Messages

Name	Message
volatile_qualifier	The volatile type qualifier shall not be used

AutosarC++18_03-A2.13.1

Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed	List of allowed characters after backslash.	"\"?\\abfnrtv
enforcement		automated
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
nonstandard_escape_sequence	Use of non-standard escape sequence.

AutosarC++18_03-A2.13.2

Narrow and wide string literals shall not be concatenated.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
cpp11_mode	Use rules as defined in the C++11 standard.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
mixed_string_concatenation	Concatenation of mixed string encodings
narrow_wide_concat	Concatenation of narrow and wide string literal

AutosarC++18_03-A2.13.3

Type wchar_t shall not be used.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
forbidden_types	The ir.Logical Types not to be used e.g. `C_Long_Double_Type`.	['CPP_WChar_Type']
level		required
target		architecture/design/implementation/implementation

Possible Messages

Name	Message
type_used	The type shall not be used.

AutosarC++18_03-A2.13.4

String literals shall not be assigned to non-constant pointers.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	If provided, only messages of these types are reported.	[2464]
reported_severities	List of severities to display.	('error', 'warning', 'remark')
target		architecture / design / implementation
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

AutosarC++18_03-A2.13.5

Hexadecimal constants should be upper case.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
hex_literal	Hexadecimal constants should be upper case.

AutosarC++18_03-A2.13.6

Universal character names shall be used only inside character or string literals.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_comments	Whether to allow universal character names in comments.	False
allow_in_string_literals	Whether to allow universal character names in string literals.	True
enforcement		automated
level		required
target		architecture / design / implementation

Possible Messages

Name	Message
universal_character	Use of universal character names

AutosarC++18_03-A3.1.1

It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
accept_const_fields	Whether const-qualified static fields in header files should be tolerated.	True
accept_const_variables	Whether global const variables in header files should be tolerated.	True
enforcement		automated
find_non_external_function_declaration	Whether function declarations with non-external linkage should be reported.	True
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
function_definition_in_header	Definition in header file.
static_field_def_in_header	Definition in header file.
variable_definition_in_header	Definition in header file.

AutosarC++18_03-A3.1.2

Header files, that are defined locally in the project, shall have a file name' extension of one of: ".h", ".hpp" or "..hxx".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed_extensions	Which file extensions are allowed.	set[{'h', '.hxx', 'hpp'}]
enforcement		automated
file_type	The files to check the extensions of.	User_Include_File
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
illegal_file_extension	Use only certain extensions for file names.

AutosarC++18_03-A3.1.3

Implementation files, that are defined locally in the project, should have a file name extension of ".cpp".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed_extensions	Which file extensions are allowed.	set[{'cpp'}]
enforcement		automated
file_type	The files to check the extensions of.	Primary_File
level		advisory
target		architecture/design/implementation

Possible Messages

Name	Message
illegal_file_extension	Use only certain extensions for file names.

AutosarC++18_03-A3.1.4

When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_extern	Whether to consider only extern declared arrays	True
report_definitions	Whether definitions of array variables should also be reported	True
target		design/implementation

Possible Messages

Name	Message
unbounded_array	Array declaration with unknown size.

AutosarC++18_03-A3.1.6

Trivial accessor and mutator functions should be inlined.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
exclude_constructor_destructors	Whether to exclude constructors/destructors from inspected functions.	True
exclude_operator_functions	Whether to exclude operator functions from inspected functions.	False
exclude_virtual_functions	Whether to exclude virtual functions from inspected functions.	True
level		advisory
require_this_read_write	Whether to require a read/write on this for a function to be considered a trivial accessor/mutator.	False
target		design

Possible Messages

Name	Message
missing_inline	Trivial accessor and mutator functions should be inlined.

AutosarC++18_03-A3.3.1

Objects or functions with external linkage shall be declared in a header file.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_function_declaration_in_header	Object or function with external linkage shall be declared in a header file
missing_variable_declaration_in_header	Object or function with external linkage shall be declared in a header file

AutosarC++18_03-A3.3.2

Static and thread-local objects shall be constant-initialized.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed_effects_mask	If set, static and thread-local objects may also be initialized with a non-constexpr function/constructor call that only has the specified side-effects.	None
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
constexpr_init_variable	Static and thread-local objects shall be constant-initialized.

AutosarC++18_03-A3.9.1

Typedefs that indicate size and signedness should be used in place of the basic numerical types.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_floating_point	Whether floating point types without should not be reported.	True
enforcement		automated
ignore_inherited	If true, missing typeDefs in inherited methods are not reported.	False
level		required
target		implementation
treat_plain_char_as_numerical	Whether the char type without any signedness should be reported.	True

Possible Messages

Name	Message
missing_integer_TypeDef	Use of base type outside typeDef.
wrong_integer_TypeDef	Use of badly named typeDef for base type.

AutosarC++18_03-A4.5.1

Expressions with type enum shall not be used as operands to built-in operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_use_in_operator_calls	Whether enum arguments in calls to forbidden overloaded operators should be reported.	True
target		implementation

Possible Messages

Name	Message
enum_operand_outside_comparison	Use of enum operand in arithmetic or similar context

AutosarC++18_03-A4.7.1

An integer expression shall not lead to data loss.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
abstract_interpretation_overflow	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cast_overflow	Cast on result of arithmetic computation may cause overflow
cast_truncate	Cast may truncate value
cast_underflow	Cast on result of arithmetic computation may cause underflow
overflow	Arithmetic computation may cause overflow
underflow	Arithmetic computation may cause underflow

AutosarC++18_03-A4.10.1

Only nullptr literal shall be used as the null-pointer-constant.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
require_nullptr	Require C++11 nullptr instead of macro NULL or literal zero {0}.	True
target		architecture/design/implementation

Possible Messages

Name	Message
null_constant	Only nullptr literal shall be used as the null-pointer-constant.
zero_as_null	Use of literal zero {0} as null-pointer-constant, use {} instead

AutosarC++18_03-A5.0.1

The value of an expression shall be the same under any order of evaluation that the standard permits.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_exceptions	Ignore possible exceptions in the computation of side effects. (this option only has an effect if report_calls is enabled)	False
level		required
report_calls	If True, unsequenced function calls are reported.	True
target		implementation

Possible Messages

Name	Message
unsequenced_call_volatile	Function calls unsequenced with volatile access
unsequenced_calls	Unsequenced function calls
unsequenced_read_write	Unsequenced read and write accesses
unsequenced_volatiles	Unsequenced volatile accesses
unsequenced_writes	Unsequenced writes

AutosarC++18_03-A5.0.2

The condition of an if-statement and the condition of an iteration-statement shall have type bool.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_in_conditions	If True, only logical operators inside conditions are checked.	True
report_do_while_false	Whether do ... while[0] should be reported.	True
report_while_true	Whether while[1] ... should be reported.	True
target		implementation

Possible Messages

Name	Message
nonbool_if_condition	Condition must have type bool
nonbool_logical_operator_operand	Sub-condition must have type bool
nonbool_loop_condition	Condition must have type bool

AutosarC++18_03-A5.0.3

The declaration of objects should contain no more than two levels of pointer indirection.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
max_levels	Maximum number of allowed pointer-indirection levels.	2
target		implementation

Possible Messages

Name	Message
too_many_pointer_levels	More than {} level{} of pointer indirection

AutosarC++18_03-A5.0.4

Pointer arithmetic shall not be used with pointers to non-final classes.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
consider_non_base_final	Whether to consider a class that is not used as a base class as a sort-of final class.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
non_final_pointer_arithmetic	Pointer arithmetic on non-final classes not allowed

AutosarC++18_03-A5.1.1

Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_nonconst_variable_initialization	If set to `True`, allow the initialization of a non-const variable with a string literal in addition to the initialization of a const variable. A reassignment of such a variable is still a violation.	False
allowed	Literal values that are ok.	{0.0, 1.0, 2.0}
allowed_contexts	Optional set of PIR classes or functions {[node] -> bool} for allowed contexts, e.g. Case_Label.	set[<function allow_in_type_initialization at 0x7dfa84459b0>]
allowed_logging_contexts	List of fully qualified function types that are considered logging contexts (using operator<>). If this is non-empty, `std::throw()` is considered a valid logging context as well.	['std::basic_ostream']
allowed_string_contexts	Optional set of PIR classes or functions {[node] -> bool} for allowed contexts.	set[<function allow_in_type_initialization at 0x7dfa84459b0>]
allowed_strings	Literal values that are ok.	["' '"]
enforcement		partially automated
exceptions	Optional predicate to filter out cases that should be allowed as an exception. The predicate takes a literal node and returns True if the given node is such an exception.	None
exclude_pp_literals	If True, literals in conditions of #if are ignored.	True
exclude_single_uses	If True, report only literals used more than once.	False
level		required
target		implementation

Possible Messages

Name	Message
magic_number	Use of magic literal.
magic_number_without_token	Use of magic literal.
magic_string	Use of magic string literal.
possible_magic_number	Potential use of magic literal.

AutosarC++18_03-A5.1.2

Variables shall not be implicitly captured in a lambda expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
implicit_lambda_capture	Variables shall not be implicitly captured in a lambda expression.

AutosarC++18_03-A5.1.3

Parameter list (possibly empty) shall be included in every lambda expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_parameters	Include a (possibly empty) parameter list

AutosarC++18_03-A5.1.6

Return type of a non-void return type lambda expression should be explicitly specified.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
missing_explicit_return	Add an explicit return type for the lambda.

AutosarC++18_03-A5.1.7

The underlying type of lambda expression shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
underlying_lambda_type	The underlying type of lambda expression shall not be used.

AutosarC++18_03-A5.1.8

Lambda expressions should not be defined inside another lambda expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
nested_lambda	Lambda expressions should not be defined inside another lambda expression

AutosarC++18_03-A5.1.9

Identical unnamed lambda expressions shall be replaced with a named function or a named lambda expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_empty_lambdas	If set to True, this rule will not report duplicates of the trivial empty lambda.	False
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
identical_unnamed_lambda	Identical unnamed lambdas.

AutosarC++18_03-A5.2.1

dynamic_cast should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
dynamic_cast	dynamic_cast should not be used.

AutosarC++18_03-A5.2.2

Traditional C-style casts shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_void_cast	If True, (void) is always allowed, else only for return value of calls.	False
allow_void_cast_on_call	If True, (void) is allowed, for return value of calls.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
c_cast	Use of C-style cast in C++ unit.

AutosarC++18_03-A5.2.3

A cast shall not remove any const or volatile qualification from the type of a pointer or reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
check_const	Whether casts affecting const-qualifiers should be reported	True
check_volatile	Whether casts affecting volatile-qualifiers should be reported	True
enforcement		automated
level		required
only_top_level	Whether the check only applies to the top-level pointee or recursively to deeper pointer levels as well	False
target		implementation

Possible Messages

Name	Message
cast_removes_const	Cast removes const qualification
cast_removes_volatile	Cast removes volatile qualification
implicit_cast_removes_const	Implicit cast removes const qualification
implicit_cast_removes_volatile	Implicit cast removes volatile qualification

AutosarC++18_03-A5.2.4

reinterpret_cast shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
reinterpret_cast	reinterpret_cast should not be used.

AutosarC++18_03-A5.2.6

The operands of a logical && or || shall be parenthesized if the operands contain binary operators.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed	Can be used to name additional allowed IR node types for the operands.	None
enforcement		automated
level		required
require_postfix_expression	Whether postfix or primary expressions are required as operands.	False
target		implementation

Possible Messages

Name	Message
missing_parens_for_logical_operator_operand	Operand of && or must be parenthesized.

AutosarC++18_03-A5.3.1

Evaluation of the operand to the typeid operator shall not contain side effects.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
level		required
target		implementation

Possible Messages

Name	Message
side_effect_in_typeid	Operand of "typeid" shall not contain side effects

AutosarC++18_03-A5.6.1

The right hand operand of the integer division or remainder operators shall not be equal to zero.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
abstract_interpretation_div_by_zero	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
division_by_zero	Division by zero
modulo_by_zero	Modulo by zero
possible_division_by_zero	Possible division by zero
possible_modulo_by_zero	Possible modulo by zero

AutosarC++18_03-A5.10.1

A pointer to member virtual function shall only be tested for equality with null-pointer-constant.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
member_virtual_function_comparison	Comparison of a member virtual function with non-nullptr.

AutosarC++18_03-A5.16.1

The ternary conditional operator shall not be used as a sub-expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_nested_conditional_operator	Use of nested conditional operator.

AutosarC++18_03-A6.4.1

Every switch statement shall have at least one case-clause.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
minimum_switch_cases	The number of cases a switch statement should at least have.	2
target		implementation

Possible Messages

Name	Message
empty_switch	Switch has no "case" clause.
minimum_switch_cases	Switch has too little "case" clauses.

AutosarC++18_03-A6.5.1

A for-loop that loops through all elements of the container and does not use its loop-counter shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
exclude_simple_for_loops	If set to `True` this rule will not report unused loop counters in the simple cases where the C for-loop only references loop-counters in its condition but no other variables.	True
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
could_be_range_based	For loop could be a range-based for loop.
unused_loop_counter	For loop does not use its loop counter.

AutosarC++18_03-A6.5.2

A for loop shall contain a single loop-counter which shall not have floating type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
loop_float_counter	Loop-counter of for loop shall not have floating type
loop_missing_counter	For loop has no loop-counter
loop_multiple_counters	For loop shall have only a single loop-counter

AutosarC++18_03-A6.5.3

Do statements should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
do_while_loop	Do statements should not be used.

AutosarC++18_03-A6.5.4

For-init-statement and expression should not perform actions other than loop-counter initialization and modification.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
loop_side_effect	Side effect in loop init/expression.
non_loop_counter_initialized	Non loop counter initialized.
non_loop_counter_modified	Non loop counter modified.

AutosarC++18_03-A6.6.1

The goto statement shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
goto	Do not use goto.

AutosarC++18_03-A7.1.1

Constexpr or const specifiers shall be used for immutable data declaration.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_pointer_variables	Whether variables of pointer type should be ignored.	False
level		required
only_check_unit_locals	Whether only local variables and global static variables should be checked.	False
only_immutable_data	Whether only declarations with immutable data should be checked.	True
target		implementation

Possible Messages

Name	Message
variable_missing_const	An immutable variable shall be const/constexpr qualified.

AutosarC++18_03-A7.1.3

CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_nonpointer_typedefs	Whether to allow the qualifier on the rhs of a type, if it is a non-pointer typedef.	False
allowed_typedefs	Set of names for typedefs/usings that are allowed (e.g. int32_t).	set(['int32_t', 'uint_least64_t', 'intptr_t', 'uintmax_t', 'int_fast16_t', 'intmax_t', 'int_fast8_t', 'int64_t', 'size_t', 'int_fast64_t', 'time_t', 'uint8_t', 'lldiv_t', 'int_least8_t', 'div_t', 'uint_least16_t', 'clock_t', 'uint_least32_t', 'int_least64_t', 'int_least16_t', 'int_least32_t', 'uint_least8_t', 'uintptr_t', 'max_align_t', 'int8_t', 'fpos_t', 'ldiv_t', 'uint_fast32_t', 'uint_fast64_t', 'nullptr_t', 'int_fast32_t', 'uint_fast16_t', 'uint32_t', 'ptrdiff_t', 'int16_t', 'uint64_t', 'uint16_t', 'uint_fast8_t'])
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
lhs_cv_qualifier	CV qualifier on the lhs of a type.

AutosarC++18_03-A7.1.4

The register keyword shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
register_keyword	The register keyword shall not be used

AutosarC++18_03-A7.1.5

The auto specifier shall not be used apart from following cases: (1) to declare that a variable has the same type as return type of a function call, (2) to declare that a variable has the same type as initializer of non-fundamental type, (3) to declare parameters of a generic lambda expression, (4) to declare a function template using trailing return type syntax.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_for_loop_counter	Disables message if auto is used to declare a for-loop counter.	False
allow_generic_lambda_parameters	Allows auto as parameter type in a generic lambda	True
allow_nonfundamental_initializer	Allows auto to declare variables having a function call or initializer of non-fundamental type	True
allow_template_instance	Disables message if auto stands for a template instance.	False
allowed_contexts	Set of context predicates in which auto is allowed.	set[]
allowed_types	Set of types for which auto is allowed, given as name pattern, function, or LIR class name.	set[]
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cpp11_auto	Use of C++11 auto type specifier.

AutosarC++18_03-A7.1.6

The typedef specifier shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
typedef_specifier	The typedef specifier shall not be used.

AutosarC++18_03-A7.1.7

Each expression statement and identifier declaration shall be placed on a separate line.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_for_loop_init	Whether a multi-declaration is allowed in the for-init statement	True
allow_uninitialized_simple_type	Whether a multi-declaration of simple type is allowed when there is no initialization	False
enforcement		automated
ignore_stmts	Statements to be ignored when counting statements.	['Statement_Sequence', 'C_For_Loop']
level		required
target		implementation

Possible Messages

Name	Message
multi_declaration	Multiple declarators in single declaration
multiple_statements_per_line	Multiple statements per line.

AutosarC++18_03-A7.1.9

A class, structure, or enumeration shall not be declared in the definition of its type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
type_defined_in_declaration	Type defined in declaration.

AutosarC++18_03-A7.2.1

An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
asgn_not_an_enumerator	Assignment of value which does not correspond to an enumerator
asgn_out_of_enum_range	Assignment of value outside the range of the enumerators
conversion_creating_bad_enum_value	Expression does not correspond to an enumerator in {}

AutosarC++18_03-A7.2.2

Enumeration underlying base type shall be explicitly defined.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unbased_enum	Enumeration underlying base type shall be explicitly defined.

AutosarC++18_03-A7.2.3

Enumerations shall be declared as scoped enum classes.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unscoped_enum	Enumerations shall be declared as scoped enum classes.

AutosarC++18_03-A7.2.4

In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
incomplete_enum_init	Do not initialize enumerators other than the first, or initialize all

AutosarC++18_03-A7.3.1

All overloads of a function shall be visible from where it is called.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_assignment_operator_hiding	If set to true, we allow to hide the assignment operators, as pulling in the parent members using a using-declaration may allow assigning an instance of a parent class to a variable with the type of an inheriting class.	True
enforcement		automated
level		required
report_hidden_fields	Whether to report fields hidden by means of inheritance	False
report_hidden_methods	Whether to report methods hidden by means of inheritance	True
report_other_usages	Whether to find other usages (such as function calls, or taking a fp-reference) between multiple declarations for an identifier.	True
target		implementation

Possible Messages

Name	Message
declarations_surrounding_usage	Declarations straddle a usage
declarations_surrounding_using	Declarations straddle a using-declaration
hiding	{ } hides { }

AutosarC++18_03-A7.4.1

The asm declaration shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_assembler	Use of assembler.

AutosarC++18_03-A7.5.1

A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_const_reference	Whether only to report returns of parameters with reference to const.	True
target		implementation

Possible Messages

Name	Message
returning_reference_to_refparam	Returning reference(pointer to reference parameter.

AutosarC++18_03-A7.5.2

Functions shall not call themselves, either directly or indirectly.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_cycle	Cycle of recursive functions
recursive_function	Function is directly recursive

AutosarC++18_03-A7.6.1

Functions declared with the [[noreturn]] attribute shall not return.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
noreturn_violation	Do not return from a noreturn function.

AutosarC++18_03-A8.2.1

When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_trailing_return	Use trailing return-type syntax for function templates.

AutosarC++18_03-A8.4.1

Functions shall not be defined using the ellipsis notation.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_declarations	Allow declarations of functions with variable number of arguments (e.g. for existing library functions); only report definitions.	True
ignore_inherited	Do not report functions inheriting a variable number of arguments.	False
level		required
target		implementation

Possible Messages

Name	Message
ellipsis_parameter	Function definitions shall not use ellipsis

AutosarC++18_03-A8.4.2

All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
lambda_return_missing_value	Return without value in non-void lambda expression
missing_return	Non-void function needs return with value at end.
missing_return_in_lambda	Non-void lambda expression needs return with value at end.
return_missing_value	Return without value in non-void function

AutosarC++18_03-A8.4.4

Multiple output values from a function should be returned as a struct or tuple.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_modifying_calls	Whether a function may call a function on a parameter that modifies it, without regarding it as a violation here.	True
allow_move_parameters	Whether a move parameter (a rvalue reference) may be written to without triggering a violation. Note: move-constructors and move-assignments are allowed in either case.	False
allow_this_modification	Whether a function may call a member-function on a parameter that modifies this, that is the internal parameter state ('allow_modifying_calls' must be False if this is set to False)	True
enforcement		automated
exclude_constructors	Whether not to report parameters on constructors	True
exclude_operators	Whether not to report parameters on operator functions. Note: if set to True, this also sets 'exclude_shift_operators=True'.	False
exclude_shift_operators	Whether not to report parameters on shift operator functions (operator<<(), operator<<=(), operator>>(), and operator>>=).	False
level		advisory
only_report_multiple_output_parameters	Whether to only report output-parameters, if a function has more than one or also returns a value.	True
record_field_modification_threshold	A percentage of how many record fields may be modified, before the record pointer is considered an output parameter. 0 : every field modification counts as an output parameter 100: the record pointer is only regarded an output parameter if all field are modified	0
target		design

Possible Messages

Name	Message
output_parameter	Use return value instead of output parameter.

AutosarC++18_03-A8.4.5

"consume" parameters declared as X && shall always be moved from.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
exclude_move_assignments_and_constructors	If set to 'True', "consume" parameters in move-assignments and -constructors are not reported, even if they are not moved from.	True
level		required
target		design

Possible Messages

Name	Message
missing_move_consume	"consume" parameters shall always be moved from

AutosarC++18_03-A8.4.6

"forward" parameters declared as T && shall always be forwarded.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		design

Possible Messages

Name	Message
only_forward	"forward" parameters shall always be forwarded.

AutosarC++18_03-A8.4.7

"in" parameters for "cheap to copy" types shall be passed by value.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
exclude_operator_functions	If set to `True`, in parameters with "cheap to copy" types that are not passed by value are not reported for operator functions, as one might not have an option to change the signature.	False
level		required
size_threshold	Size of a type to be considered cheap to copy in number of words.	2
target		design
word_size	Size of a machine word in number of bytes. If None, uses the size of a pointer.	None

Possible Messages

Name	Message
input_parameter	Pass input parameters by value

AutosarC++18_03-A8.4.8

Output parameters shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_modifying_calls	Whether a function may call a function on a parameter that modifies it, without regarding it as a violation here.	True
allow_move_parameters	Whether a move parameter (a rvalue reference) may be written to without triggering a violation. Note: move-constructors and move-assignments are allowed in either case.	False
allow_this_modification	Whether a function may call a member-function on a parameter that modifies this, that is the internal parameter state ('allow_modifying_calls' must be False if this is set to False)	True
enforcement		automated
exclude_constructors	Whether not to report parameters on constructors	False
exclude_operators	Whether not to report parameters on operator functions. Note: if set to True, this also sets 'exclude_shift_operators=True'.	False
exclude_shift_operators	Whether not to report parameters on shift operator functions (operator<<(), operator<<=(), operator>>(), and operator>>=).	False
level		required
only_report_multiple_output_parameters	Whether to only report output-parameters, if a function has more than one or also returns a value.	False
record_field_modification_threshold	A percentage of how many record fields may be modified, before the record pointer is considered an output parameter. 0 : every field modification counts as an output parameter 100: the record pointer is only regarded an output parameter if all field are modified	0
target		design

Possible Messages

Name	Message
output_parameter	Don't use output parameters.

AutosarC++18_03-A8.4.9

"in-out" parameters declared as T & shall be modified.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
exclude_constructors	Whether not to report parameters on constructors	True
level		required
target		design

Possible Messages

Name	Message
complete_replacement	"in-out" completely replaced without being read.
missing_modify	"in-out" parameters not modified, consider making it const.

AutosarC++18_03-A8.5.1

In an initialization list, the order of initialization shall be following: (1) virtual base classes in depth and left to right order of the inheritance graph, (2) direct base classes in left to right order of inheritance list, (3) non-static data members in the order they were declared in the class definition.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	If provided, only messages of these types are reported.	[1719]
reported_severities	List of severities to display.	('error', 'warning', 'remark')
target		implementation
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

AutosarC++18_03-A8.5.2

Braced-initialization {}, without equals sign, shall be used for variable initialization.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
old_style_variable_init	Braced-initialization {}, without equals sign, shall be used for variable initialization.

AutosarC++18_03-A8.5.3

A variable of type auto shall not be initialized using {} or ={} braced-initialization.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
braced_auto_variable_init	A variable of type auto shall not be initialized using {} or ={} braced-initialization.

AutosarC++18_03-A8.5.4

A constructor taking parameter of type std::initializer_list shall only be defined in classes that internally store a collection of objects.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
level		advisory
target		implementation

Possible Messages

Name	Message
constructor_init_list_container	The class should be a container if a constructor taking std::initializer_list is defined.

AutosarC++18_03-A9.5.1

Unions shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_tagged_unions	Whether tagged unions (nested in a struct that has an enum discriminator) should be allowed	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
union	Unions shall not be used

AutosarC++18_03-A9.6.1

Bit-fields shall be either unsigned integral, or enumeration [with underlying type of unsigned integral type].

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
invalid_bitfield	Bit-fields shall be either unsigned integral, or enumeration (with underlying type of unsigned integral type).

AutosarC++18_03-A10.1.1

Class shall not be derived from more than one base class which is not an interface class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_pure_interfaces	Whether C++ interfaces are allowed, i.e. classes with only pure virtual members.	True
level		required
target		implementation

Possible Messages

Name	Message
multiple_inheritance	Use of multiple inheritance.

AutosarC++18_03-A10.2.1

Non-virtual member functions shall not be redefined in derived classes.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
redefinition_of_nonvirtual_function	Redefinition/hiding of non-virtual function.

AutosarC++18_03-A10.3.1

Virtual function declaration shall contain exactly one of the three specifiers: {1} virtual, {2} override, {3} final.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
disable_for_destructors	If set, destructors are not checked.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
invalid_combination	Use only one of {1} virtual, {2} override, {3} final.

AutosarC++18_03-A10.3.2

Each overriding virtual function shall be declared with the override or final specifier.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_final	If set to True, don't report overriding virtual functions declared with final.	True
enforcement		automated
ignore_destructors	If set to False, also report destructors. Note that not all compilers support this.	False
level		required
target		implementation

Possible Messages

Name	Message
missing_override	Override of functions is only permitted with keyword override/final.

AutosarC++18_03-A10.3.3

Virtual functions shall not be introduced in a final class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
virtual_method	Virtual functions shall not be introduced in a final class.
virtual_method_override	Virtual functions shall not be overriden without final in a final class.

AutosarC++18_03-A10.3.5

A user-defined assignment operator shall not be virtual.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
assignment_virtual	A user-defined assignment operator shall not be virtual.

AutosarC++18_03-A11.0.1

A non-POD type should be defined as class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
require_class_for_pod	Whether POD types should be classes as well	False
target		implementation

Possible Messages

Name	Message
cpp_struct	Use of struct in C++ unit.

AutosarC++18_03-A11.0.2

A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
no_function_member	Struct shall not provide any member functions or methods.
no_struct_as_base	A struct shall not be a base of another struct or class.
no_struct_inheritance	A struct shall not inherit from another struct or class.
non_public_member	Structs shall only have public data members.

AutosarC++18_03-A11.3.1

Friend declarations shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
friend_class	Do not use friend class/struct/union declarations.
friend_decl	Do not use friend declarations.

AutosarC++18_03-A12.0.1

If a class declares a copy or move operation, or a destructor, either via "=default", "=delete", or via a user-provided declaration, then all others of these five special member functions shall be declared as well.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_defaulted_destructor_only	Allow classes with a defaulted destructor and no other special member functions.	False
allow_destructor_only	Allow all destructors without copy or move constructors	False
allow_empty_destructor	Allow empty destructors without copy or move constructors.	False
allow_missing_destructor	Suppress messages about missing destructors.	False
enforcement		automated
ignore_pod_classes	Whether POD classes should be checked at all	False
level		required
target		implementation

Possible Messages

Name	Message
missing_constructor_and_asgn	Class with destructor should also declare a copy or move constructor and assignment operator.
missing_copy_asgn	Class with copy constructor is missing copy assignment operator.
missing_copy_constructor	Class with copy assignment operator is missing copy constructor.
missing_destructor	Class with copy or move constructors or assignment operators should also declare a destructor.
missing_move_asgn	Class with move constructor is missing move assignment operator.
missing_move_constructor	Class with move assignment operator is missing move constructor.

AutosarC++18_03-A12.1.1

Constructors shall explicitly initialize all virtual base classes, all direct non-virtual base classes and all non-static data members.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
init_functions	Names of functions to be inspected as well when called directly from constructor.	('Init', 'init')
inspect_directly_called_methods	Inspect all methods directly called from constructor.	False
level		required
only_member_initializer_list	Only inspect member initializer list and not the constructor body/methods.	False
report_missing_field_constructors	Enables detection of constructors which rely on implicit field constructor calls.	True
target		implementation

Possible Messages

Name	Message
implicit_field_init	Field is only implicitly initialized in constructor.
missing_base_class_init	Base class is not explicitly initialized in constructor.
missing_field_init	Field is not initialized in constructor.

AutosarC++18_03-A12.1.2

Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_member_initializer_list	Only inspect member initializer list and not the constructor body/methods.	True
target		implementation

Possible Messages

Name	Message
nsdmi_mixed	Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type.

AutosarC++18_03-A12.1.3

If all user-defined constructors of a class initialize data members with constant values that are the same across all constructors, then data members shall be initialized using NSDMI instead.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_constructors_of_types	List of fully qualified type names which constructors are considered literals, without regard to the actual arguments.	[]
allow_literal_like_constructors	Whether to consider constructors that take only literals as arguments a literal. If set to true, this is checked recursively.	False
enforcement		automated
level		required
min_number_common_inits	Minimum number (inclusive) of common initializations to enforce use of NSDMI.	1
target		implementation

Possible Messages

Name	Message
use_nsdmi	Use NSDMI for common constant initializations ({}).

AutosarC++18_03-A12.1.4

All constructors that are callable with a single argument of fundamental type shall be declared explicit.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
consider_only_fundamental_types	Whether this check should be limited to single arguments of fundamental type or should also be applied to user defined types.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
constructor_missing_explicit	Constructor shall be declared explicit

AutosarC++18_03-A12.1.5

Common class initialization for non-constant members shall be done by a delegating constructor.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_copy_move_constructor	Whether to allow direct field initialization without a delegating constructor call for copy and move constructors.	False
enforcement		partially automated
level		required
target		implementation

Possible Messages

Name	Message
delegating_constructor	Use delegating constructor for common initialization.

AutosarC++18_03-A12.1.6

Derived classes that do not need further explicit initialization and require all the constructors from the base class shall use inheriting constructors.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_inheriting	Use inheriting constructors if possible.

AutosarC++18_03-A12.4.1

Destructor of a base class shall be public virtual, public override or protected non-virtual.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
base_class_destructor	Destructor of a base class shall be public virtual, public override or protected non-virtual.

AutosarC++18_03-A12.4.2

If a public destructor of a class is non-virtual, then the class should be declared final.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
non_final	If a public destructor of a class is non-virtual, then the class should be declared final.

AutosarC++18_03-A12.6.1

All class data members that are initialized by the constructor shall be initialized using member initializers.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
init_functions	Names of functions to be inspected as well when called directly from constructor.	{'Init', 'init'}
inspect_directly_called_methods	Inspect all methods directly called from constructor.	False
level		required
only_member_initializer_list	Only inspect member initializer list and not the constructor body/methods.	True
report_missing_field_constructors	Enables detection of constructors which rely on implicit field constructor calls.	True
target		implementation

Possible Messages

Name	Message
implicit_field_init	Field is only implicitly initialized in constructor.
missing_field_init	Field is not initialized in constructor.

AutosarC++18_03-A12.8.1

A copy constructor shall only initialize its base classes and the non-static members of the class of which it is a member.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_copy_constructor	Whether to report side effects on copy constructors.	True
report_move_constructor	Whether to report side effects on move constructors.	True
target		implementation

Possible Messages

Name	Message
copy_ctor_with_side_effect	Copy Constructor has side-effect
move_ctor_with_side_effect	Move Constructor has side-effect

AutosarC++18_03-A12.8.2

User-defined copy and move assignment operators should use user-defined no-throw swap function.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
asgn_no_swap	A move/copy assignment operator shall use a no-throw swap function.
asgn_not_nothrow	Used swap function is not no-throw.

AutosarC++18_03-A12.8.3

Moved-from object shall not be read-accessed.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		partially automated
inspect_class_field_moves	Detecting moved-from accesses on class fields requires heavy computation. This can be disabled by setting this option to False, but would result in false-positives in the context of class fields.	True
level		required
read_move_function_exceptions	Names of functions that are considered to leave the moved from objects in a well-specified state and are therefore except.	['std::unique_ptr::unique_ptr', 'std::unique_ptr::operator=', 'std::shared_ptr::shared_ptr', 'std::shared_ptr::operator=', 'std::weak_ptr::weak_ptr', 'std::weak_ptr::operator=', 'std::basic_filebuf::basic_filebuf', 'std::basic_filebuf::operator=', 'std::thread::thread', 'std::thread::operator=', 'std::unique_lock::unique_lock', 'std::unique_lock::operator=', 'std::shared_lock::shared_lock', 'std::shared_lock::operator=', 'std::promise::promise', 'std::promise::operator=', 'std::future::future', 'std::future::operator=', 'std::shared_future::shared_future', 'std::shared_future::operator=', 'std::packaged_task::packaged_task', 'std::packaged_task::operator=']
read_move_type_exceptions	Names of types that are considered to be left well-specified state after a move and are therefore except.	['std::basic_ios']
target		implementation

Possible Messages

Name	Message
moved_from_read	Don't read-access a moved-from object

AutosarC++18_03-A12.8.4

Move constructor shall not initialize its class members and base classes using copy semantics.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
move_constructor	Move constructor shall not initialize using copy semantics.

AutosarC++18_03-A12.8.6

Copy and move constructors and copy assignment and move assignment operators shall be declared protected or defined "=delete" in base class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
special_asgn	Copy and move assignment operators shall be declared protected or defined "=delete" in base class.
special_ctor	Copy and move constructors shall be declared protected or defined "=delete" in base class.

AutosarC++18_03-A12.8.7

Assignment operators should be declared with the ref-qualifier &.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
asgn_missing	Assignment operators should be declared with the ref-qualifier &.

AutosarC++18_03-A13.1.2

User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
user_defined_literal_naming	User-defined literals shall have a suffix matching "[a-zA-Z]+".

AutosarC++18_03-A13.1.3

User defined literals operators shall only perform conversion of passed parameters.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
literal_side_effects	User-defined literals shall not have side effects.

AutosarC++18_03-A13.2.1

An assignment operator shall return a reference to "this".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
asgn_ref_this	An assignment operator shall return a reference to "this".

AutosarC++18_03-A13.2.2

A binary arithmetic operator and a bitwise operator shall return a "prvalue".

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_bitwise_shift	Whether to allow non-basic values for operator>> or operator<<.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
arith_bitwise_basic_value	Binary arithmetic or bitwise operator shall return a basic value.

AutosarC++18_03-A13.2.3

A relational operator shall return a boolean value.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
relational_bool	A relational operator shall return a boolean value.

AutosarC++18_03-A13.3.1

A function that contains "forwarding reference" as its argument shall not be overloaded.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
overloaded_fref	Functions that contain forwarding reference parameters shall not be overloaded

AutosarC++18_03-A13.5.1

If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
array_access_non_const	No const version of operator[] implemented.

AutosarC++18_03-A13.5.2

All user-defined conversion operators shall be defined explicit.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
user_defined_conversion_explicit	All user-defined conversion operators shall be defined explicit.

AutosarC++18_03-A13.5.3

User-defined conversion operators should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
no_user_defined_conversion	User-defined conversion operators should not be used.

AutosarC++18_03-A13.5.4

If two opposite operators are defined, one shall be defined in terms of the other.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
implement_in_terms	Implement in terms of other operator

AutosarC++18_03-A13.6.1

Digit sequences separators ' shall only be used as follows: {1} for decimal, every 3 digits, {2} for hexadecimal, every 2 digits, {3} for binary, every 4 digits.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
digit_separator	Possibly unreadable digit separators.

AutosarC++18_03-A14.7.2

All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_type_declaration_file	Also allow specializations in the same file of a user-defined type for which the specialization is declared.	True
enforcement		automated
level		required
relax_if_template_only_declared	Allow specializations that are not declared in the header file if the template itself is only declared but not defined in the header.	False
target		implementation

Possible Messages

Name	Message
template_specialization_in_different_file	Specialization not declared in same file as primary template

AutosarC++18_03-A14.8.2

Overloaded function templates shall not be explicitly specialized.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
specialization_of_overloaded_template	Overloaded function templates shall not be explicitly specialized

AutosarC++18_03-A15.1.1

Only instances of types derived from std::exception shall be thrown.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throwing_std_exception	Only instances of types derived from std::exception shall be thrown.

AutosarC++18_03-A15.1.2

An exception object should not have pointer type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throwing_pointer	Exception object of pointer type

AutosarC++18_03-A15.1.3

All thrown exceptions should be unique.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
throwing_duplicate	All thrown exceptions should be unique.

AutosarC++18_03-A15.2.1

Constructors that are not noexcept shall not be invoked before program startup.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
static_field_only_noexcept	Constructor called that may throw an exception in static field.
static_variable_only_noexcept	Constructor called that may throw an exception in static variable.

AutosarC++18_03-A15.3.3

Main function and a task main function shall catch at least: base class exceptions from all third-party libraries used, std::exception and all otherwise unhandled exceptions.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		partially-automated
external_base_exceptions	Sequence of external/third-party exception base-classes that should be caught	[]
inspect_thread_main	Whether to also inspect thread main functions.	True
level		required
std_base_exceptions	Sequence of cpp-std exception base-classes that should be caught	['std::exception']
target		implementation

Possible Messages

Name	Message
missing_catch_all	Catch-all required around main program body
missing_catch_handler	Handler for {} needed

AutosarC++18_03-A15.3.4

Catch-all (ellipsis and std::exception) handlers shall be used only in (a) main, (b) task main functions, (c) in functions that are supposed to isolate independent components and (d) when calling third-party code that uses exceptions not according to AUTOSAR C++14 guidelines.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_extern_c_functions	Whether to allow a catch-all in `extern C` functions.	False
allow_in_functions_matching	If not None, a `re.compile()`ed object where functions whose qualified name matches the regex are allowed to contain catch-alls.	None
allow_in_main	Whether to allow a catch-all in the main() function.	True
allow_in_thread_main	Whether to allow a catch-all in thread-main functions.	True
allow_rethrow	Whether to allow a catch-all with a re-throw.	False
autosar_exception_model	The default model for what counts as an checked / unchecked exception.	<bauhaus.ir.autosar.exceptions.autosar_exceptions.AutosarExceptionModel object at 0x7dfa83c4650>
disallow_std_exception	Whether to consider catching the literal std::exception as a catch-all.	True
enforcement		non-automated
level		required
target		implementation

Possible Messages

Name	Message
catch_all	Use of catch(...).
catch_std_exception	Catching std::exceptions is too general.

AutosarC++18_03-A15.3.5

A class type exception shall always be caught by reference.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_class_types	Whether all types should be caught by reference or only class types.	True
target		implementation

Possible Messages

Name	Message
catch_without_reference	A class type exception shall always be caught by reference.

AutosarC++18_03-A15.4.1

Dynamic exception-specification shall not be used.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

AutosarC++18_03-A15.4.2

If a function is declared to be noexcept, noexcept(true) or noexcept(<true condition>), then it shall not exit with an exception.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
generate_violation_path	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignore_constructor_destructor	Whether to ignore escaping exceptions from constructors and destructors.	False
ignore_throwing_functions	Whether to ignore noexcept specification violations on function that actually throw an exception.	False
ignore_unknown_routines	Whether to ignore extern or only declared routines.	False
level		required
report_noexcept_falseViolations	Whether to report cases where the function is declared noexcept(false), but no exceptions are ever thrown (directly or indirectly).	False
target		implementation

Possible Messages

Name	Message
implicit_noexcept_spec_violation_without	Function implicitly declared noexcept(false) but no exceptions will be thrown.
noexcept_spec_violation_with	Exception violates function's noexcept-specification.
noexcept_spec_violation_without	Function declared noexcept(false) but no exceptions will be thrown.

AutosarC++18_03-A15.4.3

Function's noexcept specification shall be either identical or more restrictive across all translation units and all overriders.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
noexcept_mismatch_redecl	Function's noexcept specification shall be identical
noexcept_mismatch_override	exception specification for virtual function "{}" is incompatible with that of overridden function "{}"

AutosarC++18_03-A15.4.4

A declaration of non-throwing function shall contain noexcept specification.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
generateViolationPath	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignoreConstructorDestructor	Whether to ignore escaping exceptions from constructors and destructors.	False
ignoreThrowingFunctions	Whether to ignore noexcept specification violations on function that actually throw an exception.	True
ignoreUnknownRoutines	Whether to ignore extern or only declared routines.	True
level		required
reportNoexceptFalseViolations	Whether to report cases where the function is declared noexcept(false), but no exceptions are ever thrown (directly or indirectly).	True
target		implementation

Possible Messages

Name	Message
implicit_noexcept_spec_violation_without	Function implicitly declared noexcept(false) but no exceptions will be thrown.
noexcept_spec_missing	Explicit noexcept-specification missing.
noexcept_spec_violation_with	Exception violates function's noexcept-specification.
noexcept_spec_violation_without	Function declared noexcept(false) but no exceptions will be thrown.

AutosarC++18_03-A15.4.5

Checked exceptions that could be thrown from a function shall be specified together with the function declaration and they shall be identical in all function declarations and for all its overriders.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
autosar_exception_model	The default model for what counts as an checked / unchecked exception.	18_03
enforcement		automated
ignore_definitions_if_declaration_documented	Instead of requiring the documentation to be repeated for every declaration (including definition), with this option set, the rule only checks the non-defining declarations if at least one non-defining declaration exists.	False
level		required
match_qualified_name	Matches the fully-qualified name when comparing documented exceptions with what can actually occur. If set to 'False', this rule will accept any suffix of the qualified name of an exception class as the documentation string.	True
target		implementation
throw_marker	The command to document an exception to be thrown.	@throw

Possible Messages

Name	Message
differing_documented	Documented exceptions differ from overridden method.
document_exception	Document checked exception {} using {}.
superflous_documented	Documented exceptions {} probably never thrown.

AutosarC++18_03-A15.5.1

All user-provided class destructors, deallocation functions, move constructors, move assignment operators and swap functions shall not exit with an exception. A noexcept exception specification shall be added to these functions as appropriate.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
routine_may_except	Function must not exit with an exception.
routine_not_noexcept	Function shall be explicitly declared noexcept if appropriate.

AutosarC++18_03-A15.5.2

Program shall not be abruptly terminated. In particular, an implicit or explicit invocation of std::abort(), std::quick_exit(), std::_Exit(), std::terminate() shall not be done.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
allowed_exceptions	Exceptions that are allowed to escape from destructors.	['bad_alloc', 'bad_cast', 'failure', 'runtime_error', 'system_error']
blacklist	Dictionary of header globbing to [list of] function name globbing(s) of forbidden functions.	dict(...)
constructors	Whether to consider constructors.	False
destructors	Whether to consider destructors.	True
enforcement		partially automated
generateViolationPath	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignoreConstructorDestructor	Whether to ignore escaping exceptions from constructors and destructors.	False
ignoreUnknownRoutines	Whether to ignore extern or only declared routines.	False
inspectAtExitHandlers	Whether to also inspect at_exit() handlers-functions.	True
inspectThreadMain	Whether to also inspect thread main functions.	True
level		required
reportOnlyOneExceptionPerFunction	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
target		implementation

Possible Messages

Name	Message
exceptionEscapingConstructor	Escaping exception from constructor.
exceptionEscapingDestructor	Escaping exception from destructor.
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionEscapingMain	Uncaught exception escaping from main or additional entry point
exceptionSpecificationViolation	Exception violates function's exception-specification.
forbiddenLibfuncCall	Call to forbidden function.

AutosarC++18_03-A15.5.3

The terminate() function shall not be called implicitly.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
allowed_exceptions	Exceptions that are allowed to escape from destructors.	['bad_alloc', 'bad_cast', 'failure', 'runtime_error', 'system_error']
constructors	Whether to consider constructors.	False
destructors	Whether to consider destructors.	True
enforcement		automated
generateViolationPath	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignoreConstructorDestructor	Whether to ignore escaping exceptions from constructors and destructors.	True
ignoreThrowingFunctions	Whether to ignore noexcept specification violations on function that actually throw an exception.	False
ignoreUnknownRoutines	Whether to ignore extern or only declared routines.	False
inspectAtExitHandlers	Whether to also inspect at_exit() handlers-functions.	True
inspectThreadMain	Whether to also inspect thread main functions.	True
level		required
reportNoexceptFalseViolations	Whether to report cases where the function is declared noexcept(false), but no exceptions are ever thrown (directly or indirectly).	False
reportOnlyOneExceptionPerFunction	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
required	Dict which lists required operations per resource. The mapping gives each case a description which maps to a dict for key "Required_Functions", "Resource_Parameter_Empty".	dict(...)
resources	Configuration of resources and operations on them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)
target		implementation

Possible Messages

Name	Message
exceptionEscapingConstructor	Escaping exception from constructor.
exceptionEscapingDestructor	Escaping exception from destructor.
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionEscapingMain	Uncaught exception escaping from main or additional entry point
exceptionSpecificationViolation	Exception violates function's exception-specification.
implicitNoexceptSpecViolationWithout	Function implicitly declared noexcept(false) but no exceptions will be thrown.
noexceptSpecViolationWith	Exception violates function's noexcept-specification.
noexceptSpecViolationWithout	Function declared noexcept(false) but no exceptions will be thrown.
possiblyRequiredOperation	This thread is possibly joinable on destructor call
requiredOperation	This thread is joinable on destructor call

AutosarC++18_03-A16.0.1

The pre-processor shall only be used for file inclusion and include guards.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
accept_conditional_includes	Whether to accept #ifs for conditional includes.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_macro_definition	Macros are only allowed for include guards
line_directive	The pre-processor shall only be used for file inclusion and include guards.
object_macro_definition	Macros are only allowed for include guards
pp_if	Conditional compilation is only allowed for include guards
pragma	The pre-processor shall only be used for file inclusion and include guards.
undef	The pre-processor shall only be used for file inclusion and include guards.

AutosarC++18_03-A16.2.1

The ', ", /*, //, \ characters shall not occur in a header file name or in #include directive.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
check_quotation_mark	Whether system includes should be checked for containing a ".	True
enforcement		automated
forbidden	The substrings to check for. " will be added for system-includes.	set(['//', '\\\\', """", /*'])
level		required
target		implementation

Possible Messages

Name	Message
nonstandard_include_character	Non-standard character in #include directive

AutosarC++18_03-A16.6.1

#error directive shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
error	Use of #error

AutosarC++18_03-A16.7.1

The #pragma directive shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pragma	Use of #pragma

AutosarC++18_03-A17.0.1

Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
additional_reserved_identifiers	Names listed here are seen as violations as well.	set(['assert', 'errno', 'defined'])
allow_reserved_identifier_as_include_guard	If True, macros used for include guards are not checked for being a reserved identifier.	False
check_for_keyword	Whether macro names being keywords should be reported as well.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
macro_having_reserved_name	Definition of reserved identifier or standard library element
undef_of_reserved_name	#undef of reserved identifier or standard library element

AutosarC++18_03-A17.6.1

Non-standard entities shall not be added to standard namespaces.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
std_specialization_blacklist		('std::integral_constant', 'std::is_void', 'std::is_null_pointer', 'std::is_integral', 'std::is_floating_point', 'std::is_array', 'std::is_enum', 'std::is_union', 'std::is_class', 'std::is_function', 'std::is_pointer', 'std::is_lvalue_reference', 'std::is_rvalue_reference', 'std::is_member_object_pointer', 'std::is_member_function_pointer', 'std::is_fundamental', 'std::is_arithmetic', 'std::is_scalar', 'std::is_object', 'std::is_compound', 'std::is_reference', 'std::is_member_pointer', 'std::is_const', 'std::is_volatile', 'std::is_trivial', 'std::is_trivially_copyable', 'std::is_standard_layout', 'std::is_pod', 'std::is_literal_type', 'std::has_unique_object_representations', 'std::is_empty', 'std::is_polymorphic', 'std::is_abstract', 'std::is_final', 'std::is_aggregate', 'std::is_signed', 'std::is_unsigned', 'std::is_constructible', 'std::is_trivially_constructible', 'std::is_nothrow_constructible', 'std::is_default_constructible', 'std::is_trivially_default_constructible', 'std::is_nothrow_default_constructible', 'std::is_copy_constructible', 'std::is_trivially_copy_constructible', 'std::is_nothrow_copy_constructible', 'std::is_move_constructible', 'std::is_trivially_move_constructible', 'std::is_nothrow_move_constructible', 'std::is_assignable', 'std::is_trivially_assignable', 'std::is_nothrowAssignable', 'std::is_copyAssignable', 'std::is_trivially_copyAssignable', 'std::is_nothrow_copyAssignable', 'std::is_moveAssignable', 'std::is_trivially_moveAssignable', 'std::is_nothrow_moveAssignable', 'std::is_destructible', 'std::is_trivially_destructible', 'std::is_nothrow_destructible', 'std::has_virtual_destructor', 'std::is_swappable_with', 'std::is_swappable', 'std::is_nothrow_swappable_with', 'std::is_nothrow_swappable', 'std::alignment_of', 'std::rank', 'std::extent', 'std::is_same', 'std::is_base_of', 'std::is_convertible', 'std::is_nothrow_convertible', 'std::is_invocable', 'std::is_invocable_r', 'std::is_nothrow_invocable', 'std::is_nothrow_invocable_r', 'std::remove_cv', 'std::remove_const', 'std::remove_volatile', 'std::add_cv', 'std::add_const', 'std::add_volatile', 'std::remove_reference', 'std::add_lvalue_reference', 'std::add_rvalue_reference', 'std::remove_pointer', 'std::add_pointer', 'std::make_signed', 'std::make_unsigned', 'std::remove_extent', 'std::remove_all_extents', 'std::aligned_storage', 'std::aligned_union', 'std::decay', 'std::remove_cvref', 'std::enable_if', 'std::conditional', 'std::common_type', 'std::underlying_type', 'std::result_of', 'std::invoke_result', 'std::void_t', 'std::conjunction', 'std::disjunction', 'std::negation', 'std::endian', 'std::unary_function', 'std::binary_function')
std_specialization_whitelist		('std::common_type',)
target		implementation

Possible Messages

Name	Message
std_extension	Invalid addition to std namespace
std_specialization	Invalid std template specialization

AutosarC++18_03-A18.0.1

The C library shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
c_lib_header	Include <{}> instead of <{}>.
cpp_lib_header_with_suffix	Include <{}> instead of <{}>.

AutosarC++18_03-A18.0.2

The error state of a conversion from string to a numeric value shall be checked.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
check_functions	Fully qualified names of functions to check the state of the input stream.	['std::basic_ios::fail', 'std::ios_base::fail']
enforcement		automated
functions_to_check	Fully qualified names of functions after which a call to one of the check functions is required	['std::basic_istream::operator>>']
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	stdlib
symbols	Names of symbols which are forbidden.	['atof', 'atoi', 'atol', 'atoll']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.
missing_basic_ios_fail	Use basic_ios::fail() after reading from input streams.

AutosarC++18_03-A18.0.3

The library <clocale> (locale.h) and the setlocale function shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	locale
symbols	Names of symbols which are forbidden.	['setlocale']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++18_03-A18.1.1

C-style arrays should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
c_array_used	C-style arrays should not be used.

AutosarC++18_03-A18.1.2

The std::vector<bool> specialization shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
type_used	The type vector<bool> shall not be used.

AutosarC++18_03-A18.1.3

The std::auto_ptr type shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
type_used	The std::auto_ptr shall not be used.

AutosarC++18_03-A18.1.4

A pointer pointing to an element of an array of objects shall not be passed to a smart pointer of single object type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
type_used	{} shall not refer to an array type.

AutosarC++18_03-A18.1.6

All std::hash specializations for user-defined types shall have a noexcept function call operator.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
specialization_of_hash_except	std::hash specialization shall have noexcept call operator.

AutosarC++18_03-A18.5.1

Functions malloc, calloc, realloc and free shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_user_new_delete_operator	Whether to allow the library functions inside user defined new/delete operator overloads.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
stdlib_memory_function_call	Dynamic memory management function from <stdlib.h> is called.

AutosarC++18_03-A18.5.2

Operators new and delete shall not be called explicitly.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_for_types_and_subtypes	A list of fully-qualified type names for which the new and delete operators are allowed. This might be necessary if you are working with 3rd party libraries like Qt.	[]
enforcement		partially automated
level		required
target		implementation

Possible Messages

Name	Message
new_delete_call	Operators new and delete shall not be called explicitly.

AutosarC++18_03-A18.5.3

The form of delete operator shall match the form of new operator used to allocate the memory.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. See the base class for more options.	dict(...)
target		implementation

Possible Messages

Name	Message
possible_wrong_release	Resource possibly released using wrong function (allocation used {0})
wrong_release	Resource released using wrong function (allocation used {0})

AutosarC++18_03-A18.5.4

If a project has sized or unsized version of operator "delete" globally defined, then both sized and unsized versions shall be defined.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
only_sized	Only the sized version of operator delete is defined.
only_unsized	Only the unsized version of operator delete is defined.

AutosarC++18_03-A18.5.8

Objects that do not outlive a function shall have automatic storage duration.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
allow_smart_ptr_from_function_return	Whether to allow a smart pointer returned from a function as a local variable in a function. Calling the smart pointer constructors or the helper functions std::make_{shared,unique,..} will remain a violation.	True
enforcement		partially automated
level		required
target		implementation

Possible Messages

Name	Message
could_be_scoped	Local objects shall be allocated on the stack.

AutosarC++18_03-A18.9.1

The std::bind shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	functional
symbols	Names of symbols which are forbidden.	['std::bind']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	False

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity std::bind from <{}>.

AutosarC++18_03-A18.9.2

Forwarding values to other functions shall be done via: (1) std::move if the value is an rvalue reference, (2) std::forward if the value is forwarding reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
forwarding_forwarding_reference	Use std::forward if the value is a forwarding reference.
forwarding_rvalue_reference	Use std::move if the value is a rvalue reference.

AutosarC++18_03-A18.9.3

The std::move shall not be used on objects declared const or const&.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
std_move_const	Call to std::move with argument declared const/const&.

AutosarC++18_03-A21.8.1

Arguments to character-handling functions shall be representable as an unsigned char.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
functions	Names of functions that require an unsigned char cast.	['isalnum', 'isalpha', 'islower', 'isupper', 'isdigit', 'isxdigit', 'iscntrl', 'isgraph', 'isspace', 'isblank', 'isprint', 'ispunct', 'tolower', 'toupper']
level		required
target		implementation

Possible Messages

Name	Message
missing_uchar_cast	Missing explicit cast to unsigned char.
missing_uchar_cast_on_routine_literal	Missing explicit cast to unsigned char.

AutosarC++18_03-A23.0.1

An iterator shall not be implicitly converted to const_iterator.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
implicit_const_iterator	An iterator shall not be implicitly converted to const_iterator.

AutosarC++18_03-A26.5.1

Pseudorandom numbers shall not be generated using std::rand().

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
blacklist	Dictionary of header globbing to [list of] function name globbing(s) of forbidden functions.	dict{...}
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
forbidden_libfunc_call	Call to forbidden function.

AutosarC++18_03-A26.5.2

Random number engines shall not be default-initialized.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
additional_blacklist	Sequence of additional forbidden class names where calling the default constructor is forbidden.	{}
blacklist	Sequence of forbidden class names where calling the default constructor is forbidden.	{'std::minstd_rand0', 'std::mt19937', 'std::mt19937_64', 'std::ranlux24_base<T>', 'std::ranlux48_base', 'std::ranlux24', 'std::ranlux48', 'std::knuth_b', 'std::default_random_engine', 'std::linear_congruential_engine', 'std::mersenne_twister_engine', 'std::subtract_with_carry_engine'}
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
forbidden_default_constructor_call	Call to forbidden default constructor.

AutosarC++18_03-A27.0.4

C-style strings shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
c_string_used	C-style strings should not be used.

AutosarC++18_03-M0.1.1

There shall be no unreachable code.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_dead_if_branch_with_const_literal_condition	Whether bodies of if-branches with constant literal conditions should be reported.	True
target		implementation

Possible Messages

Name	Message
unreachable_code	Unreachable code

AutosarC++18_03-M0.1.2

A project shall not contain infeasible paths.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_do_while_false	Whether do ... while(0) should be reported.	True
report_while_true	Whether while(1) ... should be reported.	True
target		implementation

Possible Messages

Name	Message
dead_false_branch	Condition is always true: Else branch is dead
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Parameter condition is always true: Else branch is dead
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Parameter comparison to NULL is always true: Else branch is dead
dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Parameter comparison to NULL is always false: Then branch is dead
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Parameter condition is always false: Then branch is dead
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Condition is always false: Then branch is dead
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Variable condition is always true: Else branch is dead
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Variable condition is always false: Then branch is dead
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead

AutosarC++18_03-M0.1.3

A project shall not contain unused variables.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
ignore_types	Variables of types named here are ignored in this check. Globbing patterns are supported.	[]
level		required
only_check_unit_locals	Whether only global static variables and local variables should be checked.	False
report_global_constants	Whether unused global constants should be reported.	False
report_undefined_variables	Whether only-declared variables should be reported.	True
target		implementation
treat_initialization_as_use	Whether an explicit initialization should be considered a use of the variable.	True
treat_side_effect_constructors_as_use	Whether variables should be seen as used if they are of a class type and initialized through a call to a constructor having a side-effect, e.g. std::lock_guard	False

Possible Messages

Name	Message
unused_field	Unused field
unused_variable	Unused variable

AutosarC++18_03-M0.1.4

A project shall not contain non-volatile POD variables having only one use.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_if_address_taken	Allow using a variable only once if that use involves taking the address of the variable.	False
enforcement		automated
level		required
report_fields	Select whether fields used only once should be reported as well.	True
target		implementation

Possible Messages

Name	Message
field_referenced_only_once	{ } referenced only once
unreferenced_initialized_field	{ } initialized but not referenced
unreferenced_initialized_variable	{ } initialized but not referenced
variable_used_only_once	{ } referenced only once

AutosarC++18_03-M0.1.8

All functions with void return type shall have external side effect(s).

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
assume_effect_when_undefined	Whether all calls to undefined [external] functions should be seen as having a side-effect.	True
enforcement		automated
exceptions	Names of functions that should be excluded from the check.	main
exclude_constructors	If True, tolerate constructors with no side-effect.	False
exclude_virtual_destructors	If True, tolerate virtual destructors with no side-effect.	False
level		required
target		implementation

Possible Messages

Name	Message
void_func_without_side_effect	Void function has no external side-effect

AutosarC++18_03-M0.1.9

There shall be no dead code.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
allow_void_var	Whether {void}var; should be allowed or reported.	True
assume_effect_for_all_calls	Whether all calls should be seen as having a side-effect.	False
assume_effect_when_undefined	Whether all calls to undefined (external) functions should be seen as having a side-effect.	True
enforcement		automated
level		required
report_dead_initializations	Whether initialisations may be reported as dead.	True
report_do_while_false	Whether do ... while[0] should be reported.	True
report_while_true	Whether while[1] ... should be reported.	True
target		implementation
tolerate_void_cast	Whether a {void} cast is accepted.	True

Possible Messages

Name	Message
conditional_unused_def	Result of assignment is not used along some path(s) (disabled)
dead_false_branch	Redundant code, condition is always true
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Redundant code, parameter condition is always true
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Redundant code, parameter comparison to NULL is always true
dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Redundant code, parameter comparison to NULL is always false
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Redundant code, parameter condition is always false
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Redundant code, condition is always false
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Redundant code, variable condition is always true
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Redundant code, variable condition is always false
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead
init_used_in_other_isr	Initialization is only used in some interrupt handler
no_effect	Non-null statement without side-effect
removable_declaration	Declaration can be removed
removable_statement	Statement can be removed
unused_def	Dead (redundant) code: result of assignment is not used
unused_init	Unused initialization
used_in_other_isr	Result of assignment is only used in some interrupt handler

AutosarC++18_03-M0.1.10

Every defined function shall be called at least once.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
ignore_copy_assignment_operators	Whether uncalled copy assignment operators are allowed. Other rules like the "rule of three" might require copy assignment operators as part of good class design, even if there are currently no callers of the copy assignment operator.	False
ignore_copy_constructors	Whether uncalled copy constructors are allowed. Other rules like the "rule of three" might require copy constructors as part of good class design, even if there are currently no callers of the copy constructor.	False
level		required
only_check_internal_or_private_methods	Whether only function with internal linkage or private functions should be checked.	False
only_check_static_or_private_methods	Whether only static functions or private functions should be checked.	False
only_check_unit_locals	Whether only global static functions should be checked.	False
target		implementation

Possible Messages

Name	Message
defined_function_not_called	Defined function is not called

AutosarC++18_03-M0.2.1

An object shall not be assigned to an overlapping object.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
assignment_to_overlapping	An object shall not be assigned to an overlapping object

AutosarC++18_03-M0.3.1

Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
abstract_interpretation_div_by_zero	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
abstract_interpretation_overflow	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
argument_checks	Configuration of [dis]allowed values in addition to what iranalysis.config provides. This is a dict key -> dict with keys 'Functions', 'Parameter', and 'Allowed'/'Disallowed'. The parameter number counting starts at 0. Arguments passed in for this parameters are checked against the value specification given with '[Dis]allowed'.	dict(...)
enforcement		automated
execute_checks_covered_in_other_rules	Whether iranalysis-based checks covered by other Misra-C++:2008 rules should be executed here as well.	False
forbidden	Dict which lists forbidden operations per resource. The mapping gives each case a description which maps to a dict for key "Forbidden_Functions", "Mode_Parameter", "Mode".	dict(...)
level		required
modes	Describes when an allocating call opens a file in read or write mode. This is a dict resource -> dict with keys "Resource_Parameter", "Check_Write_Write", and "Modes" (being a dict with key "Read" and "Write")	dict(...)
perform_global_analysis	Whether global analysis (using iranalysis) should be done, or just a local analysis.	True
report_dead_initializations	Whether initialisations may be reported as dead.	True
report_do_while_false	Whether do ... while[0] should be reported.	True
report_while_true	Whether while[1] ... should be reported.	True
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)
target		implementation

Possible Messages

Name	Message
argumentViolation	Argument not within allowed values
arithmetic_out_of_bounds	Pointer arithmetic on {1} creates pointer outside array bounds of {0}
asgn_not_an_enumerator	Assignment of value which does not correspond to an enumerator
asgn_out_of_enum_range	Assignment of value outside the range of the enumerators
cast_overflow	Cast on result of arithmetic computation may cause overflow
cast_truncate	Cast may truncate value
cast_underflow	Cast on result of arithmetic computation may cause underflow
conditional_unused_def	Result of assignment is not used along some path(s)
deadCatch	Dead exception handler
dead_false_branch	Condition is always true: Else branch is dead
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Parameter condition is always true: Else branch is dead
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Parameter comparison to NULL is always true: Else branch is dead

dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Parameter comparison to NULL is always false: Then branch is dead
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Parameter condition is always false: Then branch is dead
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Condition is always false: Then branch is dead
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Variable condition is always true: Else branch is dead
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Variable condition is always false: Then branch is dead
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead
division_by_zero	Division by zero
double_free	Dynamic memory released here was already released earlier
escaping_address	Escaping address of local variable
forbidden_operation	The way in which this resource was allocated forbids this operation
init_used_in_other_isr	Initialization is only used in some interrupt handler
memory_leak	Call allocates leaking memory
mode_conflict	Same file used for both reading and writing
modulo_by_zero	Modulo by zero
null_deref	Pointer is NULL at dereference
out_of_bounds	Access into array is out of bounds
overflow	Arithmetic computation may cause overflow
pass_as_pointer_to_const_param	Passing uninitialized variable by pointer as function parameter with pointer-to-const type
possible_argumentViolation	Argument possibly not within allowed values
possible_division_by_zero	Possible division by zero
possible_double_free	Dynamic memory released here possibly already released earlier
possible_indirect_out_of_bounds	Pointer-indirect access through {1} might be out of bounds accessing {0}
possible_memory_leak	Call allocates possibly leaking memory
possible_mode_conflict	Same file possibly used for both reading and writing
possible_modulo_by_zero	Possible modulo by zero
possible_null_deref	Pointer may be NULL at dereference
possible_out_of_bounds	Access into array might be out of bounds
possible_stack_free	{0} possibly released by call to {0} is a stack object
possible_uninit	Use of possibly uninitialized variable
possible_unrelated_ptr_comparison	Comparing possibly unrelated pointers
possible_unrelated_ptr_subtraction	Subtracting possibly unrelated pointers
possible_use_after_free	Dynamic memory possibly used after it was previously released

possible_wrong_release	Resource possibly released using wrong function (allocation used {0})
possiblyEscapingAddress	Possibly escaping address of local variable (as target of {1})
possiblyForbiddenOperation	The way in which this resource was allocated possibly forbids this operation
stack_free	{ } released by call to { } is a stack object
underflow	Arithmetic computation may cause underflow
uninit	Use of uninitialized variable
unrelated_ptr_comparison	Comparing unrelated pointers
unrelated_ptr_subtraction	Subtracting unrelated pointers
unused_def	Result of assignment is not used
unused_init	Unused initialization
use_after_free	Dynamic memory used after it was previously released
used_in_other_isr	Result of assignment is only used in some interrupt handler
wrong_release	Resource released using wrong function (allocation used {0})

AutosarC++18_03-M0.3.2

If a function generates error information, then that error information shall be tested.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
inspect_template_instances	Whether calls in template instances should be reported.	False
level		required
relevant_functions	If provided, only calls to these functions are inspected.	[]
target		implementation

Possible Messages

Name	Message
discarded_error_with_entity	Integral return value (possibly error code) discarded.
discarded_error_without_entity	Integral return value (possibly error code) discarded.

AutosarC++18_03-M0.4.2

Use of floating-point arithmetic shall be documented.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
float_arithetic	Use of floating-point arithmetic

AutosarC++18_03-M2.7.1

The character sequence /* shall not be used within a C-style comment.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_cpp_comments	Whether to look for /* in C++-style comments as well.	False
enforcement		automated
files_to_check	Files to apply this check to (Primary_File / User_Include_File / System_Include_File)	{'Primary_File', 'User_Include_File'}
level		required
target		implementation

Possible Messages

Name	Message
nested_c_comment	C-style comment containing /* sequence.

AutosarC++18_03-M2.10.1

Different identifiers shall be typographically unambiguous.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
distinguish_c_name_spaces	Whether to allow similar identifiers in different C name spaces (type tag/label/member/ordinary identifier)	True
distinguish_macro_names	Whether to allow macros to same similar names as other identifiers.	False
distinguish_type_names	Whether to allow types to have similar names as other identifiers. For example, a variable declaration "MyClass myClass;" is allowed by this option. Note: unlike the option 'distinguish_c_name_spaces', this option also distinguishes typedef names, not just struct/class tag names.	False
enforcement		automated
exclude_system_headers	Whether to exclude identifiers appearing in system headers	False
level		required
normalizations	Which pairs of characters should be seen as ambiguous	[('0', 'O'), ('1', 'l'), ('l', '1'), ('i', 'I'), ('5', 'S'), ('2', 'Z'), ('8', 'B'), ('rn', 'm'), ('n', 'h'), ('_', '_')]
target		architecture/design/implementation

Possible Messages

Name	Message
ambiguous	Identifiers are typographically ambiguous
casing	Identifiers only differ in casing

AutosarC++18_03-M2.13.2

Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
octal_escape_sequence	Use of octal escape sequence.
octal_literal	Use of octal literal.

AutosarC++18_03-M2.13.3

A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_only_hex_and_octal	Whether only hex and octal literals should be checked.	True
consider_conversion	Selects whether literals with an implicit conversion from signed to unsigned are reported.	False
consider_pp_expressions	Whether literals in #if conditions are checked.	False
enforcement		automated
level		required
report_all_unsuffixed_hex_and_octal	Whether hex and octal literals should always be reported when they are unsuffixed, independent of the environment	False
target		architecture/design/implementation

Possible Messages

Name	Message
missing_hex_suffix	Hexadecimal and octal constants require "U" suffix
missing_u_suffix	Constant of unsigned type requires "U" suffix

AutosarC++18_03-M2.13.4

Literal suffixes shall be upper case.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
lowercase_suffix	Literal suffix should be upper case

AutosarC++18_03-M3.1.2

Functions shall not be declared at block scope.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
local_function_declaration	Functions shall not be declared at block scope.

AutosarC++18_03-M3.2.1

All declarations of an object or function shall have compatible types.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_parameter_type_qualifier_differences	If enabled, do not report a violation for different top-level type qualifiers in parameters: int f(int p); int f(const int p) {} Top-level qualifiers are only relevant for the implementation of the function and have no effect on callers, so it is not necessary to replicate them in the header file.	False
check_parameter_types	Whether parameter types should be compared	False
check_undefined	Whether only-declared routines and variables should also be checked.	True
enforcement		automated
level		required
require_exact_match	Whether to check for identical or compatible types (for routine return types).	False
target		implementation

Possible Messages

Name	Message
decl_return_type_mismatch	Return type at declarations differs
decl_type_mismatch	Type at declarations differs
parameter_type_mismatch	Parameter type at definition differs from type at declaration
return_type_mismatch	Return type at definition differs from type at declaration
type_mismatch	Type at definition differs from type at declaration

AutosarC++18_03-M3.2.2

The One Definition Rule shall not be violated.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
class_struct_difference	{}
different_enumerators	{}
different_field_types	{}
different_fields	{}
general_odrViolation	{}

AutosarC++18_03-M3.2.3

A type, object or function that is used in multiple translation units shall be declared in one and only one file.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_only_one_decl	Enables message if there are multiple declarations but all in same file.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_declared_in_multiple_files	Function declared in more than one file.
function_declared_multiple_times	Function declared more than once.
type_declared_in_multiple_files	Type declared in more than one file.
type_declared_multiple_times	Type declared more than once.
variable_declared_in_multiple_files	Variable declared in more than one file.
variable_declared_multiple_times	Variable declared more than once.

AutosarC++18_03-M3.2.4

An identifier with external linkage shall have exactly one definition.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_undefined_specialized_template	Select whether undefined templates should be reported if specializations of them exist.	True
allow_unselected_declarations	Select whether to report declarations with no definition if they are unused.	True
allowed_undefined_functions	Regular expressions for functions which are tolerated without a definition, e.g. compiler intrinsics.	[_.*]
allowed_undefined_types	Regular expressions for types which are tolerated without a definition.	[_.*]
allowed_undefined_variables	Regular expressions for variables which are tolerated without a definition.	[_.*]
check_composite_types	Check class/struct/union types for having no definition even if they have no external linkage.	False
check_types_only_with_external_linkage	Whether the types should only be checked when they have external linkage	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
composite_defined_multiple_times	Composite type has multiple definitions.
enum_defined_multiple_times	Enumeration type has multiple definitions.
function_defined_multiple_times	Type, object or function has multiple definitions.
undefined_function	Type, object or function without definition
undefined_type	Type without definition
undefined_variable	Type, object or function without definition
variable_defined_multiple_times	Type, object or function has multiple definitions.

AutosarC++18_03-M3.3.2

If a function has internal linkage then all re-declarations shall include the static storage class specifier.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_decl_missing_static	Declaration should use static storage class specifier
function_def_missing_static	Definition should use static storage class specifier
variable_decl_missing_static	Declaration should use static storage class specifier
variable_def_missing_static	Definition should use static storage class specifier

AutosarC++18_03-M3.4.1

An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_moving_to_other_primary_file	When a variable is only used in a single file X, but currently implemented in a different file Y, this controls whether the check suggests to move it into X and make it static there.	False
check_classes	Whether to report structs/classes/unions which are only used in a single function or file	True
check_loop_counter	Whether to report for-loop counters being declared before the loop that could be declared in the for-init part instead from C99 on	False
enforcement		automated
exclude_function_locals	If True, variables that could even be function-local are not reported.	True
exclude_undefined	Whether only-declared symbols should be reported as well	True
level		required
move_global_const_into_function	Controls suggestions for global constants that could be declared locally in a function.	True
only_check_unit_locals	Whether only global static variables should be checked.	False
target		implementation

Possible Messages

Name	Message
locality_block	{ } can be declared in a more local scope.
locality_file	{ } can be declared locally in primary file.
locality_function	Global { } can be declared inside function.
locality_loop_init	{ } can be declared in the for-loop's initialization.
var_file_static	{ } can be declared static in primary file.

AutosarC++18_03-M3.9.1

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_missing_qualifiers	If True, tolerate differences in the use of explicit namespace/class qualifiers.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
parameter_type_tokens_mismatch	Type of redeclaration is not token-for-token identical
return_type_tokens_mismatch	Type of redeclaration is not token-for-token identical
variable_type_tokens_mismatch	Type of redeclaration is not token-for-token identical

AutosarC++18_03-M3.9.3

The underlying bit representations of floating-point values shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
float_bit_representation	Use of bit representation of a float value.

AutosarC++18_03-M4.5.1

Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_sizeof	Allows the use of sizeof(bool_var). The Misra rule forbids all uses of expressions of type bool except for those explicitly allowed; and does not allow sizeof. However this is likely an oversight; there is no good reason to prevent the use of sizeof(bool_var). This option also allows alignof(bool_var) in C++11.	False
allow_unevaluated_operator	Allows the use of boolean expressions in contexts where the expression is not evaluated. As of C++17, these are: typeid(bool_var), noexcept(bool_var) and decltype(bool_var). sizeof(bool_var) is controlled by the separate option 'allow_sizeof'.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bool_operand_in_bad_operator	Use of boolean operand in '{}' operator
bool_operand_outside_logical_and_relational_op	Use of boolean operand with integral promotion

AutosarC++18_03-M4.5.3

Expressions with type [plain] char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
char_operand_outside_comparison	Use of character operand in forbidden context

AutosarC++18_03-M4.10.1

NULL shall not be used as an integer value.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
null_as_int	Use of NULL as integer value

AutosarC++18_03-M4.10.2

Literal zero (0) shall not be used as the null-pointer-constant.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
require_nullptr	Require C++11 nullptr instead of macro NULL or literal zero (0).	False
target		implementation

Possible Messages

Name	Message
zero_as_null	Use of literal zero (0) as null-pointer-constant, use {} instead

AutosarC++18_03-M5.0.2

Limited dependence should be placed on C++ operator precedence rules in expressions.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_algebraic_order	Whether parens can be omitted for multiplicative operators as operands in additive operators	False
allow_redundant_parentheses_in_logical_operator	Allow redundant parentheses around unary operators when they appear as operand of '&&' or ' '. This option can be used to avoid a conflict with MisraC++-5.2.1, which requires parentheses in these instances.	True
enforcement		automated
level		required
report_unnecessary_parentheses	Controls whether unnecessary use of parentheses on the right side of assignments or around unary operators are reported.	True
target		implementation

Possible Messages

Name	Message
assignment_rhs_in_parens	No parentheses required for right-hand side of assignment
missing_parens_depends_on_precedence	Parentheses required to avoid dependence on precedence rules
unary_op_in_parens	No parentheses required for unary operator

AutosarC++18_03-M5.0.3

A value expression shall not be implicitly converted to a different underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes], [SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
target		implementation
type_category	Selection of type categories to consider for type changes inside same category.	[SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Result of cvalue expression implicitly converted to different underlying type
cast_changes_type_inside_category	Result of cvalue expression implicitly converted to different underlying type

AutosarC++18_03-M5.0.4

An implicit integral conversion shall not change the signedness of the underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_constant_conversions	If this option is enabled, the rule is relaxed to allow implicit conversions of constant integer expressions whenever the constant value fits into the target type.	True
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
report_signed_to_unsigned	Whether conversion from signed to unsigned shall be reported.	True
report_unsigned_to_signed	Whether conversion from unsigned to signed shall be reported.	True
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_from_signed_to_unsigned	Implicit integral conversion changes signedness of underlying type
cast_from_unsigned_to_signed	Implicit integral conversion changes signedness of underlying type

AutosarC++18_03-M5.0.5

There shall be no implicit floating-integral conversions.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	<code>[[[SignedTypes, UnsignedTypes], [FloatingTypes]], [[FloatingTypes], [SignedTypes, UnsignedTypes]]]</code>
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Implicit floating-integral conversion

AutosarC++18_03-M5.0.6

An implicit integral or floating-point conversion shall not reduce the size of the underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
report_narrowing	Whether narrowing casts should be reported.	True
report_widening	Whether widening casts should be reported.	False
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
size_source_category	Selection of type categories of the operand being cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
size_target_category	Selection of type categories appearing as target types of the cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
narrowing_cast	Implicit conversion reduces size of underlying type
widening_cast	Conversion to larger type

AutosarC++18_03-M5.0.7

There shall be no explicit floating-integral conversions of a cvalue expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, CharacterTypes], [FloatingTypes]], [[FloatingTypes], [SignedTypes, UnsignedTypes, CharacterTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Explicit floating-integral conversion of cvalue expression

AutosarC++18_03-M5.0.8

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
report_narrowing	Whether narrowing casts should be reported.	False
report_widening	Whether widening casts should be reported.	True
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
size_source_category	Selection of type categories of the operand being cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
size_target_category	Selection of type categories appearing as target types of the cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
narrowing_cast	Conversion to smaller type
widening_cast	Explicit conversion increases size of underlying type of cvalue expression

AutosarC++18_03-M5.0.9

An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
report_signed_to_unsigned	Whether conversion from signed to unsigned shall be reported.	True
report_unsigned_to_signed	Whether conversion from unsigned to signed shall be reported.	True
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_from_signed_to_unsigned	Explicit conversion changes signedness of underlying type of cvalue expression
cast_from_unsigned_to_signed	Explicit conversion changes signedness of underlying type of cvalue expression

AutosarC++18_03-M5.0.10

If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitop_small_without_cast	Bitwise operator requires cast to underlying type on result

AutosarC++18_03-M5.0.11

The plain char type shall only be used for the storage and use of character values.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
char_used_numerically	Plain char datatype used for non-character data.

AutosarC++18_03-M5.0.12

signed char and unsigned char type shall only be used for the storage and use of numeric values.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
numeric_char_used_as_character	Signed/unsigned char datatype used for character data.

AutosarC++18_03-M5.0.14

The first operand of a conditional-operator shall have type bool.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
nonbool_conditional_operator_condition	Condition must have type bool

AutosarC++18_03-M5.0.15

Array indexing shall be the only form of pointer arithmetic.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
indexing_only_on_identifiers	Report array indexing on pointers only for variables (`ptr[i]`), not for other pointer expressions (e.g. `get_ptr()[i]`). This option is meant to suppress the violations introduced by the BAUHAUS-12021 bugfix in version 6.9.6.	False
level		required
target		implementation

Possible Messages

Name	Message
array_indexing_on_pointer	Array indexing only allowed for arrays
pointer_arithmetic	Pointer arithmetic not allowed
pointer_increment_decrement	Pointer arithmetic not allowed

AutosarC++18_03-M5.0.16

A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
arithmetic_on_nonarray	Pointer arithmetic on {1} operates on non-array target {0}
arithmetic_out_of_bounds	Pointer arithmetic on {1} creates pointer outside array bounds of {0}
out_of_bounds	Access into array is out of bounds
possible_indirect_out_of_bounds	Pointer-indirect access through {1} might be out of bounds accessing {0}
possible_out_of_bounds	Access into array might be out of bounds

AutosarC++18_03-M5.0.17

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
possible_unrelated_ptr_subtraction	Subtracting possibly unrelated pointers
unrelated_ptr_subtraction	Subtracting unrelated pointers

AutosarC++18_03-M5.0.18

`>`, `>=`, `<`, `<=` shall not be applied to objects of pointer type, except where they point to the same array.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
possible_unrelated_ptr_comparison	Comparing possibly unrelated pointers
unrelated_ptr_comparison	Comparing unrelated pointers

AutosarC++18_03-M5.0.20

Non-constant operands to a binary bitwise operator shall have the same underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitop_with_different_operand_types	Non-constant operands of bitwise operator differ in underlying type
shortcut_bitop_with_different_operand_types	Non-constant operands of bitwise operator differ in underlying type

AutosarC++18_03-M5.0.21

Bitwise operators shall only be applied to operands of unsigned underlying type.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitop_on_enum	Bitwise operator on enum underlying type
bitop_on_signed	Bitwise operator on signed underlying type

AutosarC++18_03-M5.2.2

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cannot_cast_virtual_base	Cannot convert pointer to base class {} to pointer to derived class {} -- base class is virtual
missing_dynamic_cast_on_virtual_base	Use dynamic_cast on virtual base class

AutosarC++18_03-M5.2.3

Casts from a base class to a derived class should not be performed on polymorphic types.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cast_from_polybase_to_derived	Cast from polymorphic base class to derived class

AutosarC++18_03-M5.2.6

A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[FunctionPointerTypes], [VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes, NullPointerTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_category	Selection of type categories to consider for type changes inside same category.	[FunctionPointerTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion of function pointer to other type
cast_changes_type_inside_category	Conversion of function pointer to other function pointer type

AutosarC++18_03-M5.2.8

An object with integer type or pointer to void type shall not be converted to an object with pointer type.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, VoidPointerTypes], [ObjectPointerTypes, IncompletePointerTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, [T*](void*)x will not be reported.	True
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion from void* or integer to pointer type

AutosarC++18_03-M5.2.9

A cast should not convert a pointer type to an integral type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes], [SignedTypes, UnsignedTypes, BoolTypes, EnumTypes, CharacterTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion between pointer and integral type

AutosarC++18_03-M5.2.10

The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
forbid_all_operators	If True, forbids mixing with any kind of operator; else only with arithmetic operators.	False
level		required
target		implementation

Possible Messages

Name	Message
increment_mixed_with_operator	Increment or decrement mixed with other operators

AutosarC++18_03-M5.2.11

The comma operator, && operator and the || operator shall not be overloaded.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
invalid	Selection of disallowed operator overloads by name.	['operator&&', 'operator ', 'operator,']
invalid_kinds	Selection of disallowed operator overloads by operator kind.	[]
level		required
target		implementation

Possible Messages

Name	Message
forbidden_operator_overload	Overloaded version of comma operator or && or

AutosarC++18_03-M5.2.12

An identifier with array type passed as a function argument shall not decay to a pointer.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
decay	Array to pointer decay

AutosarC++18_03-M5.3.1

Each operand of the ! operator, the logical && or the logical || operators shall have type bool.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_in_conditions	If True, only logical operators inside conditions are checked.	False
target		implementation

Possible Messages

Name	Message
nonbool_logical_operator_operand	Operand of logical operator shall be of type bool

AutosarC++18_03-M5.3.2

The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_literals	If True, integer literals are also disallowed as operands.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unary_minus_on_unsigned	Unary minus applied to unsigned

AutosarC++18_03-M5.3.3

The unary & operator shall not be overloaded.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
invalid	Selection of disallowed operator overloads by name.	[]
invalid_kinds	Selection of disallowed operator overloads by operator kind.	[7]
level		required
target		implementation

Possible Messages

Name	Message
forbidden_operator_overload	Overloaded version of unary &

AutosarC++18_03-M5.3.4

Evaluation of the operand to the sizeof operator shall not contain side effects.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
side_effect_in_sizeof	Operand of "sizeof" shall not contain side effects

AutosarC++18_03-M5.8.1

The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_nonconst_cases	Whether a message should be printed for non-constant cases missing a range-checking if statement	False
target		implementation
use_essential_type	Whether essential type should be computed for allowed value range.	False

Possible Messages

Name	Message
excessive_shift	Right operand of shift operator has value {}, but should be in range 0 to {}
possible_excessive_shift	Right operand of shift operator should be checked to be in range 0 to {}

AutosarC++18_03-M5.14.1

The right-hand operand of a logical `&&` or `||` operator shall not contain side effects.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_volatile_access	Whether access to a volatile variable should be reported as side effect.	False
assume_effect_when_undefined	Whether all calls to undefined (external) functions should be seen as having a side-effect.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
call_with_side_effect	Call in right-hand operand of <code>&&</code> or <code> </code> may have side-effect
modifies_local_var	Right-hand operand of <code>&&</code> or <code> </code> modifies ' <code>{}</code> '
side_effect	Right-hand operand of <code>&&</code> or <code> </code> has side-effect

AutosarC++18_03-M5.17.1

The semantic equivalence between a binary operator and its assignment operator form shall be preserved.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_call_relation	If True, checks whether there is a call relation between binary and assignment version.	True
enforcement		automated
ignore_stream_operators	If True, allows definitions of <code>operator<<()</code> and <code>operator>>()</code> without the corresponding assignment operator. Note this will also allow <code>operator<<=()</code> and <code>operator>>=()</code> as they no longer have an equivalent binary form.	False
level		required
target		implementation

Possible Messages

Name	Message
missing_assignment_version	Missing overload for corresponding assignment version of operator
missing_binary_version	Missing overload for corresponding binary version of operator
missing_call_to_assignment_version	There is no call relation between this operator and its assignment version to ensure semantic equivalence
missing_call_to_binary_version	There is no call relation between this operator and its binary version to ensure semantic equivalence

AutosarC++18_03-M5.18.1

The comma operator shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_comma	Use of comma sequence operator.

AutosarC++18_03-M5.19.1

Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
const_expression_overflow	Overflow during evaluation of constant expression.

AutosarC++18_03-M6.2.1

Assignment operators shall not be used in sub-expressions.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
assignment_result_used	Assignment inside sub-expression.

AutosarC++18_03-M6.2.2

Floating-point expressions shall not be directly or indirectly tested for equality or inequality.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
float_equality_comparison	Testing a float for (in)equality.
float_indirect_equality_comparison	Indirectly testing a float for (in)equality.

AutosarC++18_03-M6.2.3

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_empty_macros	Whether a macro invocation before the ; is allowed if it expands to nothing.	False
allow_nonempty_macros	Whether a non-empty macro invocation before the ; is allowed.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
null_statement_not_isolated	Null statement not on a line by itself

AutosarC++18_03-M6.3.1

The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
loop_body_not_compound	Loop body shall be a statement sequence.
switch_body_not_compound	switch body shall be a statement sequence.

AutosarC++18_03-M6.4.1

An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if

statement.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
dangerous_else	Block or if statement required as else branch.
dangerous_then	Block required as then branch.

AutosarC++18_03-M6.4.2

All if ... else if constructs shall be terminated with an else clause.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
defensive_else	Controls if the else branch of if-cascade's should contain a suitable comment or not be empty.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
non_defensive_else	Last else in the if-cascade must not be empty or contain a comment.
unterminated_if_cascade	Last if in the if-cascade has no else.

AutosarC++18_03-M6.4.3

A switch statement shall be a well-formed switch statement.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
acceptable_end_items	This set characterizes allowed last items at the end of a switch case	('Exit_Switch', 'Throw_Expression')
acceptable_middle_items	This set characterizes allowed last items in the middle of a switch case	('Exit_Switch', 'Nondefault_Case_Label', 'Throw_Expression')
allow_as_first	Whether to allow the "default" case as the first case in a switch.	False
allow_complete_enum	Whether a switch having a case for all enumerators is tolerated without a default.	True
allow_empty_statements	Whether empty statements at the end of switch-case should be ignored.	False
allow_fallthrough_comment	Whether a comment with text "fall" and "through" is allowed to mark fallthrough.	True
allow_fallthrough_macro	Whether a macro called __fallthrough is allowed to mark fallthrough.	True
allow_if	Whether an if containing acceptable items at the end of both branches is accepted.	False
allow_noreturn_function_calls	Whether a function call to a [[noreturn]] function is considered a valid termination.	False
comment	The comment that indicates an intentional fallthrough. Can be a list of comments.	[]
disallow_multiple_compound_statements	Whether there can be more than one toplevel compound statement.	True
disallowed_nested_statements	Statement types not allowed inside compound statements within case-clause body.	None
disallowed_toplevel_statements	Statement types not allowed in case-clause body on toplevel.	('Named_Label', 'Unconditional_Branch', 'Declaration_Statement')
enforcement		automated
level		required
minimum_switch_cases	The number of cases a switch statement should at least have.	1
target		implementation

Possible Messages

Name	Message
case_label_nested	A switch label shall only be used in a switch.
default_not_last	"default" clause is not last in switch.
empty_default	Default clause should contain a statement or comment.
empty_switch	Switch has no "case" clause.
forbidden_compound_statement_in_case	At most one compound statement is allowed on toplevel in case-clauses.
forbidden_nested_statement_in_case	Statement is not allowed inside compound statement in case-clauses.
forbidden_toplevel_statement_in_case	Statement is not allowed outside a compound statement in case-clauses.
minimum_switch_cases	Switch has too little "case" clauses.
missing_default	Switch has no "default" clause.
switch_body_not_compound	A switch statement shall be a well-formed switch statement.
switch_not_well_formed	Switch has a statement before the first case label.
switch_over_bool	Switch expression is effectively Boolean.
unterminated_case	This switch-case is not terminated with {}, last statement is {}.
unterminated_empty_case	This switch-case is empty, not terminated with {}.

AutosarC++18_03-M6.4.4

A switch label shall only be used when the most closely-enclosing compound-statement is the body of a switch-statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
case_label_nested	A switch label shall only be used in a switch.

AutosarC++18_03-M6.4.5

An unconditional throw or break statement shall terminate every non-empty switch-clause.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
acceptable_end_items	This set characterizes allowed last items at the end of a switch case	('Exit_Switch', 'Throw_Expression')
acceptable_middle_items	This set characterizes allowed last items in the middle of a switch case	('Exit_Switch', 'Nondefault_Case_Label', 'Throw_Expression')
allow_empty_statements	Whether empty statements at the end of switch-case should be ignored.	False
allow_fallthrough_comment	Whether a comment with text "fall" and "through" is allowed to mark fallthrough.	True
allow_fallthrough_macro	Whether a macro called __fallthrough is allowed to mark fallthrough.	True
allow_if	Whether an if containing acceptable items at the end of both branches is accepted.	False
allow_noreturn_function_calls	Whether a function call to a [[noreturn]] function is considered a valid termination.	False
comment	The comment that indicates an intentional fallthrough. Can be a list of comments.	[]
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unterminated_case	This switch-case is not terminated with {}, last statement is {}.
unterminated_empty_case	This switch-case is empty, not terminated with {}.

AutosarC++18_03-M6.4.6

The final clause of a switch statement shall be the default clause.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_as_first	Whether to allow the "default" case as the first case in a switch.	False
allow_complete_enum	Whether a switch having a case for all enumerators is tolerated without a default.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
default_not_last	"default" clause is not last in switch.
empty_default	Default clause should contain a statement or comment.
missing_default	Switch has no "default" clause.

AutosarC++18_03-M6.4.7

The condition of a switch statement shall not have bool type.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
switch_over_bool	Switch condition shall not have bool type.

AutosarC++18_03-M6.5.2

If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
stepping_loop_uses_equality_check	Loop-counter shall not be tested with equality operator if not modified by -- or ++

AutosarC++18_03-M6.5.3

The loop-counter shall not be modified within condition or statement.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
loop_counter_modified_in_condition	Loop-counter shall not be modified within condition
modified_loop_counter	Loop-counter shall not be modified within loop body

AutosarC++18_03-M6.5.4

The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_constexpr	Allow constexpr as a constant <n>.	True
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
nonconst_loop_increment	Loop-counter shall be modified by one of: --, ++, -=n, or +=n (with constant n)

AutosarC++18_03-M6.5.5

A loop-control-variable other than the loop-counter shall not be modified within condition or expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
modified_loop_control_variable	Loop-control variable (other than counter) shall not be modified within condition or expression

AutosarC++18_03-M6.5.6

A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
nonbool_loop_control_variable	Loop-control variable (other than counter) shall have type bool

AutosarC++18_03-M6.6.1

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
goto_label_block	Label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

AutosarC++18_03-M6.6.2

The goto statement shall jump to a label declared later in the same function body.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
backwards_goto	Label referenced by a goto statement shall be declared later in same function.

AutosarC++18_03-M6.6.3

The continue statement shall only be used within a well-formed for loop.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
report_outside_for_loops	Whether to report continue statements in do..while/while loops	True
target		implementation

Possible Messages

Name	Message
continue_in_bad_loop	The continue statement shall only be used within a well-formed for loop

AutosarC++18_03-M7.1.2

A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_all_variables	Whether non-parameter variables should be checked as well.	False
enforcement		automated
ignore_forwarding_references	Whether to allow C++11 forwarding references (universal references) even if all template instances would also work with a const lvalue reference.	True
ignore_move_constructors	Whether to allow non-const rvalue references on constructors.	True
level		required
report_overrides	Whether to report violations on overrides. For virtual methods, this rule will report a violation only if the parameter can be marked const in all related methods in the inheritance hierarchy. Thus, the same violation is reported for each method in the inheritance hierarchy. You can set this option to False to report such violations only for the base methods.	True
target		implementation

Possible Messages

Name	Message
parameter_can_point_to_const	{ } can be declared as pointer/reference to const.

AutosarC++18_03-M7.3.1

The global namespace shall only contain main, namespace declarations and extern "C" declarations.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
global_symbol	Symbol not allowed in global namespace.

AutosarC++18_03-M7.3.2

The identifier main shall not be used for a function other than the global function main.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
nonglobal_function_named_main	The identifier main shall not be used for a function other than the global function main

AutosarC++18_03-M7.3.3

There shall be no unnamed namespaces in header files.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unnamed_namespace_in_header	Unnamed namespaces in header file

AutosarC++18_03-M7.3.4

Using-directives shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
using_directive	Using-directives shall not be used

AutosarC++18_03-M7.3.6

using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
using_declaration_in_header	Using-declaration in header file
using_namespace_in_header	Using-directive in header file

AutosarC++18_03-M7.4.1

All usage of assembler shall be documented.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_assembler	Usage of assembler shall be documented

AutosarC++18_03-M7.4.2

Assembler instructions shall only be introduced using the asm declaration.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pragma_asm	Assembler instructions shall only be introduced using the asm declaration

AutosarC++18_03-M7.4.3

Assembly language shall be encapsulated and isolated.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
asm_not_encapsulated	Assembly language shall be encapsulated and isolated.

AutosarC++18_03-M7.5.1

A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
global_reference_to_local_var	Address of local variable escapes via global variable.
parameter_reference_to_local_var	Address of local variable escapes via parameter.
returning_reference_to_local_var	Returning reference/pointer to local variable.

AutosarC++18_03-M7.5.2

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_longer_living_local	Whether assignment to a longer-living local variable should be accepted.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
possibly_leaking_reference_to_local_variable	Address of local variable is assigned to longer-living object.

AutosarC++18_03-M8.0.1

An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_in_for_loop_init	Whether a multi-declaration is allowed in the for-init statement	False
allow_uninitialized_simple_type	Whether a multi-declaration of simple type is allowed when there is no initialization	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
multi_declaration	Multiple declarators in single declaration

AutosarC++18_03-M8.3.1

Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
redefinition_uses_different_default_argument	Default argument differs from the one in redefined method

AutosarC++18_03-M8.4.2

The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
ignore_casing	Whether to ignore casing for the check.	False
include_redefinitions	Whether declarations of redefining methods should be considered together with the redefined ones.	True
inspect_only_defined_routines	If False, re-declarations of undefined functions are compared to earlier declarations as well.	False
level		required
target		implementation

Possible Messages

Name	Message
parameter_name_mismatch	Different name used for parameter

AutosarC++18_03-M8.4.4

A function identifier shall either be used to call the function or it shall be preceded by &.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value	
allowed	Qualified names of functions of which it is allowed to take the address implicitly, e.g. C++ I/O manipulators	['std::endl', 'std::flush', 'std::boolalpha', 'std::noboolalpha', 'std::showbase', 'std::noshowbase', 'std::showpoint', 'std::noshowpoint', 'std::showpos', 'std::noshowpos', 'std::skipws', 'std::noskipws', 'std::uppercase', 'std::nouppercase', 'std::unitbuf', 'std::nounitbuf', 'std::internal', 'std::left', 'std::right', 'std::dec', 'std::hex', 'std::oct', 'std::fixed', 'std::scientific', 'std::hexfloat', 'std::defaultfloat', 'std::ws', 'std::ends', 'std::resetiosflag', 'std::setiosflag', 'std::setbase', 'std::setfill', 'std::setprecision', 'std::setw', 'std::setw', 'std::get_money', 'std::put_money', 'std::get_time', 'std::put_time', 'std::quoted']	
enforcement		automated	
level		required	
target		implementation	

Possible Messages

Name	Message
implicit_routine_address	Taking address of function without &

AutosarC++18_03-M8.5.2

Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_any_zero_initialization	Whether {0} is allowed in nested aggregates as well.	False
allow_toplevel_zero_initialization	Whether {0} is allowed as top-level initialization for the complete aggregate	True
allowed_classes_without_braces	For the listed class types, don't require braces. This is useful for classes that transparently wrap a single contained value. For example, with this option "std::array<int, 3> a = {1,2,3};" is OK even though there could be two sets of braces (one for "std::array", the other and for its inner "int[3]" array).	['std::array']
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
incomplete_aggregate_initialization	Provide explicit initializer for each part of initialized entity
missing_aggregate_initialization_braces	Use braces to indicate structure of initialized entity

AutosarC++18_03-M9.3.1

const member functions shall not return non-const pointers or references to class-data.

Input: IR

Configuration

Name	Explanation	Value
enforcement		automated
include_copies_of_pointer_fields	Whether returning a pointer field should be reported as well.	True
inspect_only_const_methods	Whether all methods or only const methods should be checked.	True
level		required
only_report_references	Whether pointer and reference to field should be reported, or just references.	False
smart_pointer_names	Names that are considered smart pointers.	['unique_ptr', 'shared_ptr', 'weak_ptr']
target		implementation

Possible Messages

Name	Message
returning_nonconst_member_reference	Returning non-const reference/pointer to class data.

AutosarC++18_03-M9.3.3

If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_candidates_for_const	If False, avoid reporting methods that can be made const.	True
report_candidates_for_static	If False, avoid reporting methods that can be made static.	True
target		implementation
test_operators_for_static	If True, check whether a method can be made static is also applied to operator methods	False

Possible Messages

Name	Message
method_can_be_const	Method can be declared const.
method_can_be_static	Method can be declared static.

AutosarC++18_03-M9.6.1

When the absolute positioning of bits representing a bit-field is required, then the behaviour and packing of bit-fields shall be documented.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitfield	Usage of bit-fields shall be documented

AutosarC++18_03-M10.1.1

Classes should not be derived from virtual bases.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
virtual_inheritance	Classes should not be derived from virtual bases.

AutosarC++18_03-M10.1.2

A base class shall only be declared virtual if it is used in a diamond hierarchy.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
virtual_inheritance_outside_diamond	A base class shall only be declared virtual if it is used in a diamond hierarchy.

AutosarC++18_03-M10.1.3

An accessible base class shall not be both virtual and non-virtual in the same hierarchy.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
base_class_being_virtual_and_nonvirtual	Has base class which is both virtual and non-virtual.

AutosarC++18_03-M10.2.1

All accessible entity names within a multiple inheritance hierarchy should be unique.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
ambiguous_member	All accessible entity names within a multiple inheritance hierarchy should be unique
use_of_ambiguous_name	{ } is ambiguous

AutosarC++18_03-M10.3.3

A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pure_redefinition	Pure redefinition of non-pure virtual function.

AutosarC++18_03-M11.0.1

Member data in non-POD class types shall be private.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_protected_members	If True, protected members are tolerated.	False
allowed	Specifies allowed fields as pairs [class name pattern, field name pattern]. Example: [re.compile('.*'), re.compile('x')] to allow x in all classes.	[]
enforcement		automated
ignore_const_members	If True, non-private const members are tolerated.	False
ignore_pod	Whether fields in POD classes should be reported.	True
ignore_structs	Whether fields in structs should be reported.	False
ignore_templates	Whether fields in generic templates should be reported.	True
level		required
target		implementation

Possible Messages

Name	Message
protected_field	Member data in non-POD class types shall be private.
public_field	Member data in non-POD class types shall be private.

AutosarC++18_03-M12.1.1

An object's dynamic type shall not be used from the body of its constructor or destructor.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
constructor_using_dynamic_cast	Dynamic cast used in constructor/destructor.
constructor_using_typeid	Typeid on polymorphic class used in constructor/destructor.
constructor_using_virtual_call	Virtual call used in constructor/destructor.

AutosarC++18_03-M14.5.3

A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_copy_asgn_for_template	Class has template assignment operator but no copy assignment operator

AutosarC++18_03-M14.6.1

In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unqualified_dependent_member_access	Use qualifiers or this-> to select name that may be found in that dependent base

AutosarC++18_03-M15.0.3

Control shall not be transferred into a try or catch block using a goto or a switch statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
goto_into_try	Goto jumps into try or catch block
switch_into_try	Switch statement jumps into try or catch block

AutosarC++18_03-M15.1.1

The assignment-expression of a throw statement shall not itself cause an exception to be thrown.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throw_expression_raises_exception	Expression of throw may itself raise an exception

AutosarC++18_03-M15.1.2

NULL shall not be thrown explicitly.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throwing_null	NULL shall not be thrown explicitly

AutosarC++18_03-M15.1.3

An empty throw (throw;) shall only be used in the compound-statement of a catch handler.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
rethrow_outside_catch	Rethrow outside any catch block

AutosarC++18_03-M15.3.1

Exceptions shall be raised only after start-up and before termination of the program.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_only_one_exception_per_function	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
report_only_uncaught	Whether the check shall report all throws or just those not caught.	False
target		implementation

Possible Messages

Name	Message
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionRaisedInInitialization	Exception raised in initialization or finalization

AutosarC++18_03-M15.3.3

Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
handlerUsesField	Handler of a function-try-block shall not reference non-static members from this class or its bases

AutosarC++18_03-M15.3.4

Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
inspect_at_exit_handlers	Whether to also inspect at_exit() handlers-functions.	True
inspect_thread_main	Whether to also inspect thread main functions.	True
level		required
report_only_one_exception_per_function	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
target		implementation

Possible Messages

Name	Message
exception_escaping_initialization	Uncaught exception raised in initialization or finalization
exception_escaping_main	Uncaught exception escaping from main or additional entry point

AutosarC++18_03-M15.3.6

Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
wrong_catch_order	Catch handlers in wrong order.

AutosarC++18_03-M15.3.7

Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
catch_all_not_last	Catch-all shall occur as last handler.

AutosarC++18_03-M16.0.1

#include directives in a file shall only be preceded by other preprocessor directives or comments.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_extern_blocks	Whether to allow includes in `extern` blocks and not report a violation.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
include_after_code	#include should only be preceded by comments or preprocessor directives

AutosarC++18_03-M16.0.2

Macros shall only be #define'd or #undef'd in the global namespace.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
local_macro	#define or #undef not in global namespace

AutosarC++18_03-M16.0.5

Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
directives	Preprocessing directives to check for in macro arguments.	(`#if', `#ifdef', `#ifndef', `#elif', `#else', `#endif', `#pragma', `#warning', `#error', `#line', `#include', `#include_next', `#ident', `#region', `#endregion', `#asm', `#endasm', `#define', `#undef')
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pp_directive_as_macro_arg	Preprocessing directive used in macro argument.

AutosarC++18_03-M16.0.6

In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unsafe_macro_param	Macro parameter not enclosed in parentheses

AutosarC++18_03-M16.0.7

Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
undefined_macro_used	Zero used for undefined preprocessing identifier

AutosarC++18_03-M16.0.8

If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
malformed_pp_directive	Malformed or unknown preprocessing directive

AutosarC++18_03-M16.1.1

The defined preprocessor operator shall only be used in one of the two standard forms.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
nonstandard_defined	Non-standard use of defined operator

AutosarC++18_03-M16.1.2

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_endif	The #endif for this directive is missing
missing_if	The #if for this directive is missing

AutosarC++18_03-M16.2.3

Include guards shall be provided.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
macro_name_restrictions	Python iterable of functions with parameters (file, define, macro) to perform additional checks on the macro used as include guard in the given file. Should return None if macro is accepted, else a message to print for the violation. The parameters are: PIR File node for the header, token for the macro #define or #pragma once, and the name of the macro or the string 'pragma once'.	None
target		implementation

Possible Messages

Name	Message
include_guard_missing	Missing include guard.
include_guard_partial	Include guard does not cover complete file.
include_guard_with_wrong_define	Bad include guard: #define uses wrong macro, should be {}.
include_guard_without_define	Incomplete include guard: missing #define {}.

AutosarC++18_03-M16.3.1

There shall be at most one occurrence of the # or ## operators in a single macro definition.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_only_concatenated_stringized	If True, only macros where a parameter is stringized and the result used for token concatenation are reported.	False
target		implementation

Possible Messages

Name	Message
concatenated_stringized	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator
more_than_one_macro_hash	Only one # or ## allowed in macro definition

AutosarC++18_03-M16.3.2

The # and ## operators should not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
hash_in_macro	The # and ## operators should not be used.

AutosarC++18_03-M17.0.2

The names of standard library macros and objects shall not be reused.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_locals	Whether parameters and local variables should also be checked	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
reused_macro_object_libname	The names of standard library macros and objects shall not be reused.

AutosarC++18_03-M17.0.3

The names of standard library functions shall not be overridden.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_function_declarations	Whether nondefining function declarations with library names are allowed	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
reused_routine_libname	The names of standard library functions shall not be overridden.

AutosarC++18_03-M17.0.5

The setjmp macro and the longjmp function shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	setjmp
symbols	Names of symbols which are forbidden.	['setjmp', 'longjmp']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++18_03-M18.0.3

The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	stdlib
symbols	Names of symbols which are forbidden.	['abort', 'exit', 'getenv', 'system']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib{.h} -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++18_03-M18.0.4

The time handling functions of library <ctime> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	time
symbols	Names of symbols which are forbidden.	['clock', 'difftime', 'mktime', 'time', 'asctime', 'ctime', 'gmtime', 'localtime', 'strftime']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib{.h} -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++18_03-M18.0.5

The unbounded functions of library <cstring> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	string
symbols	Names of symbols which are forbidden.	['strcpy', 'strcmp', 'strcat', 'strchr', 'strspn', 'strcspn', 'strpbrk', 'strrchr', 'strstr', 'strtok', 'strlen']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h} -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++18_03-M18.2.1

The macro offsetof shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	stddef
symbols	Names of symbols which are forbidden.	['offsetof']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h} -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++18_03-M18.7.1

The signal handling facilities of <csignal> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header (used for symbols like NULL and size_t which are defined in multiple headers)	['NULL', 'size_t']
enforcement		automated
header	Name of the header file which should not be used.	signal
level		required
target		implementation

Possible Messages

Name	Message
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.

AutosarC++18_03-M19.3.1

The error indicator errno shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	['errno', 'stdlib', 'stddef']
symbols	Names of symbols which are forbidden.	['errno']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <cerrno>.

AutosarC++18_03-M27.0.1

The stream input/output library <cstdio> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header (used for symbols like NULL and size_t which are defined in multiple headers)	['NULL', 'size_t']
enforcement		automated
header	Name of the header file which should not be used.	stdio
level		required
target		implementation

Possible Messages

Name	Message
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.

Rules in Group AutosarC++18_10

AutosarC++18_10-A0.1.1

A project shall not contain instances of non-volatile variables being given values that are never subsequently used.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_dead_initializations	Whether initialisations may be reported as dead.	True
target		implementation

Possible Messages

Name	Message
conditional_unused_def	Result of assignment is not used along some path(s)
init_used_in_other_isr	Initialization is only used in some interrupt handler
unused_def	Result of assignment is not used
unused_init	Unused initialization
used_in_other_isr	Result of assignment is only used in some interrupt handler

AutosarC++18_10-A0.1.2

The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed_functions	Calls to these functions are ignored.	frozenset(['strncat', 'memmove', 'memset', 'strcat', 'strcpy', 'memcpy', 'strncpy'])
enforcement		automated
inspect_template_instances	Whether calls in template instances should be reported.	False
level		required
target		implementation

Possible Messages

Name	Message
discarded_return_with_entity	Return value of function discarded.
discarded_return_without_entity	Return value of function discarded.

AutosarC++18_10-A0.1.3

Every defined function shall be called at least once.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_copy_assignment_operators	Whether uncalled copy assignment operators are allowed. Other rules like the "rule of three" might require copy assignment operators as part of good class design, even if there are currently no callers of the copy assignment operator.	False
ignore_copy_constructors	Whether uncalled copy constructors are allowed. Other rules like the "rule of three" might require copy constructors as part of good class design, even if there are currently no callers of the copy constructor.	False
level		required
only_check_internal_or_private_methods	Whether only function with internal linkage or private functions should be checked.	True
only_check_static_or_private_methods	Whether only static functions or private functions should be checked.	False
only_check_unit_locals	Whether only global static functions should be checked.	False
target		implementation

Possible Messages

Name	Message
defined_function_not_called	Defined function is not called

AutosarC++18_10-A0.1.4

There shall be no unused named parameters in non-virtual functions.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
inspect_nonvirtual_functions	Whether parameters of non-virtual functions should be checked.	True
inspect_virtual_functions	Whether virtual functions should be checked.	False
level		required
report_unnamed_parameters	Whether unnamed unused parameters should be reported as well.	False
target		implementation

Possible Messages

Name	Message
unused_parameter	Unused named parameter of non-virtual function

AutosarC++18_10-A0.1.5

There shall be no unused named parameters in the set of parameters for a virtual function and all the functions that override it.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
inspect_nonvirtual_functions	Whether non-virtual functions should be checked.	False
inspect_virtual_functions	Whether parameters of virtual functions should be checked. Violations will be reported in the base class when none of the derived classes make use of the parameter.	True
level		required
report_unnamed_parameters	Whether unnamed unused parameters should be reported as well.	False
target		implementation

Possible Messages

Name	Message
unused_parameter	Unused named parameter of virtual function

AutosarC++18_10-A0.1.6

A project shall not contain unused type declarations.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unused_type	Unused type declaration

AutosarC++18_10-A0.4.2

Type long double shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
forbidden_types	The ir.Logical Types not to be used e.g. `C_Long_Double_Type`.	['C_Long_Double_Type']
level		required
target		implementation

Possible Messages

Name	Message
type_used	The type shall not be used.

AutosarC++18_10-A1.1.1

All code shall conform to ISO/IEC 14882:2014 - Programming Language C++ and shall not use deprecated features.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	The set of messages regarding syntax and constraint violations.	set([2464, 2465, 1444, 2567, 2381, 2221, 1909, 1215])
reported_severities	List of severities to display.	('error', 'warning')
show_include_path	If True, violations regarding nested includes print the #include path.	False
standards	List of allowed frontend command-line options to select language standard.	['--strict', '-A']
target		implementation
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

Possible Messages

Name	Message
access_control	Number of access control declarations is higher than translation limit (which is {})
at_quick_exit_functions	Number of functions registered with at_quick_exit() is higher than translation limit (which is {})
atexit_functions	Number of functions registered with atexit() is higher than translation limit (which is {})
base_classes	Number of direct and indirect base classes is higher than translation limit (which is {})
call_arguments	Number of arguments is higher than translation limit (which is {})
catch_handlers	Number of catch handlers is higher than translation limit (which is {})
chars_in_line	Number of characters in logical line is higher than translation limit (which is {})
chars_in_string	Number of characters in string is higher than translation limit (which is {})
cpp_members	Number of members is higher than translation limit (which is {})

data_members	Number of data members is higher than translation limit (which is {})
direct_bases	Number of direct base classes is higher than translation limit (which is {})
enumerators	Number of enumerators is higher than translation limit (which is {})
expr_in_core_constant	Number of full expressions evaluated within a core constant expression is higher than translation limit (which is {})
external_identifiers	Number of external identifiers in this unit is higher than translation limit (which is {})
final_virtual_functions	Number of final overriding virtual functions is higher than translation limit (which is {})
friends	Number of friend declarations is higher than translation limit (which is {})
function_parameters	Number of parameters is higher than translation limit (which is {})
macro_arguments	Number of macro arguments is higher than translation limit (which is {})
macro_parameters	Number of macro parameters is higher than translation limit (which is {})
macros	Number of macro definitions in this unit is higher than translation limit (which is {})
member_initializers	Number of member initializers is higher than translation limit (which is {})
nested_blocks	Number of nested blocks is higher than translation limit (which is {})
nested_composites	Nesting level of structs and unions is higher than translation limit (which is {})
nested_declarators	Number of pointer/array/function declarators is higher than translation limit (which is {})
nested_externals	Number of nested external specifications is higher than translation limit (which is {})
nested_includes	Nesting level of #includes is higher than translation limit (which is {})
nested_preprocessor_ifs	Nesting level of #if is higher than translation limit (which is {})
nested_template_instantiations	Number of nested template instantiations is higher than translation limit (which is {})
object_size	Number of bytes in object is higher than translation limit (which is {})
parenthesized_declarators	Nesting level of parenthesized declarators is higher than translation limit (which is {})
parenthesized_expressions	Nesting level of parenthesized subexpressions is higher than translation limit (which is {})
placeholders	Number of placeholders is higher than translation limit (which is {})
recursive_constexpr	Number of recursive constexpr function invocations is higher than translation limit (which is {})
scope_qualifications	Number of scope qualifications is higher than translation limit (which is {})
significant_chars_external	Number of significant initial characters for external identifier is higher than translation limit (which is {})
significant_chars_internal	Number of significant initial characters for internal identifier is higher than translation limit (which is {})
static_members	Number of static members is higher than translation limit (which is {})
std_compile	You have to compile with option {}
switch_cases	Number of cases is higher than translation limit (which is {})
template_params	Number of template parameters is higher than translation limit (which is {})
throw_spec	Number of throw specifications is higher than translation limit (which is {})
variables_in_block	Number of variables in block is higher than translation limit (which is {})
virtual_bases	Number of virtual base classes is higher than translation limit (which is {})

AutosarC++18_10-A1.4.3

All code should compile free of compiler warnings.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	If provided, only messages of these types are reported.	None
reported_severities	List of severities to display.	('error', 'warning')
target		implementation
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	False

AutosarC++18_10-A2.3.1

Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_comments	Whether to allow characters not in the basic source character set in comments.	False
allow_in_unicode_strings	Whether to allow characters not in the basic source character set in unicode string literals.	True
allow_in_wide_strings	Whether to allow characters not in the basic source character set in wide string literals.	True
basic_characters_list	The basic character list as per rule.	\\\\\\ abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ#\\~!@%^&_=\\\"\\@
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
forbidden_characters	Forbidden characters used.

AutosarC++18_10-A2.5.1

Trigraphs shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_comments	Allow the use of trigraphs in comments	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
trigraph_use	Use of trigraph.

AutosarC++18_10-A2.5.2

Digraphs should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
digraph_use	Digraph used.

AutosarC++18_10-A2.7.1

The character \ shall not occur as a last character of a C++ comment.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
line_splicing_in_cpp_comment	Line-splicing shall not be used in // comments.

AutosarC++18_10-A2.7.2

Sections of code shall not be "commented out".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
files_to_check	Files to be checked, e.g. Primary_File or File.	File
level		required
normal_text	Tokens that indicate non-code comments. This is the first step of checking a comment. If any of these words exist in the comment text, it is regarded as not containing code.	{'and', 'or', 'but', 'now', 'to', 'is', 'are', 'only', 'be', 'has', 'the', 'with', 'because', 'when', 'oder', 'und', '//', '#', 'AXIVION', '++++', '---', '===='})
normal_text_regex	Regular expression to match normal non-code text. This is the second step of checking a comment. If this regex matches the comment text, the comment is regarded as not containing code.	\b[\w\d_]+\s+[\w\d_]+\s+[\w\d_]+\b
suspicious	Tokens that indicate code in comments. This is the final step of checking a comment. If the first two steps did not exclude the comment from further inspection, the comment text is searched for any of these suspicious tokens. The comment is regarded as containing code, if any suspicious token is found.	{'=', '==', '>=', '<=', '[', ']', ':', '->', '->*', '/*', 'if', 'while', 'for', 'if (', 'while (', 'for (', '#pragma', '#else', '#endif', '#if', '#include', '++', '--')}
target		implementation

Possible Messages

Name	Message
code_in_c_comment	Sections of code shall not be commented out using C-style comments.
code_in_cpp_comment	Sections of code should not be commented out using C++ comments.

AutosarC++18_10-A2.7.3

All declarations of "user-defined" types, static and non-static data members, functions and methods shall be preceded by documentation.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_inherited	If True, a definition does not need documentation, if a corresponding declaration is documented.	False
allow_missing_documentation_on_private	If True, a class-member definition does not need documentation, if it is `private`.	False
allow_missing_documentation_on_protected	If True, a class-member definition does not need documentation, if it is `protected`.	False
doxygen_start	Start of a valid Doxygen comment.	{'/**', '///'}
enforcement		automated
ignore_defaulted	If True, defaulted function declarations are not checked for comments.	False
ignore_deleted	If True, deleted function declarations are not checked for comments.	False
ignore_redefinitions	If True, method redefinitions are not checked as they can 'inherit' the comment from the redefined method.	False
ignore_tool_comments	An optional compiled regular expression. Comments where this regex finds a matching substring are ignored in the search for a doxygen comment (e.g. control-comments of other tools).	None
level		required
node_types	IR node types to check for preceding Doxygen comment.	{'Routine_Definition', 'Routine_Declaration', 'Named_Type_Definition', 'Field_Definition'}
target		implementation

Possible Messages

Name	Message
missing_doxygen_comment_before_def	No Doxygen comment before declaration.

AutosarC++18_10-A2.8.1

A header file name should reflect the logical entity for which it provides declarations.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
file_type	Type of source_file file to examine.	User_Include_File
junk_characters	Characters that should not be considered in the filename to type similarity.	_-::
level		required
named_node_types	Physical IR node types to inspect for logical names to compare against the file name	{'Definition', 'No_Def_Declaration'}
similarity_threshold	Ratio of how similar the names need to be.	0.9
target		architecture/design/implementation

Possible Messages

Name	Message
source_file_name_not_type	The header should be named as a type it declares.

AutosarC++18_10-A2.8.2

An implementation file name should reflect the logical entity for which it provides definitions.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
file_type	Type of source_file file to examine.	Primary_File
junk_characters	Characters that should not be considered in the filename to type similarity.	_-::
level		advisory
named_node_types	Physical IR node types to inspect for logical names to compare against the file name	('Definition', 'No_Def_Declaration')
similarity_threshold	Ratio of how similar the names need to be.	0.9
target		architecture / design / implementation

Possible Messages

Name	Message
source_file_name_not_type	The file should be named as a type it declares.

AutosarC++18_10-A2.10.1

Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_classes	Whether identifiers in local classes can hide surrounding ones	True
allow_in_lambdas	Whether identifiers in lambdas can hide surrounding ones	True
allow_in_local_classes	Whether identifiers in local classes can hide surrounding ones	False
allow_in_namespaces	Whether identifiers in namespaces can hide surrounding ones	True
allow_parameter_in_function_declaration	Whether identifiers of parameters in function declarations may hide surrounding ones, given the function declaration uses a different identifier for the same parameter that does not hide a surrounding one.	False
check_inheritance_hiding	Whether to check for hiding by inheritance.	False
check_nested_classes	Whether symbols from inner classes should be reported as hiding symbols of outer classes.	False
distinguish_name_spaces	Whether C name spaces should be distinguished.	False
enforcement		automated
hiding_entities	LIR node types to consider symbols that hide other symbols.	{'Variable', 'Typedef_Type', 'Field', 'Parameter'}
level		required
maxlen	Number of significant characters {or None}	None
target		architecture/design/implementation
tolerate_constructor_parameters_hiding_fields	Allow constructor parameters to hide fields	False
tolerate_function_prototype_argument_hiding	Whether to tolerate arguments of a function prototype, that itself is a function argument, that have the same identifier as a argument of the outer function. If 'True' is given, the rule will still report cases, where argument identifiers match up and the function prototype refers to arguments of the outer function (e.g. by `decltype()`).	False
tolerate_macro_local_identifier	Whether an identifier being local to a macro is allowed to hide identifiers from outside the macro	False
unchecked_types	Deprecated legacy option, kept to avoid breaking old configurations.	{'Routine_Call', 'Stack_Object_Definition'}

Possible Messages

Name	Message
hiding	{ } hides { }

AutosarC++18_10-A2.10.4

The identifier name of a non-member object or function with static storage duration should not be reused.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_static_no_linkage	Whether to allow same identifiers for static objects without linkage (e.g. function local static objects)	False
enforcement		automated
func_filter	Restricts which functions are considered.	None
level		required
target		implementation
use_unqualified_names	If set to True, ns1::Type is considered equal to ns2::Type.	False
var_filter	Restricts which variables are considered.	None

Possible Messages

Name	Message
reused_global_function_name	Name of object or function with static storage duration reused.
reused_global_variable_name	Name of object or function with static storage duration reused.

AutosarC++18_10-A2.10.5

The identifier name of a non-member object or function with static storage duration should not be reused.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_static_no_linkage	Whether to allow same identifiers for static objects without linkage (e.g. function local static objects)	True
enforcement		automated
func_filter	Restricts which functions are considered.	None
level		required
target		design/architecture
use_unqualified_names	If set to True, ns1::Type is considered equal to ns2::Type.	True
var_filter	Restricts which variables are considered.	None

Possible Messages

Name	Message
reused_global_function_name	Name of object or function with static storage duration reused.
reused_global_variable_name	Name of object or function with static storage duration reused.

AutosarC++18_10-A2.10.6

A class or enumeration name shall not be hidden by a variable, function or enumerator declaration in the same scope.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
invert_findings	Whether to invert primary and secondary SLocs for violations	True
level		required
target		implementation

Possible Messages

Name	Message
reused_type	Type should not be hidden by a name in the same scope.

AutosarC++18_10-A2.11.1

Volatile keyword shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		design / implementation

Possible Messages

Name	Message
volatile_qualifier	The volatile type qualifier shall not be used

AutosarC++18_10-A2.13.1

Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed	List of allowed characters after backslash.	"\"?\\abfnrtv
enforcement		automated
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
nonstandard_escape_sequence	Use of non-standard escape sequence.

AutosarC++18_10-A2.13.2

Narrow and wide string literals shall not be concatenated.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
cpp11_mode	Use rules as defined in the C++11 standard.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
mixed_string_concatenation	Concatenation of mixed string encodings
narrow_wide_concat	Concatenation of narrow and wide string literal

AutosarC++18_10-A2.13.3

Type wchar_t shall not be used.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
forbidden_types	The ir.Logical Types not to be used e.g. `C_Long_Double_Type`.	['CPP_WChar_Type']
level		required
target		architecture/design/implementation/implementation

Possible Messages

Name	Message
type_used	The type shall not be used.

AutosarC++18_10-A2.13.4

String literals shall not be assigned to non-constant pointers.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	If provided, only messages of these types are reported.	[2464]
reported_severities	List of severities to display.	('error', 'warning', 'remark')
target		architecture / design / implementation
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

AutosarC++18_10-A2.13.5

Hexadecimal constants should be upper case.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
hex_literal	Hexadecimal constants should be upper case.

AutosarC++18_10-A2.13.6

Universal character names shall be used only inside character or string literals.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_comments	Whether to allow universal character names in comments.	False
allow_in_string_literals	Whether to allow universal character names in string literals.	True
enforcement		automated
level		required
target		architecture / design / implementation

Possible Messages

Name	Message
universal_character	Use of universal character names

AutosarC++18_10-A3.1.1

It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
accept_const_fields	Whether const-qualified static fields in header files should be tolerated.	True
accept_const_variables	Whether global const variables in header files should be tolerated.	True
enforcement		automated
find_non_external_function_declaration	Whether function declarations with non-external linkage should be reported.	True
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
function_definition_in_header	Definition in header file.
static_field_def_in_header	Definition in header file.
variable_definition_in_header	Definition in header file.

AutosarC++18_10-A3.1.2

Header files, that are defined locally in the project, shall have a file name' extension of one of: ".h", ".hpp" or "..hxx".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed_extensions	Which file extensions are allowed.	set[{'h', '.hxx', 'hpp'}]
enforcement		automated
file_type	The files to check the extensions of.	User_Include_File
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
illegal_file_extension	Use only certain extensions for file names.

AutosarC++18_10-A3.1.3

Implementation files, that are defined locally in the project, should have a file name extension of ".cpp".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed_extensions	Which file extensions are allowed.	set[{'cpp'}]
enforcement		automated
file_type	The files to check the extensions of.	Primary_File
level		advisory
target		architecture/design/implementation

Possible Messages

Name	Message
illegal_file_extension	Use only certain extensions for file names.

AutosarC++18_10-A3.1.4

When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_extern	Whether to consider only extern declared arrays	True
report_definitions	Whether definitions of array variables should also be reported	True
target		design/implementation

Possible Messages

Name	Message
unbounded_array	Array declaration with unknown size.

AutosarC++18_10-A3.1.6

Trivial accessor and mutator functions should be inlined.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
exclude_constructor_destructors	Whether to exclude constructors/destructors from inspected functions.	True
exclude_operator_functions	Whether to exclude operator functions from inspected functions.	False
exclude_virtual_functions	Whether to exclude virtual functions from inspected functions.	True
level		advisory
require_this_read_write	Whether to require a read/write on this for a function to be considered a trivial accessor/mutator.	False
target		design

Possible Messages

Name	Message
missing_inline	Trivial accessor and mutator functions should be inlined.

AutosarC++18_10-A3.3.1

Objects or functions with external linkage shall be declared in a header file.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_function_declaration_in_header	Object or function with external linkage shall be declared in a header file
missing_variable_declaration_in_header	Object or function with external linkage shall be declared in a header file

AutosarC++18_10-A3.3.2

Static and thread-local objects shall be constant-initialized.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed_effects_mask	If set, static and thread-local objects may also be initialized with a non-constexpr function/constructor call that only has the specified side-effects.	None
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
constexpr_init_variable	Static and thread-local objects shall be constant-initialized.

AutosarC++18_10-A3.9.1

Typedefs that indicate size and signedness should be used in place of the basic numerical types.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_floating_point	Whether floating point types without should not be reported.	True
enforcement		automated
ignore_inherited	If true, missing typeDefs in inherited methods are not reported.	False
level		required
target		implementation
treat_plain_char_as_numerical	Whether the char type without any signedness should be reported.	True

Possible Messages

Name	Message
missing_integer_TypeDef	Use of base type outside typeDef.
wrong_integer_TypeDef	Use of badly named typeDef for base type.

AutosarC++18_10-A4.5.1

Expressions with type enum shall not be used as operands to built-in operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_use_in_operator_calls	Whether enum arguments in calls to forbidden overloaded operators should be reported.	True
target		implementation

Possible Messages

Name	Message
enum_operand_outside_comparison	Use of enum operand in arithmetic or similar context

AutosarC++18_10-A4.7.1

An integer expression shall not lead to data loss.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
abstract_interpretation_overflow	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cast_overflow	Cast on result of arithmetic computation may cause overflow
cast_truncate	Cast may truncate value
cast_underflow	Cast on result of arithmetic computation may cause underflow
overflow	Arithmetic computation may cause overflow
underflow	Arithmetic computation may cause underflow

AutosarC++18_10-A4.10.1

Only nullptr literal shall be used as the null-pointer-constant.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
require_nullptr	Require C++11 nullptr instead of macro NULL or literal zero {0}.	True
target		architecture/design/implementation

Possible Messages

Name	Message
null_constant	Only nullptr literal shall be used as the null-pointer-constant.
zero_as_null	Use of literal zero {0} as null-pointer-constant, use {} instead

AutosarC++18_10-A5.0.1

The value of an expression shall be the same under any order of evaluation that the standard permits.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_exceptions	Ignore possible exceptions in the computation of side effects. (this option only has an effect if report_calls is enabled)	False
level		required
report_calls	If True, unsequenced function calls are reported.	True
target		implementation

Possible Messages

Name	Message
unsequenced_call_volatile	Function calls unsequenced with volatile access
unsequenced_calls	Unsequenced function calls
unsequenced_read_write	Unsequenced read and write accesses
unsequenced_volatiles	Unsequenced volatile accesses
unsequenced_writes	Unsequenced writes

AutosarC++18_10-A5.0.2

The condition of an if-statement and the condition of an iteration-statement shall have type bool.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_in_conditions	If True, only logical operators inside conditions are checked.	True
report_do_while_false	Whether do ... while[0] should be reported.	True
report_while_true	Whether while[1] ... should be reported.	True
target		implementation

Possible Messages

Name	Message
nonbool_if_condition	Condition must have type bool
nonbool_logical_operator_operand	Sub-condition must have type bool
nonbool_loop_condition	Condition must have type bool

AutosarC++18_10-A5.0.3

The declaration of objects should contain no more than two levels of pointer indirection.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
max_levels	Maximum number of allowed pointer-indirection levels.	2
target		implementation

Possible Messages

Name	Message
too_many_pointer_levels	More than {} level{} of pointer indirection

AutosarC++18_10-A5.0.4

Pointer arithmetic shall not be used with pointers to non-final classes.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
consider_non_base_final	Whether to consider a class that is not used as a base class as a sort-of final class.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
non_final_pointer_arithmetic	Pointer arithmetic on non-final classes not allowed

AutosarC++18_10-A5.1.1

Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_nonconst_variable_initialization	If set to `True`, allow the initialization of a non-const variable with a string literal in addition to the initialization of a const variable. A reassignment of such a variable is still a violation.	False
allowed	Literal values that are ok.	{0.0, 1.0, 2.0}
allowed_contexts	Optional set of PIR classes or functions {[node] -> bool} for allowed contexts, e.g. Case_Label.	set[<function allow_in_type_initialization at 0x7dfa84459b0>]
allowed_logging_contexts	List of fully qualified function types that are considered logging contexts [using operator<>]. If this is non-empty, `std::throw()` is considered a valid logging context as well.	['std::basic_ostream']
allowed_string_contexts	Optional set of PIR classes or functions {[node] -> bool} for allowed contexts.	set[<function allow_in_type_initialization at 0x7dfa84459b0>]
allowed_strings	Literal values that are ok.	['', ' ']
enforcement		partially automated
exceptions	Optional predicate to filter out cases that should be allowed as an exception. The predicate takes a literal node and returns True if the given node is such an exception.	None
exclude_pp_literals	If True, literals in conditions of #if are ignored.	True
exclude_single_uses	If True, report only literals used more than once.	False
level		required
target		implementation

Possible Messages

Name	Message
magic_number	Use of magic literal.
magic_number_without_token	Use of magic literal.
magic_string	Use of magic string literal.
possible_magic_number	Potential use of magic literal.

AutosarC++18_10-A5.1.2

Variables shall not be implicitly captured in a lambda expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
implicit_lambda_capture	Variables shall not be implicitly captured in a lambda expression.

AutosarC++18_10-A5.1.3

Parameter list (possibly empty) shall be included in every lambda expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_parameters	Include a (possibly empty) parameter list

AutosarC++18_10-A5.1.6

Return type of a non-void return type lambda expression should be explicitly specified.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
missing_explicit_return	Add an explicit return type for the lambda.

AutosarC++18_10-A5.1.7

The underlying type of lambda expression shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
underlying_lambda_type	The underlying type of lambda expression shall not be used.

AutosarC++18_10-A5.1.8

Lambda expressions should not be defined inside another lambda expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
nested_lambda	Lambda expressions should not be defined inside another lambda expression

AutosarC++18_10-A5.1.9

Identical unnamed lambda expressions shall be replaced with a named function or a named lambda expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_empty_lambdas	If set to True, this rule will not report duplicates of the trivial empty lambda.	False
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
identical_unnamed_lambda	Identical unnamed lambdas.

AutosarC++18_10-A5.2.1

dynamic_cast should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
dynamic_cast	dynamic_cast should not be used.

AutosarC++18_10-A5.2.2

Traditional C-style casts shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_void_cast	If True, (void) is always allowed, else only for return value of calls.	False
allow_void_cast_on_call	If True, (void) is allowed, for return value of calls.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
c_cast	Use of C-style cast in C++ unit.

AutosarC++18_10-A5.2.3

A cast shall not remove any const or volatile qualification from the type of a pointer or reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
check_const	Whether casts affecting const-qualifiers should be reported	True
check_volatile	Whether casts affecting volatile-qualifiers should be reported	True
enforcement		automated
level		required
only_top_level	Whether the check only applies to the top-level pointee or recursively to deeper pointer levels as well	False
target		implementation

Possible Messages

Name	Message
cast_removes_const	Cast removes const qualification
cast_removes_volatile	Cast removes volatile qualification
implicit_cast_removes_const	Implicit cast removes const qualification
implicit_cast_removes_volatile	Implicit cast removes volatile qualification

AutosarC++18_10-A5.2.4

reinterpret_cast shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
reinterpret_cast	reinterpret_cast should not be used.

AutosarC++18_10-A5.2.6

The operands of a logical && or || shall be parenthesized if the operands contain binary operators.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed	Can be used to name additional allowed IR node types for the operands.	None
enforcement		automated
level		required
require_postfix_expression	Whether postfix or primary expressions are required as operands.	False
target		implementation

Possible Messages

Name	Message
missing_parens_for_logical_operator_operand	Operand of && or must be parenthesized.

AutosarC++18_10-A5.3.1

Evaluation of the operand to the typeid operator shall not contain side effects.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
level		required
target		implementation

Possible Messages

Name	Message
side_effect_in_typeid	Operand of "typeid" shall not contain side effects

AutosarC++18_10-A5.6.1

The right hand operand of the integer division or remainder operators shall not be equal to zero.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
abstract_interpretation_div_by_zero	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
division_by_zero	Division by zero
modulo_by_zero	Modulo by zero
possible_division_by_zero	Possible division by zero
possible_modulo_by_zero	Possible modulo by zero

AutosarC++18_10-A5.10.1

A pointer to member virtual function shall only be tested for equality with null-pointer-constant.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
member_virtual_function_comparison	Comparison of a member virtual function with non-nullptr.

AutosarC++18_10-A5.16.1

The ternary conditional operator shall not be used as a sub-expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_nested_conditional_operator	Use of nested conditional operator.

AutosarC++18_10-A6.4.1

Every switch statement shall have at least one case-clause.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
minimum_switch_cases	The number of cases a switch statement should at least have.	2
target		implementation

Possible Messages

Name	Message
empty_switch	Switch has no "case" clause.
minimum_switch_cases	Switch has too little "case" clauses.

AutosarC++18_10-A6.5.1

A for-loop that loops through all elements of the container and does not use its loop-counter shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
exclude_simple_for_loops	If set to `True` this rule will not report unused loop counters in the simple cases where the C for-loop only references loop-counters in its condition but no other variables.	True
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
could_be_range_based	For loop could be a range-based for loop.
unused_loop_counter	For loop does not use its loop counter.

AutosarC++18_10-A6.5.2

A for loop shall contain a single loop-counter which shall not have floating type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
loop_float_counter	Loop-counter of for loop shall not have floating type
loop_missing_counter	For loop has no loop-counter
loop_multiple_counters	For loop shall have only a single loop-counter

AutosarC++18_10-A6.5.3

Do statements should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
do_while_loop	Do statements should not be used.

AutosarC++18_10-A6.5.4

For-init-statement and expression should not perform actions other than loop-counter initialization and modification.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
loop_side_effect	Side effect in loop init/expression.
non_loop_counter_initialized	Non loop counter initialized.
non_loop_counter_modified	Non loop counter modified.

AutosarC++18_10-A6.6.1

The goto statement shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
goto	Do not use goto.

AutosarC++18_10-A7.1.1

Constexpr or const specifiers shall be used for immutable data declaration.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_pointer_variables	Whether variables of pointer type should be ignored.	False
level		required
only_check_unit_locals	Whether only local variables and global static variables should be checked.	False
only_immutable_data	Whether only declarations with immutable data should be checked.	True
target		implementation

Possible Messages

Name	Message
variable_missing_const	An immutable variable shall be const/constexpr qualified.

AutosarC++18_10-A7.1.3

CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_nonpointer_typedefs	Whether to allow the qualifier on the rhs of a type, if it is a non-pointer typedef.	False
allowed_typedefs	Set of names for typedefs/usings that are allowed (e.g. int32_t).	set(['int32_t', 'uint_least64_t', 'intptr_t', 'uintmax_t', 'int_fast16_t', 'intmax_t', 'int_fast8_t', 'int64_t', 'size_t', 'int_fast64_t', 'time_t', 'uint8_t', 'lldiv_t', 'int_least8_t', 'div_t', 'uint_least16_t', 'clock_t', 'uint_least32_t', 'int_least64_t', 'int_least16_t', 'int_least32_t', 'uint_least8_t', 'uintptr_t', 'max_align_t', 'int8_t', 'fpos_t', 'ldiv_t', 'uint_fast32_t', 'uint_fast64_t', 'nullptr_t', 'int_fast32_t', 'uint_fast16_t', 'uint32_t', 'ptrdiff_t', 'int16_t', 'uint64_t', 'uint16_t', 'uint_fast8_t'])
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
lhs_cv_qualifier	CV qualifier on the lhs of a type.

AutosarC++18_10-A7.1.4

The register keyword shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
register_keyword	The register keyword shall not be used

AutosarC++18_10-A7.1.5

The auto specifier shall not be used apart from following cases: [1] to declare that a variable has the same type as return type of a function call, [2] to declare that a variable has the same type as initializer of non-fundamental type, [3] to declare parameters of a generic lambda expression, [4] to declare a function template using trailing return type syntax.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_for_loop_counter	Disables message if auto is used to declare a for-loop counter.	False
allow_generic_lambda_parameters	Allows auto as parameter type in a generic lambda	True
allow_nonfundamental_initializer	Allows auto to declare variables having a function call or initializer of non-fundamental type	True
allow_template_instance	Disables message if auto stands for a template instance.	False
allowed_contexts	Set of context predicates in which auto is allowed.	set[]
allowed_types	Set of types for which auto is allowed, given as name pattern, function, or LIR class name.	set[]
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cpp11_auto	Use of C++11 auto type specifier.

AutosarC++18_10-A7.1.6

The typedef specifier shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
typedef_specifier	The typedef specifier shall not be used.

AutosarC++18_10-A7.1.7

Each expression statement and identifier declaration shall be placed on a separate line.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_for_loop_init	Whether a multi-declaration is allowed in the for-init statement	True
allow_uninitialized_simple_type	Whether a multi-declaration of simple type is allowed when there is no initialization	False
enforcement		automated
ignore_stmts	Statements to be ignored when counting statements.	['Statement_Sequence', 'C_For_Loop']
level		required
target		implementation

Possible Messages

Name	Message
multi_declaration	Multiple declarators in single declaration
multiple_statements_per_line	Multiple statements per line.

AutosarC++18_10-A7.1.9

A class, structure, or enumeration shall not be declared in the definition of its type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
type_defined_in_declaration	Type defined in declaration.

AutosarC++18_10-A7.2.1

An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
asgn_not_an_enumerator	Assignment of value which does not correspond to an enumerator
asgn_out_of_enum_range	Assignment of value outside the range of the enumerators
conversion_creating_bad_enum_value	Expression does not correspond to an enumerator in {}

AutosarC++18_10-A7.2.2

Enumeration underlying base type shall be explicitly defined.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unbased_enum	Enumeration underlying base type shall be explicitly defined.

AutosarC++18_10-A7.2.3

Enumerations shall be declared as scoped enum classes.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unscoped_enum	Enumerations shall be declared as scoped enum classes.

AutosarC++18_10-A7.2.4

In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
incomplete_enum_init	Do not initialize enumerators other than the first, or initialize all

AutosarC++18_10-A7.3.1

All overloads of a function shall be visible from where it is called.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_assignment_operator_hiding	If set to true, we allow to hide the assignment operators, as pulling in the parent members using a using-declaration may allow assigning an instance of a parent class to a variable with the type of an inheriting class.	True
enforcement		automated
level		required
report_hidden_fields	Whether to report fields hidden by means of inheritance	False
report_hidden_methods	Whether to report methods hidden by means of inheritance	True
report_other_usages	Whether to find other usages (such as function calls, or taking a fp-reference) between multiple declarations for an identifier.	True
target		implementation

Possible Messages

Name	Message
declarations_surrounding_usage	Declarations straddle a usage
declarations_surrounding_using	Declarations straddle a using-declaration
hiding	{ } hides { }

AutosarC++18_10-A7.4.1

The asm declaration shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_assembler	Use of assembler.

AutosarC++18_10-A7.5.1

A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_const_reference	Whether only to report returns of parameters with reference to const.	True
target		implementation

Possible Messages

Name	Message
returning_reference_to_refparam	Returning reference(pointer to reference parameter.

AutosarC++18_10-A7.5.2

Functions shall not call themselves, either directly or indirectly.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_cycle	Cycle of recursive functions
recursive_function	Function is directly recursive

AutosarC++18_10-A7.6.1

Functions declared with the [[noreturn]] attribute shall not return.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
noreturn_violation	Do not return from a noreturn function.

AutosarC++18_10-A8.2.1

When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_trailing_return	Use trailing return-type syntax for function templates.

AutosarC++18_10-A8.4.1

Functions shall not be defined using the ellipsis notation.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_declarations	Allow declarations of functions with variable number of arguments (e.g. for existing library functions); only report definitions.	True
ignore_inherited	Do not report functions inheriting a variable number of arguments.	False
level		required
target		implementation

Possible Messages

Name	Message
ellipsis_parameter	Function definitions shall not use ellipsis

AutosarC++18_10-A8.4.2

All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
lambda_return_missing_value	Return without value in non-void lambda expression
missing_return	Non-void function needs return with value at end.
missing_return_in_lambda	Non-void lambda expression needs return with value at end.
return_missing_value	Return without value in non-void function

AutosarC++18_10-A8.4.4

Multiple output values from a function should be returned as a struct or tuple.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_modifying_calls	Whether a function may call a function on a parameter that modifies it, without regarding it as a violation here.	True
allow_move_parameters	Whether a move parameter (a rvalue reference) may be written to without triggering a violation. Note: move-constructors and move-assignments are allowed in either case.	False
allow_this_modification	Whether a function may call a member-function on a parameter that modifies this, that is the internal parameter state ('allow_modifying_calls' must be False if this is set to False)	True
enforcement		automated
exclude_constructors	Whether not to report parameters on constructors	True
exclude_operators	Whether not to report parameters on operator functions. Note: if set to True, this also sets 'exclude_shift_operators=True'.	False
exclude_shift_operators	Whether not to report parameters on shift operator functions (operator<<(), operator<<=(), operator>>(), and operator>>=).	False
level		advisory
only_report_multiple_output_parameters	Whether to only report output-parameters, if a function has more than one or also returns a value.	True
record_field_modification_threshold	A percentage of how many record fields may be modified, before the record pointer is considered an output parameter. 0 : every field modification counts as an output parameter 100: the record pointer is only regarded an output parameter if all field are modified	0
target		design

Possible Messages

Name	Message
output_parameter	Use return value instead of output parameter.

AutosarC++18_10-A8.4.5

"consume" parameters declared as X && shall always be moved from.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
exclude_move_assignments_and_constructors	If set to 'True', "consume" parameters in move-assignments and -constructors are not reported, even if they are not moved from.	True
level		required
target		design

Possible Messages

Name	Message
missing_move_consume	"consume" parameters shall always be moved from

AutosarC++18_10-A8.4.6

"forward" parameters declared as T && shall always be forwarded.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		design

Possible Messages

Name	Message
only_forward	"forward" parameters shall always be forwarded.

AutosarC++18_10-A8.4.7

"in" parameters for "cheap to copy" types shall be passed by value.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
exclude_operator_functions	If set to `True`, in parameters with "cheap to copy" types that are not passed by value are not reported for operator functions, as one might not have an option to change the signature.	False
level		required
size_threshold	Size of a type to be considered cheap to copy in number of words.	2
target		design
word_size	Size of a machine word in number of bytes. If None, uses the size of a pointer.	None

Possible Messages

Name	Message
input_parameter	Pass input parameters by value

AutosarC++18_10-A8.4.8

Output parameters shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_modifying_calls	Whether a function may call a function on a parameter that modifies it, without regarding it as a violation here.	True
allow_move_parameters	Whether a move parameter (a rvalue reference) may be written to without triggering a violation. Note: move-constructors and move-assignments are allowed in either case.	False
allow_this_modification	Whether a function may call a member-function on a parameter that modifies this, that is the internal parameter state ('allow_modifying_calls' must be False if this is set to False)	True
enforcement		automated
exclude_constructors	Whether not to report parameters on constructors	False
exclude_operators	Whether not to report parameters on operator functions. Note: if set to True, this also sets 'exclude_shift_operators=True'.	False
exclude_shift_operators	Whether not to report parameters on shift operator functions (operator<<(), operator<<=(), operator>>(), and operator>>=).	False
level		required
only_report_multiple_output_parameters	Whether to only report output-parameters, if a function has more than one or also returns a value.	False
record_field_modification_threshold	A percentage of how many record fields may be modified, before the record pointer is considered an output parameter. 0 : every field modification counts as an output parameter 100: the record pointer is only regarded an output parameter if all field are modified	0
target		design

Possible Messages

Name	Message
output_parameter	Don't use output parameters.

AutosarC++18_10-A8.4.9

"in-out" parameters declared as T & shall be modified.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
exclude_constructors	Whether not to report parameters on constructors	True
level		required
target		design

Possible Messages

Name	Message
complete_replacement	"in-out" completely replaced without being read.
missing_modify	"in-out" parameters not modified, consider making it const.

AutosarC++18_10-A8.5.1

In an initialization list, the order of initialization shall be following: (1) virtual base classes in depth and left to right order of the inheritance graph, (2) direct base classes in left to right order of inheritance list, (3) non-static data members in the order they were declared in the class definition.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	If provided, only messages of these types are reported.	[1719]
reported_severities	List of severities to display.	('error', 'warning', 'remark')
target		implementation
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

AutosarC++18_10-A8.5.2

Braced-initialization {}, without equals sign, shall be used for variable initialization.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
old_style_variable_init	Braced-initialization {}, without equals sign, shall be used for variable initialization.

AutosarC++18_10-A8.5.3

A variable of type auto shall not be initialized using {} or ={} braced-initialization.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
braced_auto_variable_init	A variable of type auto shall not be initialized using {} or ={} braced-initialization.

AutosarC++18_10-A9.5.1

Unions shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_tagged_unions	Whether tagged unions (nested in a struct that has an enum discriminator) should be allowed	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
union	Unions shall not be used

AutosarC++18_10-A10.1.1

Class shall not be derived from more than one base class which is not an interface class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_pure_interfaces	Whether C++ interfaces are allowed, i.e. classes with only pure virtual members.	True
level		required
target		implementation

Possible Messages

Name	Message
multiple_inheritance	Use of multiple inheritance.

AutosarC++18_10-A10.3.1

Virtual function declaration shall contain exactly one of the three specifiers: (1) virtual, (2) override, (3) final.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
disable_for_destructors	If set, destructors are not checked.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
invalid_combination	Use only one of {1} virtual, {2} override, {3} final.

AutosarC++18_10-A10.3.2

Each overriding virtual function shall be declared with the override or final specifier.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_final	If set to True, don't report overriding virtual functions declared with final.	True
enforcement		automated
ignore_destructors	If set to False, also report destructors. Note that not all compilers support this.	False
level		required
target		implementation

Possible Messages

Name	Message
missing_override	Override of functions is only permitted with keyword override/final.

AutosarC++18_10-A10.3.3

Virtual functions shall not be introduced in a final class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
virtual_method	Virtual functions shall not be introduced in a final class.
virtual_method_override	Virtual functions shall not be overridden without final in a final class.

AutosarC++18_10-A10.3.5

A user-defined assignment operator shall not be virtual.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
assignment_virtual	A user-defined assignment operator shall not be virtual.

AutosarC++18_10-A11.0.1

A non-POD type should be defined as class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
require_class_for_pod	Whether POD types should be classes as well	False
target		implementation

Possible Messages

Name	Message
cpp_struct	Use of struct in C++ unit.

AutosarC++18_10-A11.0.2

A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
no_function_member	Struct shall not provide any member functions or methods.
no_struct_as_base	A struct shall not be a base of another struct or class.
no_struct_inheritance	A struct shall not inherit from another struct or class.
non_public_member	Structs shall only have public data members.

AutosarC++18_10-A12.0.1

If a class declares a copy or move operation, or a destructor, either via "`=default`", "`=delete`", or via a user-provided declaration, then all others of these five special member functions shall be declared as well.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_defaulted_destructor_only	Allow classes with a defaulted destructor and no other special member functions.	False
allow_destructor_only	Allow all destructors without copy or move constructors	False
allow_empty_destructor	Allow empty destructors without copy or move constructors.	False
allow_missing_destructor	Suppress messages about missing destructors.	False
enforcement		automated
ignore_pod_classes	Whether POD classes should be checked at all	False
level		required
target		implementation

Possible Messages

Name	Message
missing_constructor_and_asgn	Class with destructor should also declare a copy or move constructor and assignment operator.
missing_copy_asgn	Class with copy constructor is missing copy assignment operator.
missing_copy_constructor	Class with copy assignment operator is missing copy constructor.
missing_destructor	Class with copy or move constructors or assignment operators should also declare a destructor.
missing_move_asgn	Class with move constructor is missing move assignment operator.
missing_move_constructor	Class with move assignment operator is missing move constructor.

AutosarC++18_10-A12.1.1

Constructors shall explicitly initialize all virtual base classes, all direct non-virtual base classes and all non-static data members.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
init_functions	Names of functions to be inspected as well when called directly from constructor.	{'Init', 'init'}
inspect_directly_called_methods	Inspect all methods directly called from constructor.	False
level		required
only_member_initializer_list	Only inspect member initializer list and not the constructor body/methods.	False
report_missing_field_constructors	Enables detection of constructors which rely on implicit field constructor calls.	True
target		implementation

Possible Messages

Name	Message
implicit_field_init	Field is only implicitly initialized in constructor.
missing_base_class_init	Base class is not explicitly initialized in constructor.
missing_field_init	Field is not initialized in constructor.

AutosarC++18_10-A12.1.2

Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_member_initializer_list	Only inspect member initializer list and not the constructor body/methods.	True
target		implementation

Possible Messages

Name	Message
nsdmi_mixed	Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type.

AutosarC++18_10-A12.1.3

If all user-defined constructors of a class initialize data members with constant values that are the same across all constructors, then data members shall be initialized using NSDMI instead.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_constructors_of_types	List of fully qualified type names which constructors are considered literals, without regard to the actual arguments.	[]
allow_literal_like_constructors	Whether to consider constructors that take only literals as arguments a literal. If set to true, this is checked recursively.	False
enforcement		automated
level		required
min_number_common_inits	Minimum number (inclusive) of common initializations to enforce use of NSDMI.	1
target		implementation

Possible Messages

Name	Message
use_nsmdi	Use NSDMI for common constant initializations ({}).

AutosarC++18_10-A12.1.4

All constructors that are callable with a single argument of fundamental type shall be declared explicit.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
consider_only_fundamental_types	Whether this check should be limited to single arguments of fundamental type or should also be applied to user defined types.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
constructor_missing_explicit	Constructor shall be declared explicit

AutosarC++18_10-A12.1.5

Common class initialization for non-constant members shall be done by a delegating constructor.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_copy_move_constructor	Whether to allow direct field initialization without a delegating constructor call for copy and move constructors.	False
enforcement		partially automated
level		required
target		implementation

Possible Messages

Name	Message
delegating_constructor	Use delegating constructor for common initialization.

AutosarC++18_10-A12.1.6

Derived classes that do not need further explicit initialization and require all the constructors from the base class shall use inheriting constructors.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_inheriting	Use inheriting constructors if possible.

AutosarC++18_10-A12.4.1

Destructor of a base class shall be public virtual, public override or protected non-virtual.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
base_class_destructor	Destructor of a base class shall be public virtual, public override or protected non-virtual.

AutosarC++18_10-A12.4.2

If a public destructor of a class is non-virtual, then the class should be declared final.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
non_final	If a public destructor of a class is non-virtual, then the class should be declared final.

AutosarC++18_10-A12.6.1

All class data members that are initialized by the constructor shall be initialized using member initializers.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
init_functions	Names of functions to be inspected as well when called directly from constructor.	('Init', 'init')
inspect_directly_called_methods	Inspect all methods directly called from constructor.	False
level		required
only_member_initializer_list	Only inspect member initializer list and not the constructor body/methods.	True
report_missing_field_constructors	Enables detection of constructors which rely on implicit field constructor calls.	True
target		implementation

Possible Messages

Name	Message
implicit_field_init	Field is only implicitly initialized in constructor.
missing_field_init	Field is not initialized in constructor.

AutosarC++18_10-A12.8.1

A copy constructor shall only initialize its base classes and the non-static members of the class of which it is a member.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_copy_constructor	Whether to report side effects on copy constructors.	True
report_move_constructor	Whether to report side effects on move constructors.	True
target		implementation

Possible Messages

Name	Message
copy_ctor_with_side_effect	Copy Constructor has side-effect
move_ctor_with_side_effect	Move Constructor has side-effect

AutosarC++18_10-A12.8.2

User-defined copy and move assignment operators should use user-defined no-throw swap function.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
asgn_no_swap	A move/copy assignment operator shall use a no-throw swap function.
asgn_not_nothrow	Used swap function is not no-throw.

AutosarC++18_10-A12.8.3

Moved-from object shall not be read-accessed.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		partially automated
inspect_class_field_moves	Detecting moved-from accesses on class fields requires heavy computation. This can be disabled by setting this option to False, but would result in false-positives in the context of class fields.	True
level		required
read_move_function_exceptions	Names of functions that are considered to leave the moved from objects in a well-specified state and are therefore except.	['std::unique_ptr::unique_ptr', 'std::unique_ptr::operator=', 'std::shared_ptr::shared_ptr', 'std::shared_ptr::operator=', 'std::weak_ptr::weak_ptr', 'std::weak_ptr::operator=', 'std::basic_filebuf::basic_filebuf', 'std::basic_filebuf::operator=', 'std::thread::thread', 'std::thread::operator=', 'std::unique_lock::unique_lock', 'std::unique_lock::operator=', 'std::shared_lock::shared_lock', 'std::shared_lock::operator=', 'std::promise::promise', 'std::promise::operator=', 'std::future::future', 'std::future::operator=', 'std::shared_future::shared_future', 'std::shared_future::operator=', 'std::packaged_task::packaged_task', 'std::packaged_task::operator=']
read_move_type_exceptions	Names of types that are considered to be left well-specified state after a move and are therefore except.	['std::basic_ios']
target		implementation

Possible Messages

Name	Message
moved_from_read	Don't read-access a moved-from object

AutosarC++18_10-A12.8.4

Move constructor shall not initialize its class members and base classes using copy semantics.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
move_constructor	Move constructor shall not initialize using copy semantics.

AutosarC++18_10-A12.8.6

Copy and move constructors and copy assignment and move assignment operators shall be declared protected or defined "=delete" in base class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
special_asgn	Copy and move assignment operators shall be declared protected or defined "=delete" in base class.
special_ctor	Copy and move constructors shall be declared protected or defined "=delete" in base class.

AutosarC++18_10-A12.8.7

Assignment operators should be declared with the ref-qualifier &.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
asgn_missing	Assignment operators should be declared with the ref-qualifier &.

AutosarC++18_10-A13.1.2

User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
user_defined_literal_naming	User-defined literals shall have a suffix matching "_[a-zA-Z]+".

AutosarC++18_10-A13.1.3

User defined literals operators shall only perform conversion of passed parameters.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
literal_side_effects	User-defined literals shall not have side effects.

AutosarC++18_10-A13.2.1

An assignment operator shall return a reference to "this".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
asgn_ref_this	An assignment operator shall return a reference to "this".

AutosarC++18_10-A13.2.2

A binary arithmetic operator and a bitwise operator shall return a "prvalue".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_bitwise_shift	Whether to allow non-basic values for operator>> or operator<<.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
arith_bitwise_basic_value	Binary arithmetic or bitwise operator shall return a basic value.

AutosarC++18_10-A13.2.3

A relational operator shall return a boolean value.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
relational_bool	A relational operator shall return a boolean value.

AutosarC++18_10-A13.3.1

A function that contains "forwarding reference" as its argument shall not be overloaded.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
overloaded_fref	Functions that contain forwarding reference parameters shall not be overloaded

AutosarC++18_10-A13.5.1

If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
array_access_non_const	No const version of operator[] implemented.

AutosarC++18_10-A13.5.2

All user-defined conversion operators shall be defined explicit.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
user_defined_conversion_explicit	All user-defined conversion operators shall be defined explicit.

AutosarC++18_10-A13.5.3

User-defined conversion operators should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
no_user_defined_conversion	User-defined conversion operators should not be used.

AutosarC++18_10-A13.5.4

If two opposite operators are defined, one shall be defined in terms of the other.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
implement_in_terms	Implement in terms of other operator

AutosarC++18_10-A13.6.1

Digit sequences separators ' shall only be used as follows: (1) for decimal, every 3 digits, (2) for hexadecimal, every 2 digits, (3) for binary, every 4 digits.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
digit_separator	Possibly unreadable digit separators.

AutosarC++18_10-A14.7.2

All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_type_declaration_file	Also allow specifications in the same file of a user-defined type for which the specialization is declared.	True
enforcement		automated
level		required
relax_if_template_only_declared	Allow specializations that are not declared in the header file if the template itself is only declared but not defined in the header.	False
target		implementation

Possible Messages

Name	Message
template_specialization_in_different_file	Specialization not declared in same file as primary template

AutosarC++18_10-A14.8.2

Overloaded function templates shall not be explicitly specialized.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
specialization_of_overloaded_template	Overloaded function templates shall not be explicitly specialized

AutosarC++18_10-A15.1.1

Only instances of types derived from std::exception shall be thrown.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throwing_std_exception	Only instances of types derived from std::exception shall be thrown.

AutosarC++18_10-A15.1.2

An exception object should not have pointer type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throwing_pointer	Exception object of pointer type

AutosarC++18_10-A15.1.3

All thrown exceptions should be unique.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
throwing_duplicate	All thrown exceptions should be unique.

AutosarC++18_10-A15.2.1

Constructors that are not noexcept shall not be invoked before program startup.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
static_field_only_noexcept	Constructor called that may throw an exception in static field.
static_variable_only_noexcept	Constructor called that may throw an exception in static variable.

AutosarC++18_10-A15.3.3

Main function and a task main function shall catch at least: base class exceptions from all third-party libraries used, std::exception and all otherwise unhandled exceptions.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		partially-automated
external_base_exceptions	Sequence of external/third-party exception base-classes that should be caught	[]
inspect_thread_main	Whether to also inspect thread main functions.	True
level		required
std_base_exceptions	Sequence of cpp-std exception base-classes that should be caught	['std::exception']
target		implementation

Possible Messages

Name	Message
missing_catch_all	Catch-all required around main program body
missing_catch_handler	Handler for {} needed

AutosarC++18_10-A15.3.4

Catch-all (ellipsis and std::exception) handlers shall be used only in (a) main, (b) task main functions, (c) in functions that are supposed to isolate independent components and (d) when calling third-party code that uses exceptions not according to AUTOSAR C++14 guidelines.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_extern_c_functions	Whether to allow a catch-all in `extern C` functions.	False
allow_in_functions_matching	If not None, a `re.compile()` ed object where functions whose qualified name matches the regex are allowed to contain catch-alls.	None
allow_in_main	Whether to allow a catch-all in the main() function.	True
allow_in_thread_main	Whether to allow a catch-all in thread-main functions.	True
allow_rethrow	Whether to allow a catch-all with a re-throw.	False
autosar_exception_model	The default model for what counts as an checked / unchecked exception.	<bauhaus.ir.autosar.exceptions.autosar_exceptions.AutosarExceptionModel object at 0x7dfa83c4650>
disallow_std_exception	Whether to consider catching the literal std::exception as a catch-all.	True
enforcement		non-automated
level		required
target		implementation

Possible Messages

Name	Message
catch_all	Use of catch(...).
catch_std_exception	Catching std::exceptions is too general.

AutosarC++18_10-A15.3.5

A class type exception shall always be caught by reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_class_types	Whether all types should be caught by reference or only class types.	True
target		implementation

Possible Messages

Name	Message
catch_without_reference	A class type exception shall always be caught by reference.

AutosarC++18_10-A15.4.1

Dynamic exception-specification shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

AutosarC++18_10-A15.4.2

If a function is declared to be noexcept, noexcept(true) or noexcept(<true condition>), then it shall not exit with an exception.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
generateViolationPath	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignoreConstructorDestructor	Whether to ignore escaping exceptions from constructors and destructors.	False
ignoreThrowingFunctions	Whether to ignore noexcept specification violations on function that actually throw an exception.	False
ignoreUnknownRoutines	Whether to ignore extern or only declared routines.	False
level		required
reportNoexceptFalseViolations	Whether to report cases where the function is declared noexcept(false), but no exceptions are ever thrown (directly or indirectly).	False
target		implementation

Possible Messages

Name	Message
implicit_noexcept_specViolationWithout	Function implicitly declared noexcept(false) but no exceptions will be thrown.
noexceptSpecViolationWith	Exception violates function's noexcept-specification.
noexceptSpecViolationWithout	Function declared noexcept(false) but no exceptions will be thrown.

AutosarC++18_10-A15.4.4

A declaration of non-throwing function shall contain noexcept specification.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
generateViolationPath	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignoreConstructorDestructor	Whether to ignore escaping exceptions from constructors and destructors.	False
ignoreThrowingFunctions	Whether to ignore noexcept specification violations on function that actually throw an exception.	True
ignoreUnknownRoutines	Whether to ignore extern or only declared routines.	True
level		required
reportNoexceptFalseViolations	Whether to report cases where the function is declared noexcept(false), but no exceptions are ever thrown (directly or indirectly).	True
target		implementation

Possible Messages

Name	Message
implicit_noexcept_specViolationWithout	Function implicitly declared noexcept(false) but no exceptions will be thrown.
noexceptSpecMissing	Explicit noexcept-specification missing.
noexceptSpecViolationWith	Exception violates function's noexcept-specification.
noexceptSpecViolationWithout	Function declared noexcept(false) but no exceptions will be thrown.

AutosarC++18_10-A15.4.5

Checked exceptions that could be thrown from a function shall be specified together with the function declaration and they shall be identical in all function declarations and for all its overriders.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
autosar_exception_model	The default model for what counts as an checked / unchecked exception.	18_03
enforcement		automated
ignore_definitions_if_declaration_documented	Instead of requiring the documentation to be repeated for every declaration (including definition), with this option set, the rule only checks the non-defining declarations if at least one non-defining declaration exists.	False
level		required
match_qualified_name	Matches the fully-qualified name when comparing documented exceptions with what can actually occur. If set to 'False', this rule will accept any suffix of the qualified name of an exception class as the documentation string.	True
target		implementation
throw_marker	The command to document an exception to be thrown.	@throw

Possible Messages

Name	Message
differing_documented	Documented exceptions differ from overridden method.
document_exception	Document checked exception {} using {}.
superflous_documented	Documented exceptions {} probably never thrown.

AutosarC++18_10-A15.5.1

All user-provided class destructors, deallocation functions, move constructors, move assignment operators and swap functions shall not exit with an exception. A noexcept exception specification shall be added to these functions as appropriate.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
routine_may_except	Function must not exit with an exception.
routine_not_noexcept	Function shall be explicitly declared noexcept if appropriate.

AutosarC++18_10-A15.5.2

Program shall not be abruptly terminated. In particular, an implicit or explicit invocation of std::abort(), std::quick_exit(), std::_Exit(), std::terminate() shall not be done.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
allowed_exceptions	Exceptions that are allowed to escape from destructors.	['bad_alloc', 'bad_cast', 'failure', 'runtime_error', 'system_error']
blacklist	Dictionary of header globbing to [list of] function name globbing(s) of forbidden functions.	dict(...)
constructors	Whether to consider constructors.	False
destructors	Whether to consider destructors.	True
enforcement		partially automated
generateViolationPath	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignoreConstructorDestructor	Whether to ignore escaping exceptions from constructors and destructors.	False
ignoreUnknownRoutines	Whether to ignore extern or only declared routines.	False
inspectAtExitHandlers	Whether to also inspect at_exit() handlers-functions.	True
inspectThreadMain	Whether to also inspect thread main functions.	True
level		required
reportOnlyOneExceptionPerFunction	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
target		implementation

Possible Messages

Name	Message
exceptionEscapingConstructor	Escaping exception from constructor.
exceptionEscapingDestructor	Escaping exception from destructor.
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionEscapingMain	Uncaught exception escaping from main or additional entry point
exceptionSpecificationViolation	Exception violates function's exception-specification.
forbiddenLibfuncCall	Call to forbidden function.

AutosarC++18_10-A15.5.3

The terminate() function shall not be called implicitly.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
allowed_exceptions	Exceptions that are allowed to escape from destructors.	['bad_alloc', 'bad_cast', 'failure', 'runtime_error', 'system_error']
constructors	Whether to consider constructors.	False
destructors	Whether to consider destructors.	True
enforcement		automated
generateViolationPath	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignoreConstructorDestructor	Whether to ignore escaping exceptions from constructors and destructors.	True
ignoreThrowingFunctions	Whether to ignore noexcept specification violations on function that actually throw an exception.	False
ignoreUnknownRoutines	Whether to ignore extern or only declared routines.	False
inspectAtExitHandlers	Whether to also inspect at_exit() handlers-functions.	True
inspectThreadMain	Whether to also inspect thread main functions.	True
level		required
reportNoexceptFalseViolations	Whether to report cases where the function is declared noexcept(false), but no exceptions are ever thrown (directly or indirectly).	False
reportOnlyOneExceptionPerFunction	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
required	Dict which lists required operations per resource. The mapping gives each case a description which maps to a dict for key "Required_Functions", "Resource_Parameter_Empty".	dict(...)
resources	Configuration of resources and operations on them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)
target		implementation

Possible Messages

Name	Message
exceptionEscapingConstructor	Escaping exception from constructor.
exceptionEscapingDestructor	Escaping exception from destructor.
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionEscapingMain	Uncaught exception escaping from main or additional entry point
exceptionSpecificationViolation	Exception violates function's exception-specification.
implicitNoexceptSpecViolationWithout	Function implicitly declared noexcept(false) but no exceptions will be thrown.
noexceptSpecViolationWith	Exception violates function's noexcept-specification.
noexceptSpecViolationWithout	Function declared noexcept(false) but no exceptions will be thrown.
possiblyRequiredOperation	This thread is possibly joinable on destructor call
requiredOperation	This thread is joinable on destructor call

AutosarC++18_10-A16.0.1

The pre-processor shall only be used for file inclusion and include guards.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
accept_conditional_includes	Whether to accept #ifs for conditional includes.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_macro_definition	Macros are only allowed for include guards
line_directive	The pre-processor shall only be used for file inclusion and include guards.
object_macro_definition	Macros are only allowed for include guards
pp_if	Conditional compilation is only allowed for include guards
pragma	The pre-processor shall only be used for file inclusion and include guards.
undef	The pre-processor shall only be used for file inclusion and include guards.

AutosarC++18_10-A16.2.1

The ', ", /*, //, \ characters shall not occur in a header file name or in #include directive.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
check_quotation_mark	Whether system includes should be checked for containing a ".	True
enforcement		automated
forbidden	The substrings to check for. " will be added for system-includes.	set(['//', '\\\\', """", /*])
level		required
target		implementation

Possible Messages

Name	Message
nonstandard_include_character	Non-standard character in #include directive

AutosarC++18_10-A16.6.1

#error directive shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
error	Use of #error

AutosarC++18_10-A16.7.1

The #pragma directive shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pragma	Use of #pragma

AutosarC++18_10-A17.0.1

Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
additional_reserved_identifiers	Names listed here are seen as violations as well.	set(['assert', 'errno', 'defined'])
allow_reserved_identifier_as_include_guard	If True, macros used for include guards are not checked for being a reserved identifier.	False
check_for_keyword	Whether macro names being keywords should be reported as well.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
macro_having_reserved_name	Definition of reserved identifier or standard library element
undef_of_reserved_name	#undef of reserved identifier or standard library element

AutosarC++18_10-A17.6.1

Non-standard entities shall not be added to standard namespaces.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
std_specialization_blacklist		('std::integral_constant', 'std::is_void', 'std::is_null_pointer', 'std::is_integral', 'std::is_floating_point', 'std::is_array', 'std::is_enum', 'std::is_union', 'std::is_class', 'std::is_function', 'std::is_pointer', 'std::is_lvalue_reference', 'std::is_rvalue_reference', 'std::is_member_object_pointer', 'std::is_member_function_pointer', 'std::is_fundamental', 'std::is_arithmetic', 'std::is_scalar', 'std::is_object', 'std::is_compound', 'std::is_reference', 'std::is_member_pointer', 'std::is_const', 'std::is_volatile', 'std::is_trivial', 'std::is_trivially_copyable', 'std::is_standard_layout', 'std::is_pod', 'std::is_literal_type', 'std::has_unique_object_representations', 'std::is_empty', 'std::is_polymorphic', 'std::is_abstract', 'std::is_final', 'std::is_aggregate', 'std::is_signed', 'std::is_unsigned', 'std::is_constructible', 'std::is_trivially_constructible', 'std::is_nothrow_constructible', 'std::is_default_constructible', 'std::is_trivially_default_constructible', 'std::is_nothrow_default_constructible', 'std::is_copy_constructible', 'std::is_trivially_copy_constructible', 'std::is_nothrow_copy_constructible', 'std::is_move_constructible', 'std::is_trivially_move_constructible', 'std::is_nothrow_move_constructible', 'std::is_assignable', 'std::is_trivially_assignable', 'std::is_nothrowAssignable', 'std::is_copyAssignable', 'std::is_trivially_copyAssignable', 'std::is_nothrow_copyAssignable', 'std::is_moveAssignable', 'std::is_trivially_moveAssignable', 'std::is_nothrow_moveAssignable', 'std::is_destructible', 'std::is_trivially_destructible', 'std::is_nothrow_destructible', 'std::has_virtual_destructor', 'std::is_swappable_with', 'std::is_swappable', 'std::is_nothrow_swappable_with', 'std::is_nothrow_swappable', 'std::alignment_of', 'std::rank', 'std::extent', 'std::is_same', 'std::is_base_of', 'std::is_convertible', 'std::is_nothrow_convertible', 'std::is_invocable', 'std::is_invocable_r', 'std::is_nothrow_invocable', 'std::is_nothrow_invocable_r', 'std::remove_cv', 'std::remove_const', 'std::remove_volatile', 'std::add_cv', 'std::add_const', 'std::add_volatile', 'std::remove_reference', 'std::add_lvalue_reference', 'std::add_rvalue_reference', 'std::remove_pointer', 'std::add_pointer', 'std::make_signed', 'std::make_unsigned', 'std::remove_extent', 'std::remove_all_extents', 'std::aligned_storage', 'std::aligned_union', 'std::decay', 'std::remove_cvref', 'std::enable_if', 'std::conditional', 'std::common_type', 'std::underlying_type', 'std::result_of', 'std::invoke_result', 'std::void_t', 'std::conjunction', 'std::disjunction', 'std::negation', 'std::endian', 'std::unary_function', 'std::binary_function')
std_specialization_whitelist		('std::common_type',)
target		implementation

Possible Messages

Name	Message
std_extension	Invalid addition to std namespace
std_specialization	Invalid std template specialization

AutosarC++18_10-A18.0.1

The C library shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
c_lib_header	Include <{}> instead of <{}>.
cpp_lib_header_with_suffix	Include <{}> instead of <{}>.

AutosarC++18_10-A18.0.2

The error state of a conversion from string to a numeric value shall be checked.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
check_functions	Fully qualified names of functions to check the state of the input stream.	['std::basic_ios::fail', 'std::ios_base::fail']
enforcement		automated
functions_to_check	Fully qualified names of functions after which a call to one of the check functions is required	['std::basic_istream::operator>>']
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	stdlib
symbols	Names of symbols which are forbidden.	['atof', 'atoi', 'atol', 'atoll']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.
missing_basic_ios_fail	Use basic_ios::fail() after reading from input streams.

AutosarC++18_10-A18.0.3

The library <clocale> (locale.h) and the setlocale function shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	locale
symbols	Names of symbols which are forbidden.	['setlocale']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++18_10-A18.1.1

C-style arrays should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
c_array_used	C-style arrays should not be used.

AutosarC++18_10-A18.1.2

The std::vector<bool> specialization shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
type_used	The type vector<bool> shall not be used.

AutosarC++18_10-A18.1.3

The std::auto_ptr type shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
type_used	The std::auto_ptr shall not be used.

AutosarC++18_10-A18.1.4

A pointer pointing to an element of an array of objects shall not be passed to a smart pointer of single object type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
type_used	{} shall not refer to an array type.

AutosarC++18_10-A18.1.6

All std::hash specializations for user-defined types shall have a noexcept function call operator.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
specialization_of_hash_except	std::hash specialization shall have noexcept call operator.

AutosarC++18_10-A18.5.1

Functions malloc, calloc, realloc and free shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_user_new_delete_operator	Whether to allow the library functions inside user defined new/delete operator overloads.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
stdlib_memory_function_call	Dynamic memory management function from <stdlib.h> is called.

AutosarC++18_10-A18.5.3

The form of delete operator shall match the form of new operator used to allocate the memory.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. See the base class for more options.	dict(...)
target		implementation

Possible Messages

Name	Message
possible_wrong_release	Resource possibly released using wrong function [allocation used {0}]
wrong_release	Resource released using wrong function [allocation used {0}]

AutosarC++18_10-A18.5.4

If a project has sized or unsized version of operator "delete" globally defined, then both sized and unsized versions shall be defined.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
only_sized	Only the sized version of operator delete is defined.
only_unsized	Only the unsized version of operator delete is defined.

AutosarC++18_10-A18.5.8

Objects that do not outlive a function shall have automatic storage duration.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
allow_smart_ptr_from_function_return	Whether to allow a smart pointer returned from a function as a local variable in a function. Calling the smart pointer constructors or the helper functions std::make_{shared,unique,..} will remain a violation.	True
enforcement		partially automated
level		required
target		implementation

Possible Messages

Name	Message
could_be_scoped	Local objects shall be allocated on the stack.

AutosarC++18_10-A18.9.1

The std::bind shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	functional
symbols	Names of symbols which are forbidden.	['std::bind']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	False

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity std::bind from <>.

AutosarC++18_10-A18.9.2

Forwarding values to other functions shall be done via: (1) std::move if the value is an rvalue reference, (2) std::forward if the value is forwarding reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
forwarding_forwarding_reference	Use std::forward if the value is a forwarding reference.
forwarding_rvalue_reference	Use std::move if the value is a rvalue reference.

AutosarC++18_10-A18.9.3

The std::move shall not be used on objects declared const or const&.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
std_move_const	Call to std::move with argument declared const/const&.

AutosarC++18_10-A21.8.1

Arguments to character-handling functions shall be representable as an unsigned char.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
functions	Names of functions that require an unsigned char cast.	['isalnum', 'isalpha', 'islower', 'isupper', 'isdigit', 'isxdigit', 'iscntrl', 'isgraph', 'isspace', 'isblank', 'isprint', 'ispunct', 'tolower', 'toupper']
level		required
target		implementation

Possible Messages

Name	Message
missing_uchar_cast	Missing explicit cast to unsigned char.
missing_uchar_cast_on_routine_literal	Missing explicit cast to unsigned char.

AutosarC++18_10-A23.0.1

An iterator shall not be implicitly converted to const_iterator.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
implicit_const_iterator	An iterator shall not be implicitly converted to const_iterator.

AutosarC++18_10-A26.5.1

Pseudorandom numbers shall not be generated using std::rand().

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
blacklist	Dictionary of header globbing to (list of) function name globbing(s) of forbidden functions.	dict(...)
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
forbidden_libfunc_call	Call to forbidden function.

AutosarC++18_10-A26.5.2

Random number engines shall not be default-initialized.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
additional_blacklist	Sequence of additional forbidden class names where calling the default constructor is forbidden.	[]
blacklist	Sequence of forbidden class names where calling the default constructor is forbidden.	('std::minstd_rand0', 'std::mt19937', 'std::mt19937_64', 'std::ranlux24_base<t>', 'std::ranlux48_base', 'std::ranlux24', 'std::ranlux48', 'std::knuth_b', 'std::default_random_engine', 'std::linear_congruential_engine', 'std::mersenne_twister_engine', 'std::subtract_with_carry_engine')
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
forbidden_default_constructor_call	Call to forbidden default constructor.

AutosarC++18_10-A27.0.4

C-style strings shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
c_string_used	C-style strings should not be used.

AutosarC++18_10-M0.1.1

There shall be no unreachable code.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_dead_if_branch_with_const_literal_condition	Whether bodies of if-branches with constant literal conditions should be reported.	True
target		implementation

Possible Messages

Name	Message
unreachable_code	Unreachable code

AutosarC++18_10-M0.1.2

A project shall not contain infeasible paths.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_do_while_false	Whether do ... while(0) should be reported.	True
report_while_true	Whether while(1) ... should be reported.	True
target		implementation

Possible Messages

Name	Message
dead_false_branch	Condition is always true: Else branch is dead
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Parameter condition is always true: Else branch is dead
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Parameter comparison to NULL is always true: Else branch is dead
dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Parameter comparison to NULL is always false: Then branch is dead
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Parameter condition is always false: Then branch is dead
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Condition is always false: Then branch is dead
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Variable condition is always true: Else branch is dead
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Variable condition is always false: Then branch is dead
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead

AutosarC++18_10-M0.1.3

A project shall not contain unused variables.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
ignore_types	Variables of types named here are ignored in this check. Globbing patterns are supported.	[]
level		required
only_check_unit_locals	Whether only global static variables and local variables should be checked.	False
report_global_constants	Whether unused global constants should be reported.	False
report_undefined_variables	Whether only-declared variables should be reported.	True
target		implementation
treat_initialization_as_use	Whether an explicit initialization should be considered a use of the variable.	True
treat_side_effect_constructors_as_use	Whether variables should be seen as used if they are of a class type and initialized through a call to a constructor having a side-effect, e.g. std::lock_guard	False

Possible Messages

Name	Message
unused_field	Unused field
unused_variable	Unused variable

AutosarC++18_10-M0.1.4

A project shall not contain non-volatile POD variables having only one use.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_if_address_taken	Allow using a variable only once if that use involves taking the address of the variable.	False
enforcement		automated
level		required
report_fields	Select whether fields used only once should be reported as well.	True
target		implementation

Possible Messages

Name	Message
field_referenced_only_once	{ } referenced only once
unreferenced_initialized_field	{ } initialized but not referenced
unreferenced_initialized_variable	{ } initialized but not referenced
variable_used_only_once	{ } referenced only once

AutosarC++18_10-M0.1.8

All functions with void return type shall have external side effect(s).

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
assume_effect_when_undefined	Whether all calls to undefined [external] functions should be seen as having a side-effect.	True
enforcement		automated
exceptions	Names of functions that should be excluded from the check.	main
exclude_constructors	If True, tolerate constructors with no side-effect.	False
exclude_virtual_destructors	If True, tolerate virtual destructors with no side-effect.	False
level		required
target		implementation

Possible Messages

Name	Message
void_func_without_side_effect	Void function has no external side-effect

AutosarC++18_10-M0.1.9

There shall be no dead code.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
allow_void_var	Whether {void}var; should be allowed or reported.	True
assume_effect_for_all_calls	Whether all calls should be seen as having a side-effect.	False
assume_effect_when_undefined	Whether all calls to undefined (external) functions should be seen as having a side-effect.	True
enforcement		automated
level		required
report_dead_initializations	Whether initialisations may be reported as dead.	True
report_do_while_false	Whether do ... while[0] should be reported.	True
report_while_true	Whether while[1] ... should be reported.	True
target		implementation
tolerate_void_cast	Whether a {void} cast is accepted.	True

Possible Messages

Name	Message
conditional_unused_def	Result of assignment is not used along some path(s) (disabled)
dead_false_branch	Redundant code, condition is always true
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Redundant code, parameter condition is always true
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Redundant code, parameter comparison to NULL is always true
dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Redundant code, parameter comparison to NULL is always false
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Redundant code, parameter condition is always false
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Redundant code, condition is always false
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Redundant code, variable condition is always true
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Redundant code, variable condition is always false
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead
init_used_in_other_isr	Initialization is only used in some interrupt handler
no_effect	Non-null statement without side-effect
removable_declaration	Declaration can be removed
removable_statement	Statement can be removed
unused_def	Dead (redundant) code: result of assignment is not used
unused_init	Unused initialization
used_in_other_isr	Result of assignment is only used in some interrupt handler

AutosarC++18_10-M0.1.10

Every defined function shall be called at least once.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
ignore_copy_assignment_operators	Whether uncalled copy assignment operators are allowed. Other rules like the "rule of three" might require copy assignment operators as part of good class design, even if there are currently no callers of the copy assignment operator.	False
ignore_copy_constructors	Whether uncalled copy constructors are allowed. Other rules like the "rule of three" might require copy constructors as part of good class design, even if there are currently no callers of the copy constructor.	False
level		required
only_check_internal_or_private_methods	Whether only function with internal linkage or private functions should be checked.	False
only_check_static_or_private_methods	Whether only static functions or private functions should be checked.	False
only_check_unit_locals	Whether only global static functions should be checked.	False
target		implementation

Possible Messages

Name	Message
defined_function_not_called	Defined function is not called

AutosarC++18_10-M0.2.1

An object shall not be assigned to an overlapping object.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
assignment_to_overlapping	An object shall not be assigned to an overlapping object

AutosarC++18_10-M0.3.1

Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
abstract_interpretation_div_by_zero	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
abstract_interpretation_overflow	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
argument_checks	Configuration of [dis]allowed values in addition to what iranalysis.config provides. This is a dict key -> dict with keys 'Functions', 'Parameter', and 'Allowed'/'Disallowed'. The parameter number counting starts at 0. Arguments passed in for this parameters are checked against the value specification given with '[Dis]allowed'.	dict(...)
enforcement		automated
execute_checks_covered_in_other_rules	Whether iranalysis-based checks covered by other Misra-C++:2008 rules should be executed here as well.	False
forbidden	Dict which lists forbidden operations per resource. The mapping gives each case a description which maps to a dict for key "Forbidden_Functions", "Mode_Parameter", "Mode".	dict(...)
level		required
modes	Describes when an allocating call opens a file in read or write mode. This is a dict resource -> dict with keys "Resource_Parameter", "Check_Write_Write", and "Modes" (being a dict with key "Read" and "Write")	dict(...)
perform_global_analysis	Whether global analysis (using iranalysis) should be done, or just a local analysis.	True
report_dead_initializations	Whether initialisations may be reported as dead.	True
report_do_while_false	Whether do ... while[0] should be reported.	True
report_while_true	Whether while[1] ... should be reported.	True
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)
target		implementation

Possible Messages

Name	Message
argumentViolation	Argument not within allowed values
arithmetic_out_of_bounds	Pointer arithmetic on {1} creates pointer outside array bounds of {0}
asgn_not_an_enumerator	Assignment of value which does not correspond to an enumerator
asgn_out_of_enum_range	Assignment of value outside the range of the enumerators
cast_overflow	Cast on result of arithmetic computation may cause overflow
cast_truncate	Cast may truncate value
cast_underflow	Cast on result of arithmetic computation may cause underflow
conditional_unused_def	Result of assignment is not used along some path(s)
deadCatch	Dead exception handler
dead_false_branch	Condition is always true: Else branch is dead
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Parameter condition is always true: Else branch is dead
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Parameter comparison to NULL is always true: Else branch is dead

dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Parameter comparison to NULL is always false: Then branch is dead
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Parameter condition is always false: Then branch is dead
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Condition is always false: Then branch is dead
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Variable condition is always true: Else branch is dead
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Variable condition is always false: Then branch is dead
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead
division_by_zero	Division by zero
double_free	Dynamic memory released here was already released earlier
escaping_address	Escaping address of local variable
forbidden_operation	The way in which this resource was allocated forbids this operation
init_used_in_other_isr	Initialization is only used in some interrupt handler
memory_leak	Call allocates leaking memory
mode_conflict	Same file used for both reading and writing
modulo_by_zero	Modulo by zero
null_deref	Pointer is NULL at dereference
out_of_bounds	Access into array is out of bounds
overflow	Arithmetic computation may cause overflow
pass_as_pointer_to_const_param	Passing uninitialized variable by pointer as function parameter with pointer-to-const type
possible_argumentViolation	Argument possibly not within allowed values
possible_division_by_zero	Possible division by zero
possible_double_free	Dynamic memory released here possibly already released earlier
possible_indirect_out_of_bounds	Pointer-indirect access through {1} might be out of bounds accessing {0}
possible_memory_leak	Call allocates possibly leaking memory
possible_mode_conflict	Same file possibly used for both reading and writing
possible_modulo_by_zero	Possible modulo by zero
possible_null_deref	Pointer may be NULL at dereference
possible_out_of_bounds	Access into array might be out of bounds
possible_stack_free	{ } possibly released by call to { } is a stack object
possible_uninit	Use of possibly uninitialized variable
possible_unrelated_ptr_comparison	Comparing possibly unrelated pointers
possible_unrelated_ptr_subtraction	Subtracting possibly unrelated pointers
possible_use_after_free	Dynamic memory possibly used after it was previously released

possible_wrong_release	Resource possibly released using wrong function (allocation used {0})
possiblyEscapingAddress	Possibly escaping address of local variable (as target of {1})
possiblyForbiddenOperation	The way in which this resource was allocated possibly forbids this operation
stack_free	{ } released by call to { } is a stack object
underflow	Arithmetic computation may cause underflow
uninit	Use of uninitialized variable
unrelated_ptr_comparison	Comparing unrelated pointers
unrelated_ptr_subtraction	Subtracting unrelated pointers
unused_def	Result of assignment is not used
unused_init	Unused initialization
use_after_free	Dynamic memory used after it was previously released
used_in_other_isr	Result of assignment is only used in some interrupt handler
wrong_release	Resource released using wrong function (allocation used {0})

AutosarC++18_10-M0.3.2

If a function generates error information, then that error information shall be tested.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
inspect_template_instances	Whether calls in template instances should be reported.	False
level		required
relevant_functions	If provided, only calls to these functions are inspected.	[]
target		implementation

Possible Messages

Name	Message
discarded_error_with_entity	Integral return value (possibly error code) discarded.
discarded_error_without_entity	Integral return value (possibly error code) discarded.

AutosarC++18_10-M0.4.2

Use of floating-point arithmetic shall be documented.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
float_arithetic	Use of floating-point arithmetic

AutosarC++18_10-M2.7.1

The character sequence /* shall not be used within a C-style comment.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_cpp_comments	Whether to look for /* in C++-style comments as well.	False
enforcement		automated
files_to_check	Files to apply this check to (Primary_File / User_Include_File / System_Include_File)	{'Primary_File', 'User_Include_File'}
level		required
target		implementation

Possible Messages

Name	Message
nested_c_comment	C-style comment containing /* sequence.

AutosarC++18_10-M2.10.1

Different identifiers shall be typographically unambiguous.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
distinguish_c_name_spaces	Whether to allow similar identifiers in different C name spaces (type tag/label/member/ordinary identifier)	True
distinguish_macro_names	Whether to allow macros to same similar names as other identifiers.	False
distinguish_type_names	Whether to allow types to have similar names as other identifiers. For example, a variable declaration "MyClass myClass;" is allowed by this option. Note: unlike the option 'distinguish_c_name_spaces', this option also distinguishes typedef names, not just struct/class tag names.	False
enforcement		automated
exclude_system_headers	Whether to exclude identifiers appearing in system headers	False
level		required
normalizations	Which pairs of characters should be seen as ambiguous	[('0', 'O'), ('1', 'l'), ('l', '1'), ('i', 'I'), ('5', 'S'), ('2', 'Z'), ('8', 'B'), ('rn', 'm'), ('n', 'h'), ('_', '_')]
target		architecture/design/implementation

Possible Messages

Name	Message
ambiguous	Identifiers are typographically ambiguous
casing	Identifiers only differ in casing

AutosarC++18_10-M2.13.2

Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
octal_escape_sequence	Use of octal escape sequence.
octal_literal	Use of octal literal.

AutosarC++18_10-M2.13.3

A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_only_hex_and_octal	Whether only hex and octal literals should be checked.	True
consider_conversion	Selects whether literals with an implicit conversion from signed to unsigned are reported.	False
consider_pp_expressions	Whether literals in #if conditions are checked.	False
enforcement		automated
level		required
report_all_unsuffixed_hex_and_octal	Whether hex and octal literals should always be reported when they are unsuffixed, independent of the environment	False
target		architecture/design/implementation

Possible Messages

Name	Message
missing_hex_suffix	Hexadecimal and octal constants require "U" suffix
missing_u_suffix	Constant of unsigned type requires "U" suffix

AutosarC++18_10-M2.13.4

Literal suffixes shall be upper case.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
lowercase_suffix	Literal suffix should be upper case

AutosarC++18_10-M3.1.2

Functions shall not be declared at block scope.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
local_function_declaration	Functions shall not be declared at block scope.

AutosarC++18_10-M3.2.1

All declarations of an object or function shall have compatible types.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_parameter_type_qualifier_differences	If enabled, do not report a violation for different top-level type qualifiers in parameters: int f(int p); int f(const int p) {} Top-level qualifiers are only relevant for the implementation of the function and have no effect on callers, so it is not necessary to replicate them in the header file.	False
check_parameter_types	Whether parameter types should be compared	False
check_undefined	Whether only-declared routines and variables should also be checked.	True
enforcement		automated
level		required
require_exact_match	Whether to check for identical or compatible types (for routine return types).	False
target		implementation

Possible Messages

Name	Message
decl_return_type_mismatch	Return type at declarations differs
decl_type_mismatch	Type at declarations differs
parameter_type_mismatch	Parameter type at definition differs from type at declaration
return_type_mismatch	Return type at definition differs from type at declaration
type_mismatch	Type at definition differs from type at declaration

AutosarC++18_10-M3.2.2

The One Definition Rule shall not be violated.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
class_struct_difference	{}
different_enumerators	{}
different_field_types	{}
different_fields	{}
general_odrViolation	{}

AutosarC++18_10-M3.2.3

A type, object or function that is used in multiple translation units shall be declared in one and only one file.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_only_one_decl	Enables message if there are multiple declarations but all in same file.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_declared_in_multiple_files	Function declared in more than one file.
function_declared_multiple_times	Function declared more than once.
type_declared_in_multiple_files	Type declared in more than one file.
type_declared_multiple_times	Type declared more than once.
variable_declared_in_multiple_files	Variable declared in more than one file.
variable_declared_multiple_times	Variable declared more than once.

AutosarC++18_10-M3.2.4

An identifier with external linkage shall have exactly one definition.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_undefined_specialized_template	Select whether undefined templates should be reported if specializations of them exist.	True
allow_unselected_declarations	Select whether to report declarations with no definition if they are unused.	True
allowed_undefined_functions	Regular expressions for functions which are tolerated without a definition, e.g. compiler intrinsics.	[_.*]
allowed_undefined_types	Regular expressions for types which are tolerated without a definition.	[_.*]
allowed_undefined_variables	Regular expressions for variables which are tolerated without a definition.	[_.*]
check_composite_types	Check class/struct/union types for having no definition even if they have no external linkage.	False
check_types_only_with_external_linkage	Whether the types should only be checked when they have external linkage	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
composite_defined_multiple_times	Composite type has multiple definitions.
enum_defined_multiple_times	Enumeration type has multiple definitions.
function_defined_multiple_times	Type, object or function has multiple definitions.
undefined_function	Type, object or function without definition
undefined_type	Type without definition
undefined_variable	Type, object or function without definition
variable_defined_multiple_times	Type, object or function has multiple definitions.

AutosarC++18_10-M3.3.2

If a function has internal linkage then all re-declarations shall include the static storage class specifier.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_decl_missing_static	Declaration should use static storage class specifier
function_def_missing_static	Definition should use static storage class specifier
variable_decl_missing_static	Declaration should use static storage class specifier
variable_def_missing_static	Definition should use static storage class specifier

AutosarC++18_10-M3.4.1

An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_moving_to_other_primary_file	When a variable is only used in a single file X, but currently implemented in a different file Y, this controls whether the check suggests to move it into X and make it static there.	False
check_classes	Whether to report structs/classes/unions which are only used in a single function or file	True
check_loop_counter	Whether to report for-loop counters being declared before the loop that could be declared in the for-init part instead from C99 on	False
enforcement		automated
exclude_function_locals	If True, variables that could even be function-local are not reported.	True
exclude_undefined	Whether only-declared symbols should be reported as well	True
level		required
move_global_const_into_function	Controls suggestions for global constants that could be declared locally in a function.	True
only_check_unit_locals	Whether only global static variables should be checked.	False
target		implementation

Possible Messages

Name	Message
locality_block	{ } can be declared in a more local scope.
locality_file	{ } can be declared locally in primary file.
locality_function	Global { } can be declared inside function.
locality_loop_init	{ } can be declared in the for-loop's initialization.
var_file_static	{ } can be declared static in primary file.

AutosarC++18_10-M3.9.1

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_missing_qualifiers	If True, tolerate differences in the use of explicit namespace/class qualifiers.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
parameter_type_tokens_mismatch	Type of redeclaration is not token-for-token identical
return_type_tokens_mismatch	Type of redeclaration is not token-for-token identical
variable_type_tokens_mismatch	Type of redeclaration is not token-for-token identical

AutosarC++18_10-M3.9.3

The underlying bit representations of floating-point values shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
float_bit_representation	Use of bit representation of a float value.

AutosarC++18_10-M4.5.1

Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_sizeof	Allows the use of sizeof(bool_var). The Misra rule forbids all uses of expressions of type bool except for those explicitly allowed; and does not allow sizeof. However this is likely an oversight; there is no good reason to prevent the use of sizeof(bool_var). This option also allows alignof(bool_var) in C++11.	False
allow_unevaluated_operator	Allows the use of boolean expressions in contexts where the expression is not evaluated. As of C++17, these are: typeid(bool_var), noexcept(bool_var) and decltype(bool_var). sizeof(bool_var) is controlled by the separate option 'allow_sizeof'.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bool_operand_in_bad_operator	Use of boolean operand in '{}' operator
bool_operand_outside_logical_and_relational_op	Use of boolean operand with integral promotion

AutosarC++18_10-M4.5.3

Expressions with type [plain] char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
char_operand_outside_comparison	Use of character operand in forbidden context

AutosarC++18_10-M4.10.1

NULL shall not be used as an integer value.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
null_as_int	Use of NULL as integer value

AutosarC++18_10-M4.10.2

Literal zero (0) shall not be used as the null-pointer-constant.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
require_nullptr	Require C++11 nullptr instead of macro NULL or literal zero (0).	False
target		implementation

Possible Messages

Name	Message
zero_as_null	Use of literal zero (0) as null-pointer-constant, use {} instead

AutosarC++18_10-M5.0.2

Limited dependence should be placed on C++ operator precedence rules in expressions.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_algebraic_order	Whether parens can be omitted for multiplicative operators as operands in additive operators	False
allow_redundant_parentheses_in_logical_operator	Allow redundant parentheses around unary operators when they appear as operand of '&&' or ' '. This option can be used to avoid a conflict with MisraC++-5.2.1, which requires parentheses in these instances.	True
enforcement		automated
level		required
report_unnecessary_parentheses	Controls whether unnecessary use of parentheses on the right side of assignments or around unary operators are reported.	True
target		implementation

Possible Messages

Name	Message
assignment_rhs_in_parens	No parentheses required for right-hand side of assignment
missing_parens_depends_on_precedence	Parentheses required to avoid dependence on precedence rules
unary_op_in_parens	No parentheses required for unary operator

AutosarC++18_10-M5.0.3

A value expression shall not be implicitly converted to a different underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes], [SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
target		implementation
type_category	Selection of type categories to consider for type changes inside same category.	[SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Result of cvalue expression implicitly converted to different underlying type
cast_changes_type_inside_category	Result of cvalue expression implicitly converted to different underlying type

AutosarC++18_10-M5.0.4

An implicit integral conversion shall not change the signedness of the underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_constant_conversions	If this option is enabled, the rule is relaxed to allow implicit conversions of constant integer expressions whenever the constant value fits into the target type.	True
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
report_signed_to_unsigned	Whether conversion from signed to unsigned shall be reported.	True
report_unsigned_to_signed	Whether conversion from unsigned to signed shall be reported.	True
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_from_signed_to_unsigned	Implicit integral conversion changes signedness of underlying type
cast_from_unsigned_to_signed	Implicit integral conversion changes signedness of underlying type

AutosarC++18_10-M5.0.5

There shall be no implicit floating-integral conversions.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes], [FloatingTypes]], [[FloatingTypes], [SignedTypes, UnsignedTypes]]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Implicit floating-integral conversion

AutosarC++18_10-M5.0.6

An implicit integral or floating-point conversion shall not reduce the size of the underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
report_narrowing	Whether narrowing casts should be reported.	True
report_widening	Whether widening casts should be reported.	False
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
size_source_category	Selection of type categories of the operand being cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
size_target_category	Selection of type categories appearing as target types of the cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
narrowing_cast	Implicit conversion reduces size of underlying type
widening_cast	Conversion to larger type

AutosarC++18_10-M5.0.7

There shall be no explicit floating-integral conversions of a cvalue expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, CharacterTypes], [FloatingTypes]], [[FloatingTypes], [SignedTypes, UnsignedTypes, CharacterTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Explicit floating-integral conversion of cvalue expression

AutosarC++18_10-M5.0.8

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
report_narrowing	Whether narrowing casts should be reported.	False
report_widening	Whether widening casts should be reported.	True
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
size_source_category	Selection of type categories of the operand being cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
size_target_category	Selection of type categories appearing as target types of the cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
narrowing_cast	Conversion to smaller type
widening_cast	Explicit conversion increases size of underlying type of cvalue expression

AutosarC++18_10-M5.0.9

An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
report_signed_to_unsigned	Whether conversion from signed to unsigned shall be reported.	True
report_unsigned_to_signed	Whether conversion from unsigned to signed shall be reported.	True
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_from_signed_to_unsigned	Explicit conversion changes signedness of underlying type of cvalue expression
cast_from_unsigned_to_signed	Explicit conversion changes signedness of underlying type of cvalue expression

AutosarC++18_10-M5.0.10

If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitop_small_without_cast	Bitwise operator requires cast to underlying type on result

AutosarC++18_10-M5.0.11

The plain char type shall only be used for the storage and use of character values.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
char_used_numerically	Plain char datatype used for non-character data.

AutosarC++18_10-M5.0.12

signed char and unsigned char type shall only be used for the storage and use of numeric values.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
numeric_char_used_as_character	Signed/unsigned char datatype used for character data.

AutosarC++18_10-M5.0.14

The first operand of a conditional-operator shall have type bool.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
nonbool_conditional_operator_condition	Condition must have type bool

AutosarC++18_10-M5.0.15

Array indexing shall be the only form of pointer arithmetic.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
indexing_only_on_identifiers	Report array indexing on pointers only for variables (`ptr[i]`), not for other pointer expressions (e.g. `get_ptr()[i]`). This option is meant to suppress the violations introduced by the BAUHAUS-12021 bugfix in version 6.9.6.	False
level		required
target		implementation

Possible Messages

Name	Message
array_indexing_on_pointer	Array indexing only allowed for arrays
pointer_arithmetic	Pointer arithmetic not allowed
pointer_increment_decrement	Pointer arithmetic not allowed

AutosarC++18_10-M5.0.16

A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
arithmetic_on_nonarray	Pointer arithmetic on {1} operates on non-array target {0}
arithmetic_out_of_bounds	Pointer arithmetic on {1} creates pointer outside array bounds of {0}
out_of_bounds	Access into array is out of bounds
possible_indirect_out_of_bounds	Pointer-indirect access through {1} might be out of bounds accessing {0}
possible_out_of_bounds	Access into array might be out of bounds

AutosarC++18_10-M5.0.17

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
possible_unrelated_ptr_subtraction	Subtracting possibly unrelated pointers
unrelated_ptr_subtraction	Subtracting unrelated pointers

AutosarC++18_10-M5.0.18

`>`, `>=`, `<`, `<=` shall not be applied to objects of pointer type, except where they point to the same array.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
possible_unrelated_ptr_comparison	Comparing possibly unrelated pointers
unrelated_ptr_comparison	Comparing unrelated pointers

AutosarC++18_10-M5.0.20

Non-constant operands to a binary bitwise operator shall have the same underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitop_with_different_operand_types	Non-constant operands of bitwise operator differ in underlying type
shortcut_bitop_with_different_operand_types	Non-constant operands of bitwise operator differ in underlying type

AutosarC++18_10-M5.0.21

Bitwise operators shall only be applied to operands of unsigned underlying type.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitop_on_enum	Bitwise operator on enum underlying type
bitop_on_signed	Bitwise operator on signed underlying type

AutosarC++18_10-M5.2.2

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cannot_cast_virtual_base	Cannot convert pointer to base class {} to pointer to derived class {} -- base class is virtual
missing_dynamic_cast_on_virtual_base	Use dynamic_cast on virtual base class

AutosarC++18_10-M5.2.3

Casts from a base class to a derived class should not be performed on polymorphic types.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cast_from_polybase_to_derived	Cast from polymorphic base class to derived class

AutosarC++18_10-M5.2.6

A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[FunctionPointerTypes], [VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes, NullPointerTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_category	Selection of type categories to consider for type changes inside same category.	[FunctionPointerTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion of function pointer to other type
cast_changes_type_inside_category	Conversion of function pointer to other function pointer type

AutosarC++18_10-M5.2.8

An object with integer type or pointer to void type shall not be converted to an object with pointer type.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, VoidPointerTypes], [ObjectPointerTypes, IncompletePointerTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, [T*]{void*x will not be reported.	True
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion from void* or integer to pointer type

AutosarC++18_10-M5.2.9

A cast should not convert a pointer type to an integral type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes], [SignedTypes, UnsignedTypes, BoolTypes, EnumTypes, CharacterTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion between pointer and integral type

AutosarC++18_10-M5.2.10

The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
forbid_all_operators	If True, forbids mixing with any kind of operator; else only with arithmetic operators.	False
level		required
target		implementation

Possible Messages

Name	Message
increment_mixed_with_operator	Increment or decrement mixed with other operators

AutosarC++18_10-M5.2.11

The comma operator, && operator and the || operator shall not be overloaded.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
invalid	Selection of disallowed operator overloads by name.	['operator&&', 'operator ', 'operator,']
invalid_kinds	Selection of disallowed operator overloads by operator kind.	[]
level		required
target		implementation

Possible Messages

Name	Message
forbidden_operator_overload	Overloaded version of comma operator or && or

AutosarC++18_10-M5.2.12

An identifier with array type passed as a function argument shall not decay to a pointer.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
decay	Array to pointer decay

AutosarC++18_10-M5.3.1

Each operand of the ! operator, the logical && or the logical || operators shall have type bool.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_in_conditions	If True, only logical operators inside conditions are checked.	False
target		implementation

Possible Messages

Name	Message
nonbool_logical_operator_operand	Operand of logical operator shall be of type bool

AutosarC++18_10-M5.3.2

The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_literals	If True, integer literals are also disallowed as operands.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unary_minus_on_unsigned	Unary minus applied to unsigned

AutosarC++18_10-M5.3.3

The unary & operator shall not be overloaded.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
invalid	Selection of disallowed operator overloads by name.	[]
invalid_kinds	Selection of disallowed operator overloads by operator kind.	[7]
level		required
target		implementation

Possible Messages

Name	Message
forbidden_operator_overload	Overloaded version of unary &

AutosarC++18_10-M5.3.4

Evaluation of the operand to the sizeof operator shall not contain side effects.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
side_effect_in_sizeof	Operand of "sizeof" shall not contain side effects

AutosarC++18_10-M5.8.1

The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_nonconst_cases	Whether a message should be printed for non-constant cases missing a range-checking if statement	False
target		implementation
use_essential_type	Whether essential type should be computed for allowed value range.	False

Possible Messages

Name	Message
excessive_shift	Right operand of shift operator has value {}, but should be in range 0 to {}
possible_excessive_shift	Right operand of shift operator should be checked to be in range 0 to {}

AutosarC++18_10-M5.14.1

The right-hand operand of a logical `&&` or `||` operator shall not contain side effects.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_volatile_access	Whether access to a volatile variable should be reported as side effect.	False
assume_effect_when_undefined	Whether all calls to undefined (external) functions should be seen as having a side-effect.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
call_with_side_effect	Call in right-hand operand of <code>&&</code> or <code> </code> may have side-effect
modifies_local_var	Right-hand operand of <code>&&</code> or <code> </code> modifies ' <code>{}</code> '
side_effect	Right-hand operand of <code>&&</code> or <code> </code> has side-effect

AutosarC++18_10-M5.17.1

The semantic equivalence between a binary operator and its assignment operator form shall be preserved.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_call_relation	If True, checks whether there is a call relation between binary and assignment version.	True
enforcement		automated
ignore_stream_operators	If True, allows definitions of operator <code><<()</code> and operator <code>>>()</code> without the corresponding assignment operator. Note this will also allow operator <code><<=()</code> and operator <code>>>=()</code> as they no longer have an equivalent binary form.	False
level		required
target		implementation

Possible Messages

Name	Message
missing_assignment_version	Missing overload for corresponding assignment version of operator
missing_binary_version	Missing overload for corresponding binary version of operator
missing_call_to_assignment_version	There is no call relation between this operator and its assignment version to ensure semantic equivalence
missing_call_to_binary_version	There is no call relation between this operator and its binary version to ensure semantic equivalence

AutosarC++18_10-M5.18.1

The comma operator shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_comma	Use of comma sequence operator.

AutosarC++18_10-M5.19.1

Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
const_expression_overflow	Overflow during evaluation of constant expression.

AutosarC++18_10-M6.2.1

Assignment operators shall not be used in sub-expressions.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
assignment_result_used	Assignment inside sub-expression.

AutosarC++18_10-M6.2.2

Floating-point expressions shall not be directly or indirectly tested for equality or inequality.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
float_equality_comparison	Testing a float for (in)equality.
float_indirect_equality_comparison	Indirectly testing a float for (in)equality.

AutosarC++18_10-M6.2.3

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_empty_macros	Whether a macro invocation before the ; is allowed if it expands to nothing.	False
allow_nonempty_macros	Whether a non-empty macro invocation before the ; is allowed.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
null_statement_not_isolated	Null statement not on a line by itself

AutosarC++18_10-M6.3.1

The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
loop_body_not_compound	Loop body shall be a statement sequence.
switch_body_not_compound	switch body shall be a statement sequence.

AutosarC++18_10-M6.4.1

An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if

statement.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
dangerous_else	Block or if statement required as else branch.
dangerous_then	Block required as then branch.

AutosarC++18_10-M6.4.2

All if ... else if constructs shall be terminated with an else clause.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
defensive_else	Controls if the else branch of if-cascade's should contain a suitable comment or not be empty.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
non_defensive_else	Last else in the if-cascade must not be empty or contain a comment.
unterminated_if_cascade	Last if in the if-cascade has no else.

AutosarC++18_10-M6.4.3

A switch statement shall be a well-formed switch statement.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
acceptable_end_items	This set characterizes allowed last items at the end of a switch case	('Exit_Switch', 'Throw_Expression')
acceptable_middle_items	This set characterizes allowed last items in the middle of a switch case	('Exit_Switch', 'Nondefault_Case_Label', 'Throw_Expression')
allow_as_first	Whether to allow the "default" case as the first case in a switch.	False
allow_complete_enum	Whether a switch having a case for all enumerators is tolerated without a default.	True
allow_empty_statements	Whether empty statements at the end of switch-case should be ignored.	False
allow_fallthrough_comment	Whether a comment with text "fall" and "through" is allowed to mark fallthrough.	True
allow_fallthrough_macro	Whether a macro called __fallthrough is allowed to mark fallthrough.	True
allow_if	Whether an if containing acceptable items at the end of both branches is accepted.	False
allow_noreturn_function_calls	Whether a function call to a [[noreturn]] function is considered a valid termination.	False
comment	The comment that indicates an intentional fallthrough. Can be a list of comments.	[]
disallow_multiple_compound_statements	Whether there can be more than one toplevel compound statement.	True
disallowed_nested_statements	Statement types not allowed inside compound statements within case-clause body.	None
disallowed_toplevel_statements	Statement types not allowed in case-clause body on toplevel.	('Named_Label', 'Unconditional_Branch', 'Declaration_Statement')
enforcement		automated
level		required
minimum_switch_cases	The number of cases a switch statement should at least have.	1
target		implementation

Possible Messages

Name	Message
case_label_nested	A switch label shall only be used in a switch.
default_not_last	"default" clause is not last in switch.
empty_default	Default clause should contain a statement or comment.
empty_switch	Switch has no "case" clause.
forbidden_compound_statement_in_case	At most one compound statement is allowed on toplevel in case-clauses.
forbidden_nested_statement_in_case	Statement is not allowed inside compound statement in case-clauses.
forbidden_toplevel_statement_in_case	Statement is not allowed outside a compound statement in case-clauses.
minimum_switch_cases	Switch has too little "case" clauses.
missing_default	Switch has no "default" clause.
switch_body_not_compound	A switch statement shall be a well-formed switch statement.
switch_not_well_formed	Switch has a statement before the first case label.
switch_over_bool	Switch expression is effectively Boolean.
unterminated_case	This switch-case is not terminated with {}, last statement is {}.
unterminated_empty_case	This switch-case is empty, not terminated with {}.

AutosarC++18_10-M6.4.4

A switch label shall only be used when the most closely-enclosing compound-statement is the body of a switch-statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
case_label_nested	A switch label shall only be used in a switch.

AutosarC++18_10-M6.4.5

An unconditional throw or break statement shall terminate every non-empty switch-clause.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
acceptable_end_items	This set characterizes allowed last items at the end of a switch case	('Exit_Switch', 'Throw_Expression')
acceptable_middle_items	This set characterizes allowed last items in the middle of a switch case	('Exit_Switch', 'Nondefault_Case_Label', 'Throw_Expression')
allow_empty_statements	Whether empty statements at the end of switch-case should be ignored.	False
allow_fallthrough_comment	Whether a comment with text "fall" and "through" is allowed to mark fallthrough.	True
allow_fallthrough_macro	Whether a macro called __fallthrough is allowed to mark fallthrough.	True
allow_if	Whether an if containing acceptable items at the end of both branches is accepted.	False
allow_noreturn_function_calls	Whether a function call to a [[noreturn]] function is considered a valid termination.	False
comment	The comment that indicates an intentional fallthrough. Can be a list of comments.	[]
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unterminated_case	This switch-case is not terminated with {}, last statement is {}.
unterminated_empty_case	This switch-case is empty, not terminated with {}.

AutosarC++18_10-M6.4.6

The final clause of a switch statement shall be the default clause.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_as_first	Whether to allow the "default" case as the first case in a switch.	False
allow_complete_enum	Whether a switch having a case for all enumerators is tolerated without a default.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
default_not_last	"default" clause is not last in switch.
empty_default	Default clause should contain a statement or comment.
missing_default	Switch has no "default" clause.

AutosarC++18_10-M6.4.7

The condition of a switch statement shall not have bool type.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
switch_over_bool	Switch condition shall not have bool type.

AutosarC++18_10-M6.5.2

If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
stepping_loop_uses_equality_check	Loop-counter shall not be tested with equality operator if not modified by -- or ++

AutosarC++18_10-M6.5.3

The loop-counter shall not be modified within condition or statement.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
loop_counter_modified_in_condition	Loop-counter shall not be modified within condition
modified_loop_counter	Loop-counter shall not be modified within loop body

AutosarC++18_10-M6.5.4

The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_constexpr	Allow constexpr as a constant <n>.	True
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
nonconst_loop_increment	Loop-counter shall be modified by one of: --, ++, -=n, or +=n (with constant n)

AutosarC++18_10-M6.5.5

A loop-control-variable other than the loop-counter shall not be modified within condition or expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
modified_loop_control_variable	Loop-control variable (other than counter) shall not be modified within condition or expression

AutosarC++18_10-M6.5.6

A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
nonbool_loop_control_variable	Loop-control variable (other than counter) shall have type bool

AutosarC++18_10-M6.6.1

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
goto_label_block	Label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

AutosarC++18_10-M6.6.2

The goto statement shall jump to a label declared later in the same function body.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
backwards_goto	Label referenced by a goto statement shall be declared later in same function.

AutosarC++18_10-M6.6.3

The continue statement shall only be used within a well-formed for loop.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
report_outside_for_loops	Whether to report continue statements in do..while/while loops	True
target		implementation

Possible Messages

Name	Message
continue_in_bad_loop	The continue statement shall only be used within a well-formed for loop

AutosarC++18_10-M7.1.2

A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_all_variables	Whether non-parameter variables should be checked as well.	False
enforcement		automated
ignore_forwarding_references	Whether to allow C++11 forwarding references (universal references) even if all template instances would also work with a const lvalue reference.	True
ignore_move_constructors	Whether to allow non-const rvalue references on constructors.	True
level		required
report_overrides	Whether to report violations on overrides. For virtual methods, this rule will report a violation only if the parameter can be marked const in all related methods in the inheritance hierarchy. Thus, the same violation is reported for each method in the inheritance hierarchy. You can set this option to False to report such violations only for the base methods.	True
target		implementation

Possible Messages

Name	Message
parameter_can_point_to_const	{ } can be declared as pointer/reference to const.

AutosarC++18_10-M7.3.1

The global namespace shall only contain main, namespace declarations and extern "C" declarations.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
global_symbol	Symbol not allowed in global namespace.

AutosarC++18_10-M7.3.2

The identifier main shall not be used for a function other than the global function main.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
nonglobal_function_named_main	The identifier main shall not be used for a function other than the global function main

AutosarC++18_10-M7.3.3

There shall be no unnamed namespaces in header files.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unnamed_namespace_in_header	Unnamed namespaces in header file

AutosarC++18_10-M7.3.4

Using-directives shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
using_directive	Using-directives shall not be used

AutosarC++18_10-M7.3.6

using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
using_declaration_in_header	Using-declaration in header file
using_namespace_in_header	Using-directive in header file

AutosarC++18_10-M7.4.1

All usage of assembler shall be documented.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_assembler	Usage of assembler shall be documented

AutosarC++18_10-M7.4.2

Assembler instructions shall only be introduced using the asm declaration.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pragma_asm	Assembler instructions shall only be introduced using the asm declaration

AutosarC++18_10-M7.4.3

Assembly language shall be encapsulated and isolated.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
asm_not_encapsulated	Assembly language shall be encapsulated and isolated.

AutosarC++18_10-M7.5.1

A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
global_reference_to_local_var	Address of local variable escapes via global variable.
parameter_reference_to_local_var	Address of local variable escapes via parameter.
returning_reference_to_local_var	Returning reference/pointer to local variable.

AutosarC++18_10-M7.5.2

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_longer_living_local	Whether assignment to a longer-living local variable should be accepted.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
possibly_leaking_reference_to_local_variable	Address of local variable is assigned to longer-living object.

AutosarC++18_10-M8.0.1

An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_in_for_loop_init	Whether a multi-declaration is allowed in the for-init statement	False
allow_uninitialized_simple_type	Whether a multi-declaration of simple type is allowed when there is no initialization	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
multi_declaration	Multiple declarators in single declaration

AutosarC++18_10-M8.3.1

Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
redefinition_uses_different_default_argument	Default argument differs from the one in redefined method

AutosarC++18_10-M8.4.2

The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
ignore_casing	Whether to ignore casing for the check.	False
include_redefinitions	Whether declarations of redefining methods should be considered together with the redefined ones.	True
inspect_only_defined_routines	If False, re-declarations of undefined functions are compared to earlier declarations as well.	False
level		required
target		implementation

Possible Messages

Name	Message
parameter_name_mismatch	Different name used for parameter

AutosarC++18_10-M8.4.4

A function identifier shall either be used to call the function or it shall be preceded by &.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value	
allowed	Qualified names of functions of which it is allowed to take the address implicitly, e.g. C++ I/O manipulators	['std::endl', 'std::flush', 'std::boolalpha', 'std::noboolalpha', 'std::showbase', 'std::noshowbase', 'std::showpoint', 'std::noshowpoint', 'std::showpos', 'std::noshowpos', 'std::skipws', 'std::noskipws', 'std::uppercase', 'std::nouppercase', 'std::unitbuf', 'std::nounitbuf', 'std::internal', 'std::left', 'std::right', 'std::dec', 'std::hex', 'std::oct', 'std::fixed', 'std::scientific', 'std::hexfloat', 'std::defaultfloat', 'std::ws', 'std::ends', 'std::resetiosflag', 'std::setiosflag', 'std::setbase', 'std::setfill', 'std::setprecision', 'std::setw', 'std::setw', 'std::get_money', 'std::put_money', 'std::get_time', 'std::put_time', 'std::quoted']	
enforcement		automated	
level		required	
target		implementation	

Possible Messages

Name	Message
implicit_routine_address	Taking address of function without &

AutosarC++18_10-M8.5.2

Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_any_zero_initialization	Whether {0} is allowed in nested aggregates as well.	False
allow_toplevel_zero_initialization	Whether {0} is allowed as top-level initialization for the complete aggregate	True
allowed_classes_without_braces	For the listed class types, don't require braces. This is useful for classes that transparently wrap a single contained value. For example, with this option "std::array<int, 3> a = {1,2,3};" is OK even though there could be two sets of braces (one for "std::array", the other and for its inner "int[3]" array).	['std::array']
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
incomplete_aggregate_initialization	Provide explicit initializer for each part of initialized entity
missing_aggregate_initialization_braces	Use braces to indicate structure of initialized entity

AutosarC++18_10-M9.3.1

const member functions shall not return non-const pointers or references to class-data.

Input: IR

Configuration

Name	Explanation	Value
enforcement		automated
include_copies_of_pointer_fields	Whether returning a pointer field should be reported as well.	True
inspect_only_const_methods	Whether all methods or only const methods should be checked.	True
level		required
only_report_references	Whether pointer and reference to field should be reported, or just references.	False
smart_pointer_names	Names that are considered smart pointers.	['unique_ptr', 'shared_ptr', 'weak_ptr']
target		implementation

Possible Messages

Name	Message
returning_nonconst_member_reference	Returning non-const reference/pointer to class data.

AutosarC++18_10-M9.3.3

If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_candidates_for_const	If False, avoid reporting methods that can be made const.	True
report_candidates_for_static	If False, avoid reporting methods that can be made static.	True
target		implementation
test_operators_for_static	If True, check whether a method can be made static is also applied to operator methods	False

Possible Messages

Name	Message
method_can_be_const	Method can be declared const.
method_can_be_static	Method can be declared static.

AutosarC++18_10-M9.6.1

When the absolute positioning of bits representing a bit-field is required, then the behaviour and packing of bit-fields shall be documented.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitfield	Usage of bit-fields shall be documented

AutosarC++18_10-M9.6.4

Named bit-fields with signed integer type shall have a length of more than one bit.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
signed_single_bitfield	Signed bit field shall be at least 2 bits long.

AutosarC++18_10-M10.1.1

Classes should not be derived from virtual bases.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
virtual_inheritance	Classes should not be derived from virtual bases.

AutosarC++18_10-M10.1.2

A base class shall only be declared virtual if it is used in a diamond hierarchy.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
virtual_inheritance_outside_diamond	A base class shall only be declared virtual if it is used in a diamond hierarchy.

AutosarC++18_10-M10.1.3

An accessible base class shall not be both virtual and non-virtual in the same hierarchy.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
base_class_being_virtual_and_nonvirtual	Has base class which is both virtual and non-virtual.

AutosarC++18_10-M10.2.1

All accessible entity names within a multiple inheritance hierarchy should be unique.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
ambiguous_member	All accessible entity names within a multiple inheritance hierarchy should be unique
use_of_ambiguous_name	{ } is ambiguous

AutosarC++18_10-M10.3.3

A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pure_redefinition	Pure redefinition of non-pure virtual function.

AutosarC++18_10-M11.0.1

Member data in non-POD class types shall be private.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_protected_members	If True, protected members are tolerated.	False
allowed	Specifies allowed fields as pairs (class name pattern, field name pattern). Example: {re.compile('.*'), re.compile('x')} to allow x in all classes.	[]
enforcement		automated
ignore_const_members	If True, non-private const members are tolerated.	False
ignore_pod	Whether fields in POD classes should be reported.	True
ignore_structs	Whether fields in structs should be reported.	False
ignore_templates	Whether fields in generic templates should be reported.	True
level		required
target		implementation

Possible Messages

Name	Message
protected_field	Member data in non-POD class types shall be private.
public_field	Member data in non-POD class types shall be private.

AutosarC++18_10-M12.1.1

An object's dynamic type shall not be used from the body of its constructor or destructor.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
constructor_using_dynamic_cast	Dynamic cast used in constructor/destructor.
constructor_using_typeid	Typeid on polymorphic class used in constructor/destructor.
constructor_using_virtual_call	Virtual call used in constructor/destructor.

AutosarC++18_10-M14.5.3

A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_copy_asgn_for_template	Class has template assignment operator but no copy assignment operator

AutosarC++18_10-M14.6.1

In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unqualified_dependent_member_access	Use qualifiers or this-> to select name that may be found in that dependent base

AutosarC++18_10-M15.0.3

Control shall not be transferred into a try or catch block using a goto or a switch statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
goto_into_try	Goto jumps into try or catch block
switch_into_try	Switch statement jumps into try or catch block

AutosarC++18_10-M15.1.1

The assignment-expression of a throw statement shall not itself cause an exception to be thrown.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throw_expression_raises_exception	Expression of throw may itself raise an exception

AutosarC++18_10-M15.1.2

NULL shall not be thrown explicitly.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throwing_null	NULL shall not be thrown explicitly

AutosarC++18_10-M15.1.3

An empty throw (throw;) shall only be used in the compound-statement of a catch handler.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
rethrow_outside_catch	Rethrow outside any catch block

AutosarC++18_10-M15.3.1

Exceptions shall be raised only after start-up and before termination of the program.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_only_one_exception_per_function	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
report_only_uncaught	Whether the check shall report all throws or just those not caught.	False
target		implementation

Possible Messages

Name	Message
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionRaisedInInitialization	Exception raised in initialization or finalization

AutosarC++18_10-M15.3.3

Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
handlerUsesField	Handler of a function-try-block shall not reference non-static members from this class or its bases

AutosarC++18_10-M15.3.4

Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
inspect_at_exit_handlers	Whether to also inspect at_exit() handlers-functions.	True
inspect_thread_main	Whether to also inspect thread main functions.	True
level		required
report_only_one_exception_per_function	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
target		implementation

Possible Messages

Name	Message
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionEscapingMain	Uncaught exception escaping from main or additional entry point

AutosarC++18_10-M15.3.6

Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
wrongCatchOrder	Catch handlers in wrong order.

AutosarC++18_10-M15.3.7

Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
catchAllNotLast	Catch-all shall occur as last handler.

AutosarC++18_10-M16.0.1

#include directives in a file shall only be preceded by other preprocessor directives or comments.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_extern_blocks	Whether to allow includes in `extern` blocks and not report a violation.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
include_after_code	#include should only be preceded by comments or preprocessor directives

AutosarC++18_10-M16.0.2

Macros shall only be #define'd or #undef'd in the global namespace.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
local_macro	#define or #undef not in global namespace

AutosarC++18_10-M16.0.5

Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
directives	Preprocessing directives to check for in macro arguments.	('#if', '#ifdef', '#ifndef', '#elif', '#else', '#endif', '#pragma', '#warning', '#error', '#line', '#include', '#include_next', '#ident', '#region', '#endregion', '#asm', '#endasm', '#define', '#undef')
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pp_directive_as_macro_arg	Preprocessing directive used in macro argument.

AutosarC++18_10-M16.0.6

In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unsafe_macro_param	Macro parameter not enclosed in parentheses

AutosarC++18_10-M16.0.7

Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
undefined_macro_used	Zero used for undefined preprocessing identifier

AutosarC++18_10-M16.0.8

If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
malformed_pp_directive	Malformed or unknown preprocessing directive

AutosarC++18_10-M16.1.1

The defined preprocessor operator shall only be used in one of the two standard forms.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
nonstandard_defined	Non-standard use of defined operator

AutosarC++18_10-M16.1.2

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_endif	The #endif for this directive is missing
missing_if	The #if for this directive is missing

AutosarC++18_10-M16.2.3

Include guards shall be provided.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
macro_name_restrictions	Python iterable of functions with parameters {file, define, macro} to perform additional checks on the macro used as include guard in the given file. Should return None if macro is accepted, else a message to print for the violation. The parameters are: PIR File node for the header, token for the macro #define or #pragma once, and the name of the macro or the string 'pragma once'.	None
target		implementation

Possible Messages

Name	Message
include_guard_missing	Missing include guard.
include_guard_partial	Include guard does not cover complete file.
include_guard_with_wrong_define	Bad include guard: #define uses wrong macro, should be {}.
include_guard_without_define	Incomplete include guard: missing #define {}.

AutosarC++18_10-M16.3.1

There shall be at most one occurrence of the # or ## operators in a single macro definition.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_only_concatenated_stringized	If True, only macros where a parameter is stringized and the result used for token concatenation are reported.	False
target		implementation

Possible Messages

Name	Message
concatenated_stringized	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator
more_than_one_macro_hash	Only one # or ## allowed in macro definition

AutosarC++18_10-M16.3.2

The # and ## operators should not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
hash_in_macro	The # and ## operators should not be used.

AutosarC++18_10-M17.0.2

The names of standard library macros and objects shall not be reused.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_locals	Whether parameters and local variables should also be checked	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
reused_macro_object_libname	The names of standard library macros and objects shall not be reused.

AutosarC++18_10-M17.0.3

The names of standard library functions shall not be overridden.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_function_declarations	Whether nondefining function declarations with library names are allowed	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
reused_routine_libname	The names of standard library functions shall not be overridden.

AutosarC++18_10-M17.0.5

The setjmp macro and the longjmp function shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	setjmp
symbols	Names of symbols which are forbidden.	['setjmp', 'longjmp']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++18_10-M18.0.3

The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	stdlib
symbols	Names of symbols which are forbidden.	['abort', 'exit', 'getenv', 'system']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++18_10-M18.0.4

The time handling functions of library <ctime> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	time
symbols	Names of symbols which are forbidden.	['clock', 'difftime', 'mktime', 'time', 'asctime', 'ctime', 'gmtime', 'localtime', 'strftime']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h} -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++18_10-M18.0.5

The unbounded functions of library <cstring> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	string
symbols	Names of symbols which are forbidden.	['strcpy', 'strcmp', 'strcat', 'strchr', 'strspn', 'strcspn', 'strpbrk', 'strrchr', 'strstr', 'strtok', 'strlen']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h} -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++18_10-M18.2.1

The macro offsetof shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	stddef
symbols	Names of symbols which are forbidden.	['offsetof']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++18_10-M18.7.1

The signal handling facilities of <csignal> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header (used for symbols like NULL and size_t which are defined in multiple headers)	['NULL', 'size_t']
enforcement		automated
header	Name of the header file which should not be used.	signal
level		required
target		implementation

Possible Messages

Name	Message
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.

AutosarC++18_10-M19.3.1

The error indicator errno shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	['errno', 'stdlib', 'stddef']
symbols	Names of symbols which are forbidden.	['errno']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <cerrno>.

AutosarC++18_10-M27.0.1

The stream input/output library <cstdio> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header (used for symbols like NULL and size_t which are defined in multiple headers)	['NULL', 'size_t']
enforcement		automated
header	Name of the header file which should not be used.	stdio
level		required
target		implementation

Possible Messages

Name	Message
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.

Rules in Group AutosarC++19_03

AutosarC++19_03-A0.1.1

A project shall not contain instances of non-volatile variables being given values that are never subsequently used.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_dead_initializations	Whether initialisations may be reported as dead.	True
target		implementation

Possible Messages

Name	Message
conditional_unused_def	Result of assignment is not used along some path(s)
init_used_in_other_isr	Initialization is only used in some interrupt handler
unused_def	Result of assignment is not used
unused_init	Unused initialization
used_in_other_isr	Result of assignment is only used in some interrupt handler

AutosarC++19_03-A0.1.2

The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed_functions	Calls to these functions are ignored.	frozenset(['strncat', 'memmove', 'memset', 'strcat', 'strcpy', 'memcpy', 'strncpy'])
enforcement		automated
inspect_template_instances	Whether calls in template instances should be reported.	False
level		required
target		implementation

Possible Messages

Name	Message
discarded_return_with_entity	Return value of function discarded.
discarded_return_without_entity	Return value of function discarded.

AutosarC++19_03-A0.1.3

Every defined function shall be called at least once.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_copy_assignment_operators	Whether uncalled copy assignment operators are allowed. Other rules like the "rule of three" might require copy assignment operators as part of good class design, even if there are currently no callers of the copy assignment operator.	False
ignore_copy_constructors	Whether uncalled copy constructors are allowed. Other rules like the "rule of three" might require copy constructors as part of good class design, even if there are currently no callers of the copy constructor.	False
level		required
only_check_internal_or_private_methods	Whether only function with internal linkage or private functions should be checked.	True
only_check_static_or_private_methods	Whether only static functions or private functions should be checked.	False
only_check_unit_locals	Whether only global static functions should be checked.	False
target		implementation

Possible Messages

Name	Message
defined_function_not_called	Defined function is not called

AutosarC++19_03-A0.1.4

There shall be no unused named parameters in non-virtual functions.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
inspect_nonvirtual_functions	Whether parameters of non-virtual functions should be checked.	True
inspect_virtual_functions	Whether virtual functions should be checked.	False
level		required
report_unnamed_parameters	Whether unnamed unused parameters should be reported as well.	False
target		implementation

Possible Messages

Name	Message
unused_parameter	Unused named parameter of non-virtual function

AutosarC++19_03-A0.1.5

There shall be no unused named parameters in the set of parameters for a virtual function and all the functions that override it.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
inspect_nonvirtual_functions	Whether non-virtual functions should be checked.	False
inspect_virtual_functions	Whether parameters of virtual functions should be checked. Violations will be reported in the base class when none of the derived classes make use of the parameter.	True
level		required
report_unnamed_parameters	Whether unnamed unused parameters should be reported as well.	False
target		implementation

Possible Messages

Name	Message
unused_parameter	Unused named parameter of virtual function

AutosarC++19_03-A0.1.6

A project shall not contain unused type declarations.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unused_type	Unused type declaration

AutosarC++19_03-A0.4.2

Type long double shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
forbidden_types	The ir.Logical Types not to be used e.g. `C_Long_Double_Type`.	['C_Long_Double_Type']
level		required
target		implementation

Possible Messages

Name	Message
type_used	The type shall not be used.

AutosarC++19_03-A1.1.1

All code shall conform to ISO/IEC 14882:2014 - Programming Language C++ and shall not use deprecated features.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	The set of messages regarding syntax and constraint violations.	set([2464, 2465, 1444, 2567, 2381, 2221, 1909, 1215])
reported_severities	List of severities to display.	('error', 'warning')
show_include_path	If True, violations regarding nested includes print the #include path.	False
standards	List of allowed frontend command-line options to select language standard.	['--strict', '-A']
target		implementation
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

Possible Messages

Name	Message
access_control	Number of access control declarations is higher than translation limit (which is {})
at_quick_exit_functions	Number of functions registered with at_quick_exit() is higher than translation limit (which is {})
atexit_functions	Number of functions registered with atexit() is higher than translation limit (which is {})
base_classes	Number of direct and indirect base classes is higher than translation limit (which is {})
call_arguments	Number of arguments is higher than translation limit (which is {})
catch_handlers	Number of catch handlers is higher than translation limit (which is {})
chars_in_line	Number of characters in logical line is higher than translation limit (which is {})
chars_in_string	Number of characters in string is higher than translation limit (which is {})
cpp_members	Number of members is higher than translation limit (which is {})
data_members	Number of data members is higher than translation limit (which is {})
direct_bases	Number of direct base classes is higher than translation limit (which is {})
enumerators	Number of enumerators is higher than translation limit (which is {})
expr_in_core_constant	Number of full expressions evaluated within a core constant expression is higher than translation limit (which is {})
external_identifiers	Number of external identifiers in this unit is higher than translation limit (which is {})
final_virtual_functions	Number of final overriding virtual functions is higher than translation limit (which is {})
friends	Number of friend declarations is higher than translation limit (which is {})
function_parameters	Number of parameters is higher than translation limit (which is {})
macro_arguments	Number of macro arguments is higher than translation limit (which is {})
macro_parameters	Number of macro parameters is higher than translation limit (which is {})

macros	Number of macro definitions in this unit is higher than translation limit (which is {})
member_initializers	Number of member initializers is higher than translation limit (which is {})
nested_blocks	Number of nested blocks is higher than translation limit (which is {})
nested_composites	Nesting level of structs and unions is higher than translation limit (which is {})
nested_declarators	Number of pointer/array/function declarators is higher than translation limit (which is {})
nested_externals	Number of nested external specifications is higher than translation limit (which is {})
nested_includes	Nesting level of #includes is higher than translation limit (which is {})
nested_preprocessor_ifs	Nesting level of #if is higher than translation limit (which is {})
nested_template_instantiations	Number of nested template instantiations is higher than translation limit (which is {})
object_size	Number of bytes in object is higher than translation limit (which is {})
parenthesized_declarators	Nesting level of parenthesized declarators is higher than translation limit (which is {})
parenthesized_expressions	Nesting level of parenthesized subexpressions is higher than translation limit (which is {})
placeholders	Number of placeholders is higher than translation limit (which is {})
recursive_constexpr	Number of recursive constexpr function invocations is higher than translation limit (which is {})
scope_qualifications	Number of scope qualifications is higher than translation limit (which is {})
significant_chars_external	Number of significant initial characters for external identifier is higher than translation limit (which is {})
significant_chars_internal	Number of significant initial characters for internal identifier is higher than translation limit (which is {})
static_members	Number of static members is higher than translation limit (which is {})
std_compile	You have to compile with option {}
switch_cases	Number of cases is higher than translation limit (which is {})
template_params	Number of template parameters is higher than translation limit (which is {})
throw_spec	Number of throw specifications is higher than translation limit (which is {})
variables_in_block	Number of variables in block is higher than translation limit (which is {})
virtual_bases	Number of virtual base classes is higher than translation limit (which is {})

AutosarC++19_03-A1.4.3

All code should compile free of compiler warnings.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	If provided, only messages of these types are reported.	None
reported_severities	List of severities to display.	('error', 'warning')
target		implementation
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	False

AutosarC++19_03-A2.3.1

Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_comments	Whether to allow characters not in the basic source character set in comments.	False
allow_in_unicode_strings	Whether to allow characters not in the basic source character set in unicode string literals.	True
allow_in_wide_strings	Whether to allow characters not in the basic source character set in wide string literals.	True
basic_characters_list	The basic character list as per rule.	\\\\\\ abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_\\{}\\[]\\#\\()\\<\\>\\%\\:\\\\.\\?*\\+\\-\\^\\&\\~\\!\\=\\,,\\\"\\`\\@\\@
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
forbidden_characters	Forbidden characters used.

AutosarC++19_03-A2.5.1

Trigraphs shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_comments	Allow the use of trigraphs in comments	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
trigraph_use	Use of trigraph.

AutosarC++19_03-A2.5.2

Digraphs should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
digraph_use	Digraph used.

AutosarC++19_03-A2.7.1

The character \ shall not occur as a last character of a C++ comment.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
line_splicing_in_cpp_comment	Line-splicing shall not be used in // comments.

AutosarC++19_03-A2.7.2

Sections of code shall not be "commented out".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
files_to_check	Files to be checked, e.g. Primary_File or File.	File
level		required
normal_text	Tokens that indicate non-code comments. This is the first step of checking a comment. If any of these words exist in the comment text, it is regarded as not containing code.	{'and', 'or', 'but', 'now', 'to', 'is', 'are', 'only', 'be', 'has', 'the', 'with', 'because', 'when', 'oder', 'und', '//', '#', 'AXIVION', '++++', '---', '===='})
normal_text_regex	Regular expression to match normal non-code text. This is the second step of checking a comment. If this regex matches the comment text, the comment is regarded as not containing code.	\b[\w\d_]+\s+[\w\d_]+\s+[\w\d_]+\b
suspicious	Tokens that indicate code in comments. This is the final step of checking a comment. If the first two steps did not exclude the comment from further inspection, the comment text is searched for any of these suspicious tokens. The comment is regarded as containing code, if any suspicious token is found.	{'=', '==', '>=', '<=', '[', ']', ':', '->', '->*', '/*', 'if', 'while', 'for', 'if (', 'while (', 'for (', '#pragma', '#else', '#endif', '#if', '#include', '++', '--')}
target		implementation

Possible Messages

Name	Message
code_in_c_comment	Sections of code shall not be commented out using C-style comments.
code_in_cpp_comment	Sections of code should not be commented out using C++ comments.

AutosarC++19_03-A2.7.3

All declarations of "user-defined" types, static and non-static data members, functions and methods shall be preceded by documentation.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_inherited	If True, a definition does not need documentation, if a corresponding declaration is documented.	False
allow_missing_documentation_on_private	If True, a class-member definition does not need documentation, if it is `private`.	False
allow_missing_documentation_on_protected	If True, a class-member definition does not need documentation, if it is `protected`.	False
doxygen_start	Start of a valid Doxygen comment.	{'/**', '///'}
enforcement		automated
ignore_defaulted	If True, defaulted function declarations are not checked for comments.	False
ignore_deleted	If True, deleted function declarations are not checked for comments.	False
ignore_redefinitions	If True, method redefinitions are not checked as they can 'inherit' the comment from the redefined method.	False
ignore_tool_comments	An optional compiled regular expression. Comments where this regex finds a matching substring are ignored in the search for a doxygen comment (e.g. control-comments of other tools).	None
level		required
node_types	IR node types to check for preceding Doxygen comment.	{'Routine_Definition', 'Routine_Declaration', 'Named_Type_Definition', 'Field_Definition'}
target		implementation

Possible Messages

Name	Message
missing_doxygen_comment_before_def	No Doxygen comment before declaration.

AutosarC++19_03-A2.8.1

A header file name should reflect the logical entity for which it provides declarations.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
file_type	Type of source_file file to examine.	User_Include_File
junk_characters	Characters that should not be considered in the filename to type similarity.	_-::
level		required
named_node_types	Physical IR node types to inspect for logical names to compare against the file name	{'Definition', 'No_Def_Declaration'}
similarity_threshold	Ratio of how similar the names need to be.	0.9
target		architecture/design/implementation

Possible Messages

Name	Message
source_file_name_not_type	The header should be named as a type it declares.

AutosarC++19_03-A2.8.2

An implementation file name should reflect the logical entity for which it provides definitions.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
file_type	Type of source_file file to examine.	Primary_File
junk_characters	Characters that should not be considered in the filename to type similarity.	_-::
level		advisory
named_node_types	Physical IR node types to inspect for logical names to compare against the file name	('Definition', 'No_Def_Declaration')
similarity_threshold	Ratio of how similar the names need to be.	0.9
target		architecture / design / implementation

Possible Messages

Name	Message
source_file_name_not_type	The file should be named as a type it declares.

AutosarC++19_03-A2.10.1

Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_classes	Whether identifiers in local classes can hide surrounding ones	True
allow_in_lambdas	Whether identifiers in lambdas can hide surrounding ones	True
allow_in_local_classes	Whether identifiers in local classes can hide surrounding ones	False
allow_in_namespaces	Whether identifiers in namespaces can hide surrounding ones	True
allow_parameter_in_function_declaration	Whether identifiers of parameters in function declarations may hide surrounding ones, given the function declaration uses a different identifier for the same parameter that does not hide a surrounding one.	False
check_inheritance_hiding	Whether to check for hiding by inheritance.	False
check_nested_classes	Whether symbols from inner classes should be reported as hiding symbols of outer classes.	False
distinguish_name_spaces	Whether C name spaces should be distinguished.	False
enforcement		automated
hiding_entities	LIR node types to consider symbols that hide other symbols.	('Variable', 'Typedef_Type', 'Field', 'Parameter')
level		required
maxlen	Number of significant characters (or None)	None
target		architecture/design/implementation
tolerate_constructor_parameters_hiding_fields	Allow constructor parameters to hide fields	False
tolerate_function_prototype_argument_hiding	Whether to tolerate arguments of a function prototype, that itself is a function argument, that have the same identifier as a argument of the outer function. If 'True' is given, the rule will still report cases, where argument identifiers match up and the function prototype refers to arguments of the outer function (e.g. by `decltype()`).	False
tolerate_macro_local_identifier	Whether an identifier being local to a macro is allowed to hide identifiers from outside the macro	False
unchecked_types	Deprecated legacy option, kept to avoid breaking old configurations.	('Routine_Call', 'Stack_Object_Definition')

Possible Messages

Name	Message
hiding	{ } hides { }

AutosarC++19_03-A2.10.4

The identifier name of a non-member object or function with static storage duration should not be reused.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_static_no_linkage	Whether to allow same identifiers for static objects without linkage (e.g. function local static objects)	False
enforcement		automated
func_filter	Restricts which functions are considered.	None
level		required
target		implementation
use_unqualified_names	If set to True, ns1::Type is considered equal to ns2::Type.	False
var_filter	Restricts which variables are considered.	None

Possible Messages

Name	Message
reused_global_function_name	Name of object or function with static storage duration reused.
reused_global_variable_name	Name of object or function with static storage duration reused.

AutosarC++19_03-A2.10.5

The identifier name of a non-member object or function with static storage duration should not be reused.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_static_no_linkage	Whether to allow same identifiers for static objects without linkage (e.g. function local static objects)	True
enforcement		automated
func_filter	Restricts which functions are considered.	None
level		required
target		design/architecture
use_unqualified_names	If set to True, ns1::Type is considered equal to ns2::Type.	True
var_filter	Restricts which variables are considered.	None

Possible Messages

Name	Message
reused_global_function_name	Name of object or function with static storage duration reused.
reused_global_variable_name	Name of object or function with static storage duration reused.

AutosarC++19_03-A2.10.6

A class or enumeration name shall not be hidden by a variable, function or enumerator declaration in the same scope.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
invert_findings	Whether to invert primary and secondary SLocs for violations	True
level		required
target		implementation

Possible Messages

Name	Message
reused_type	Type should not be hidden by a name in the same scope.

AutosarC++19_03-A2.11.1

Volatile keyword shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		design / implementation

Possible Messages

Name	Message
volatile_qualifier	The volatile type qualifier shall not be used

AutosarC++19_03-A2.13.1

Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed	List of allowed characters after backslash.	"\"?\\abfnrtv
enforcement		automated
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
nonstandard_escape_sequence	Use of non-standard escape sequence.

AutosarC++19_03-A2.13.2

Narrow and wide string literals shall not be concatenated.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
cpp11_mode	Use rules as defined in the C++11 standard.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
mixed_string_concatenation	Concatenation of mixed string encodings
narrow_wide_concat	Concatenation of narrow and wide string literal

AutosarC++19_03-A2.13.3

Type wchar_t shall not be used.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
forbidden_types	The ir.Logical Types not to be used e.g. `C_Long_Double_Type`.	['CPP_WChar_Type']
level		required
target		architecture/design/implementation/implementation

Possible Messages

Name	Message
type_used	The type shall not be used.

AutosarC++19_03-A2.13.4

String literals shall not be assigned to non-constant pointers.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	If provided, only messages of these types are reported.	[2464]
reported_severities	List of severities to display.	('error', 'warning', 'remark')
target		architecture / design / implementation
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

AutosarC++19_03-A2.13.5

Hexadecimal constants should be upper case.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
hex_literal	Hexadecimal constants should be upper case.

AutosarC++19_03-A2.13.6

Universal character names shall be used only inside character or string literals.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_comments	Whether to allow universal character names in comments.	False
allow_in_string_literals	Whether to allow universal character names in string literals.	True
enforcement		automated
level		required
target		architecture / design / implementation

Possible Messages

Name	Message
universal_character	Use of universal character names

AutosarC++19_03-A3.1.1

It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
accept_const_fields	Whether const-qualified static fields in header files should be tolerated.	True
accept_const_variables	Whether global const variables in header files should be tolerated.	True
enforcement		automated
find_non_external_function_declaration	Whether function declarations with non-external linkage should be reported.	True
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
function_definition_in_header	Definition in header file.
static_field_def_in_header	Definition in header file.
variable_definition_in_header	Definition in header file.

AutosarC++19_03-A3.1.2

Header files, that are defined locally in the project, shall have a file name' extension of one of: ".h", ".hpp" or "..hxx".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed_extensions	Which file extensions are allowed.	set[{'h', '.hxx', 'hpp'}]
enforcement		automated
file_type	The files to check the extensions of.	User_Include_File
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
illegal_file_extension	Use only certain extensions for file names.

AutosarC++19_03-A3.1.3

Implementation files, that are defined locally in the project, should have a file name extension of ".cpp".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed_extensions	Which file extensions are allowed.	set[{'cpp'}]
enforcement		automated
file_type	The files to check the extensions of.	Primary_File
level		advisory
target		architecture/design/implementation

Possible Messages

Name	Message
illegal_file_extension	Use only certain extensions for file names.

AutosarC++19_03-A3.1.4

When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_extern	Whether to consider only extern declared arrays	True
report_definitions	Whether definitions of array variables should also be reported	True
target		design/implementation

Possible Messages

Name	Message
unbounded_array	Array declaration with unknown size.

AutosarC++19_03-A3.1.6

Trivial accessor and mutator functions should be inlined.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
exclude_constructor_destructors	Whether to exclude constructors/destructors from inspected functions.	True
exclude_operator_functions	Whether to exclude operator functions from inspected functions.	False
exclude_virtual_functions	Whether to exclude virtual functions from inspected functions.	True
level		advisory
require_this_read_write	Whether to require a read/write on this for a function to be considered a trivial accessor/mutator.	False
target		design

Possible Messages

Name	Message
missing_inline	Trivial accessor and mutator functions should be inlined.

AutosarC++19_03-A3.3.1

Objects or functions with external linkage shall be declared in a header file.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_function_declaration_in_header	Object or function with external linkage shall be declared in a header file
missing_variable_declaration_in_header	Object or function with external linkage shall be declared in a header file

AutosarC++19_03-A3.3.2

Static and thread-local objects shall be constant-initialized.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed_effects_mask	If set, static and thread-local objects may also be initialized with a non-constexpr function/constructor call that only has the specified side-effects.	None
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
constexpr_init_variable	Static and thread-local objects shall be constant-initialized.

AutosarC++19_03-A3.9.1

Typedefs that indicate size and signedness should be used in place of the basic numerical types.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_floating_point	Whether floating point types without should not be reported.	True
enforcement		automated
ignore_inherited	If true, missing typeDefs in inherited methods are not reported.	False
level		required
target		implementation
treat_plain_char_as_numerical	Whether the char type without any signedness should be reported.	True

Possible Messages

Name	Message
missing_integer_TypeDef	Use of base type outside typeDef.
wrong_integer_TypeDef	Use of badly named typeDef for base type.

AutosarC++19_03-A4.5.1

Expressions with type enum shall not be used as operands to built-in operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_use_in_operator_calls	Whether enum arguments in calls to forbidden overloaded operators should be reported.	True
target		implementation

Possible Messages

Name	Message
enum_operand_outside_comparison	Use of enum operand in arithmetic or similar context

AutosarC++19_03-A4.7.1

An integer expression shall not lead to data loss.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
abstract_interpretation_overflow	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cast_overflow	Cast on result of arithmetic computation may cause overflow
cast_truncate	Cast may truncate value
cast_underflow	Cast on result of arithmetic computation may cause underflow
overflow	Arithmetic computation may cause overflow
underflow	Arithmetic computation may cause underflow

AutosarC++19_03-A4.10.1

Only nullptr literal shall be used as the null-pointer-constant.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
require_nullptr	Require C++11 nullptr instead of macro NULL or literal zero {0}.	True
target		architecture/design/implementation

Possible Messages

Name	Message
null_constant	Only nullptr literal shall be used as the null-pointer-constant.
zero_as_null	Use of literal zero {0} as null-pointer-constant, use {} instead

AutosarC++19_03-A5.0.1

The value of an expression shall be the same under any order of evaluation that the standard permits.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_exceptions	Ignore possible exceptions in the computation of side effects. (this option only has an effect if report_calls is enabled)	False
level		required
report_calls	If True, unsequenced function calls are reported.	True
target		implementation

Possible Messages

Name	Message
unsequenced_call_volatile	Function calls unsequenced with volatile access
unsequenced_calls	Unsequenced function calls
unsequenced_read_write	Unsequenced read and write accesses
unsequenced_volatiles	Unsequenced volatile accesses
unsequenced_writes	Unsequenced writes

AutosarC++19_03-A5.0.2

The condition of an if-statement and the condition of an iteration-statement shall have type bool.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_in_conditions	If True, only logical operators inside conditions are checked.	True
report_do_while_false	Whether do ... while[0] should be reported.	True
report_while_true	Whether while[1] ... should be reported.	True
target		implementation

Possible Messages

Name	Message
nonbool_if_condition	Condition must have type bool
nonbool_logical_operator_operand	Sub-condition must have type bool
nonbool_loop_condition	Condition must have type bool

AutosarC++19_03-A5.0.3

The declaration of objects should contain no more than two levels of pointer indirection.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
max_levels	Maximum number of allowed pointer-indirection levels.	2
target		implementation

Possible Messages

Name	Message
too_many_pointer_levels	More than {} level{} of pointer indirection

AutosarC++19_03-A5.0.4

Pointer arithmetic shall not be used with pointers to non-final classes.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
consider_non_base_final	Whether to consider a class that is not used as a base class as a sort-of final class.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
non_final_pointer_arithmetic	Pointer arithmetic on non-final classes not allowed

AutosarC++19_03-A5.1.1

Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_nonconst_variable_initialization	If set to `True`, allow the initialization of a non-const variable with a string literal in addition to the initialization of a const variable. A reassignment of such a variable is still a violation.	False
allowed	Literal values that are ok.	{0.0, 1.0, 2.0}
allowed_contexts	Optional set of PIR classes or functions {[node] -> bool} for allowed contexts, e.g. Case_Label.	set[<function allow_in_type_initialization at 0x7dfa84459b0>]
allowed_logging_contexts	List of fully qualified function types that are considered logging contexts [using operator<>]. If this is non-empty, `std::throw()` is considered a valid logging context as well.	['std::basic_ostream']
allowed_string_contexts	Optional set of PIR classes or functions {[node] -> bool} for allowed contexts.	set[<function allow_in_type_initialization at 0x7dfa84459b0>]
allowed_strings	Literal values that are ok.	["' '"]
enforcement		partially automated
exceptions	Optional predicate to filter out cases that should be allowed as an exception. The predicate takes a literal node and returns True if the given node is such an exception.	None
exclude_pp_literals	If True, literals in conditions of #if are ignored.	True
exclude_single_uses	If True, report only literals used more than once.	False
level		required
target		implementation

Possible Messages

Name	Message
magic_number	Use of magic literal.
magic_number_without_token	Use of magic literal.
magic_string	Use of magic string literal.
possible_magic_number	Potential use of magic literal.

AutosarC++19_03-A5.1.2

Variables shall not be implicitly captured in a lambda expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
implicit_lambda_capture	Variables shall not be implicitly captured in a lambda expression.

AutosarC++19_03-A5.1.3

Parameter list (possibly empty) shall be included in every lambda expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_parameters	Include a (possibly empty) parameter list

AutosarC++19_03-A5.1.6

Return type of a non-void return type lambda expression should be explicitly specified.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
missing_explicit_return	Add an explicit return type for the lambda.

AutosarC++19_03-A5.1.7

The underlying type of lambda expression shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
underlying_lambda_type	The underlying type of lambda expression shall not be used.

AutosarC++19_03-A5.1.8

Lambda expressions should not be defined inside another lambda expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
nested_lambda	Lambda expressions should not be defined inside another lambda expression

AutosarC++19_03-A5.1.9

Identical unnamed lambda expressions shall be replaced with a named function or a named lambda expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_empty_lambdas	If set to True, this rule will not report duplicates of the trivial empty lambda.	False
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
identical_unnamed_lambda	Identical unnamed lambdas.

AutosarC++19_03-A5.2.1

dynamic_cast should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
dynamic_cast	dynamic_cast should not be used.

AutosarC++19_03-A5.2.2

Traditional C-style casts shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_void_cast	If True, (void) is always allowed, else only for return value of calls.	False
allow_void_cast_on_call	If True, (void) is allowed, for return value of calls.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
c_cast	Use of C-style cast in C++ unit.

AutosarC++19_03-A5.2.3

A cast shall not remove any const or volatile qualification from the type of a pointer or reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
check_const	Whether casts affecting const-qualifiers should be reported	True
check_volatile	Whether casts affecting volatile-qualifiers should be reported	True
enforcement		automated
level		required
only_top_level	Whether the check only applies to the top-level pointee or recursively to deeper pointer levels as well	False
target		implementation

Possible Messages

Name	Message
cast_removes_const	Cast removes const qualification
cast_removes_volatile	Cast removes volatile qualification
implicit_cast_removes_const	Implicit cast removes const qualification
implicit_cast_removes_volatile	Implicit cast removes volatile qualification

AutosarC++19_03-A5.2.4

reinterpret_cast shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
reinterpret_cast	reinterpret_cast should not be used.

AutosarC++19_03-A5.2.6

The operands of a logical && or || shall be parenthesized if the operands contain binary operators.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allowed	Can be used to name additional allowed IR node types for the operands.	None
enforcement		automated
level		required
require_postfix_expression	Whether postfix or primary expressions are required as operands.	False
target		implementation

Possible Messages

Name	Message
missing_parens_for_logical_operator_operand	Operand of && or must be parenthesized.

AutosarC++19_03-A5.3.1

Evaluation of the operand to the typeid operator shall not contain side effects.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		non-automated
level		required
target		implementation

Possible Messages

Name	Message
side_effect_in_typeid	Operand of "typeid" shall not contain side effects

AutosarC++19_03-A5.6.1

The right hand operand of the integer division or remainder operators shall not be equal to zero.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
abstract_interpretation_div_by_zero	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
division_by_zero	Division by zero
modulo_by_zero	Modulo by zero
possible_division_by_zero	Possible division by zero
possible_modulo_by_zero	Possible modulo by zero

AutosarC++19_03-A5.10.1

A pointer to member virtual function shall only be tested for equality with null-pointer-constant.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
member_virtual_function_comparison	Comparison of a member virtual function with non-nullptr.

AutosarC++19_03-A5.16.1

The ternary conditional operator shall not be used as a sub-expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_nested_conditional_operator	Use of nested conditional operator.

AutosarC++19_03-A6.4.1

Every switch statement shall have at least one case-clause.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
minimum_switch_cases	The number of cases a switch statement should at least have.	2
target		implementation

Possible Messages

Name	Message
empty_switch	Switch has no "case" clause.
minimum_switch_cases	Switch has too little "case" clauses.

AutosarC++19_03-A6.5.1

A for-loop that loops through all elements of the container and does not use its loop-counter shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
exclude_simple_for_loops	If set to `True` this rule will not report unused loop counters in the simple cases where the C for-loop only references loop-counters in its condition but no other variables.	True
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
could_be_range_based	For loop could be a range-based for loop.
unused_loop_counter	For loop does not use its loop counter.

AutosarC++19_03-A6.5.2

A for loop shall contain a single loop-counter which shall not have floating type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
loop_float_counter	Loop-counter of for loop shall not have floating type
loop_missing_counter	For loop has no loop-counter
loop_multiple_counters	For loop shall have only a single loop-counter

AutosarC++19_03-A6.5.3

Do statements should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
do_while_loop	Do statements should not be used.

AutosarC++19_03-A6.5.4

For-init-statement and expression should not perform actions other than loop-counter initialization and modification.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
loop_side_effect	Side effect in loop init/expression.
non_loop_counter_initialized	Non loop counter initialized.
non_loop_counter_modified	Non loop counter modified.

AutosarC++19_03-A6.6.1

The goto statement shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
goto	Do not use goto.

AutosarC++19_03-A7.1.1

Constexpr or const specifiers shall be used for immutable data declaration.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_pointer_variables	Whether variables of pointer type should be ignored.	False
level		required
only_check_unit_locals	Whether only local variables and global static variables should be checked.	False
only_immutable_data	Whether only declarations with immutable data should be checked.	True
target		implementation

Possible Messages

Name	Message
variable_missing_const	An immutable variable shall be const/constexpr qualified.

AutosarC++19_03-A7.1.3

CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_nonpointer_typedefs	Whether to allow the qualifier on the rhs of a type, if it is a non-pointer typedef.	False
allowed_typedefs	Set of names for typedefs/usings that are allowed (e.g. int32_t).	set(['int32_t', 'uint_least64_t', 'intptr_t', 'uintmax_t', 'int_fast16_t', 'intmax_t', 'int_fast8_t', 'int64_t', 'size_t', 'int_fast64_t', 'time_t', 'uint8_t', 'lldiv_t', 'int_least8_t', 'div_t', 'uint_least16_t', 'clock_t', 'uint_least32_t', 'int_least64_t', 'int_least16_t', 'int_least32_t', 'uint_least8_t', 'uintptr_t', 'max_align_t', 'int8_t', 'fpos_t', 'ldiv_t', 'uint_fast32_t', 'uint_fast64_t', 'nullptr_t', 'int_fast32_t', 'uint_fast16_t', 'uint32_t', 'ptrdiff_t', 'int16_t', 'uint64_t', 'uint16_t', 'uint_fast8_t'])
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
lhs_cv_qualifier	CV qualifier on the lhs of a type.

AutosarC++19_03-A7.1.4

The register keyword shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
register_keyword	The register keyword shall not be used

AutosarC++19_03-A7.1.5

The auto specifier shall not be used apart from following cases: (1) to declare that a variable has the same type as return type of a function call, (2) to declare that a variable has the same type as initializer of non-fundamental type, (3) to declare parameters of a generic lambda expression, (4) to declare a function template using trailing return type syntax.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_for_loop_counter	Disables message if auto is used to declare a for-loop counter.	False
allow_generic_lambda_parameters	Allows auto as parameter type in a generic lambda	True
allow_nonfundamental_initializer	Allows auto to declare variables having a function call or initializer of non-fundamental type	True
allow_template_instance	Disables message if auto stands for a template instance.	False
allowed_contexts	Set of context predicates in which auto is allowed.	set[]
allowed_types	Set of types for which auto is allowed, given as name pattern, function, or LIR class name.	set[]
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cpp11_auto	Use of C++11 auto type specifier.

AutosarC++19_03-A7.1.6

The typedef specifier shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
typedef_specifier	The typedef specifier shall not be used.

AutosarC++19_03-A7.1.7

Each expression statement and identifier declaration shall be placed on a separate line.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_for_loop_init	Whether a multi-declaration is allowed in the for-init statement	True
allow_uninitialized_simple_type	Whether a multi-declaration of simple type is allowed when there is no initialization	False
enforcement		automated
ignore_stmts	Statements to be ignored when counting statements.	['Statement_Sequence', 'C_For_Loop']
level		required
target		implementation

Possible Messages

Name	Message
multi_declaration	Multiple declarators in single declaration
multiple_statements_per_line	Multiple statements per line.

AutosarC++19_03-A7.1.9

A class, structure, or enumeration shall not be declared in the definition of its type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
type_defined_in_declaration	Type defined in declaration.

AutosarC++19_03-A7.2.1

An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
asgn_not_an_enumerator	Assignment of value which does not correspond to an enumerator
asgn_out_of_enum_range	Assignment of value outside the range of the enumerators
conversion_creating_bad_enum_value	Expression does not correspond to an enumerator in {}

AutosarC++19_03-A7.2.2

Enumeration underlying base type shall be explicitly defined.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unbased_enum	Enumeration underlying base type shall be explicitly defined.

AutosarC++19_03-A7.2.3

Enumerations shall be declared as scoped enum classes.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unscoped_enum	Enumerations shall be declared as scoped enum classes.

AutosarC++19_03-A7.2.4

In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
incomplete_enum_init	Do not initialize enumerators other than the first, or initialize all

AutosarC++19_03-A7.3.1

All overloads of a function shall be visible from where it is called.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_assignment_operator_hiding	If set to true, we allow to hide the assignment operators, as pulling in the parent members using a using-declaration may allow assigning an instance of a parent class to a variable with the type of an inheriting class.	True
enforcement		automated
level		required
report_hidden_fields	Whether to report fields hidden by means of inheritance	False
report_hidden_methods	Whether to report methods hidden by means of inheritance	True
report_other_usages	Whether to find other usages (such as function calls, or taking a fp-reference) between multiple declarations for an identifier.	True
target		implementation

Possible Messages

Name	Message
declarations_surrounding_usage	Declarations straddle a usage
declarations_surrounding_using	Declarations straddle a using-declaration
hiding	{ } hides { }

AutosarC++19_03-A7.4.1

The asm declaration shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_assembler	Use of assembler.

AutosarC++19_03-A7.5.1

A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_const_reference	Whether only to report returns of parameters with reference to const.	True
target		implementation

Possible Messages

Name	Message
returning_reference_to_refparam	Returning reference(pointer to reference parameter.

AutosarC++19_03-A7.5.2

Functions shall not call themselves, either directly or indirectly.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_cycle	Cycle of recursive functions
recursive_function	Function is directly recursive

AutosarC++19_03-A7.6.1

Functions declared with the [[noreturn]] attribute shall not return.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
noreturn_violation	Do not return from a noreturn function.

AutosarC++19_03-A8.2.1

When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_trailing_return	Use trailing return-type syntax for function templates.

AutosarC++19_03-A8.4.1

Functions shall not be defined using the ellipsis notation.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_declarations	Allow declarations of functions with variable number of arguments (e.g. for existing library functions); only report definitions.	True
ignore_inherited	Do not report functions inheriting a variable number of arguments.	False
level		required
target		implementation

Possible Messages

Name	Message
ellipsis_parameter	Function definitions shall not use ellipsis

AutosarC++19_03-A8.4.2

All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
lambda_return_missing_value	Return without value in non-void lambda expression
missing_return	Non-void function needs return with value at end.
missing_return_in_lambda	Non-void lambda expression needs return with value at end.
return_missing_value	Return without value in non-void function

AutosarC++19_03-A8.4.4

Multiple output values from a function should be returned as a struct or tuple.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_modifying_calls	Whether a function may call a function on a parameter that modifies it, without regarding it as a violation here.	True
allow_move_parameters	Whether a move parameter (a rvalue reference) may be written to without triggering a violation. Note: move-constructors and move-assignments are allowed in either case.	False
allow_this_modification	Whether a function may call a member-function on a parameter that modifies this, that is the internal parameter state ('allow_modifying_calls' must be False if this is set to False)	True
enforcement		automated
exclude_constructors	Whether not to report parameters on constructors	True
exclude_operators	Whether not to report parameters on operator functions. Note: if set to True, this also sets 'exclude_shift_operators=True'.	False
exclude_shift_operators	Whether not to report parameters on shift operator functions (operator<<(), operator<<=(), operator>>(), and operator>>=).	False
level		advisory
only_report_multiple_output_parameters	Whether to only report output-parameters, if a function has more than one or also returns a value.	True
record_field_modification_threshold	A percentage of how many record fields may be modified, before the record pointer is considered an output parameter. 0 : every field modification counts as an output parameter 100: the record pointer is only regarded an output parameter if all field are modified	0
target		design

Possible Messages

Name	Message
output_parameter	Use return value instead of output parameter.

AutosarC++19_03-A8.4.5

"consume" parameters declared as X && shall always be moved from.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
exclude_move_assignments_and_constructors	If set to 'True', "consume" parameters in move-assignments and -constructors are not reported, even if they are not moved from.	True
level		required
target		design

Possible Messages

Name	Message
missing_move_consume	"consume" parameters shall always be moved from

AutosarC++19_03-A8.4.6

"forward" parameters declared as T && shall always be forwarded.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		design

Possible Messages

Name	Message
only_forward	"forward" parameters shall always be forwarded.

AutosarC++19_03-A8.4.7

"in" parameters for "cheap to copy" types shall be passed by value.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
exclude_operator_functions	If set to `True`, in parameters with "cheap to copy" types that are not passed by value are not reported for operator functions, as one might not have an option to change the signature.	False
level		required
size_threshold	Size of a type to be considered cheap to copy in number of words.	2
target		design
word_size	Size of a machine word in number of bytes. If None, uses the size of a pointer.	None

Possible Messages

Name	Message
input_parameter	Pass input parameters by value

AutosarC++19_03-A8.4.8

Output parameters shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_modifying_calls	Whether a function may call a function on a parameter that modifies it, without regarding it as a violation here.	True
allow_move_parameters	Whether a move parameter (a rvalue reference) may be written to without triggering a violation. Note: move-constructors and move-assignments are allowed in either case.	False
allow_this_modification	Whether a function may call a member-function on a parameter that modifies this, that is the internal parameter state ('allow_modifying_calls' must be False if this is set to False)	True
enforcement		automated
exclude_constructors	Whether not to report parameters on constructors	False
exclude_operators	Whether not to report parameters on operator functions. Note: if set to True, this also sets 'exclude_shift_operators=True'.	False
exclude_shift_operators	Whether not to report parameters on shift operator functions (operator<<(), operator<<=(), operator>>(), and operator>>=).	False
level		required
only_report_multiple_output_parameters	Whether to only report output-parameters, if a function has more than one or also returns a value.	False
record_field_modification_threshold	A percentage of how many record fields may be modified, before the record pointer is considered an output parameter. 0 : every field modification counts as an output parameter 100: the record pointer is only regarded an output parameter if all field are modified	0
target		design

Possible Messages

Name	Message
output_parameter	Don't use output parameters.

AutosarC++19_03-A8.4.9

"in-out" parameters declared as T & shall be modified.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
exclude_constructors	Whether not to report parameters on constructors	True
level		required
target		design

Possible Messages

Name	Message
complete_replacement	"in-out" completely replaced without being read.
missing_modify	"in-out" parameters not modified, consider making it const.

AutosarC++19_03-A8.5.1

In an initialization list, the order of initialization shall be following: (1) virtual base classes in depth and left to right order of the inheritance graph, (2) direct base classes in left to right order of inheritance list, (3) non-static data members in the order they were declared in the class definition.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	If provided, only messages of these types are reported.	[1719]
reported_severities	List of severities to display.	('error', 'warning', 'remark')
target		implementation
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

AutosarC++19_03-A8.5.2

Braced-initialization {}, without equals sign, shall be used for variable initialization.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
old_style_variable_init	Braced-initialization {}, without equals sign, shall be used for variable initialization.

AutosarC++19_03-A8.5.3

A variable of type auto shall not be initialized using {} or ={} braced-initialization.

Input: IR
Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
braced_auto_variable_init	A variable of type auto shall not be initialized using {} or ={} braced-initialization.

AutosarC++19_03-A9.5.1

Unions shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_tagged_unions	Whether tagged unions (nested in a struct that has an enum discriminator) should be allowed	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
union	Unions shall not be used

AutosarC++19_03-A10.1.1

Class shall not be derived from more than one base class which is not an interface class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
ignore_pure_interfaces	Whether C++ interfaces are allowed, i.e. classes with only pure virtual members.	True
level		required
target		implementation

Possible Messages

Name	Message
multiple_inheritance	Use of multiple inheritance.

AutosarC++19_03-A10.3.1

Virtual function declaration shall contain exactly one of the three specifiers: (1) virtual, (2) override, (3) final.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
disable_for_destructors	If set, destructors are not checked.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
invalid_combination	Use only one of {1} virtual, {2} override, {3} final.

AutosarC++19_03-A10.3.2

Each overriding virtual function shall be declared with the override or final specifier.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_final	If set to True, don't report overriding virtual functions declared with final.	True
enforcement		automated
ignore_destructors	If set to False, also report destructors. Note that not all compilers support this.	False
level		required
target		implementation

Possible Messages

Name	Message
missing_override	Override of functions is only permitted with keyword override/final.

AutosarC++19_03-A10.3.3

Virtual functions shall not be introduced in a final class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
virtual_method	Virtual functions shall not be introduced in a final class.
virtual_method_override	Virtual functions shall not be overridden without final in a final class.

AutosarC++19_03-A10.3.5

A user-defined assignment operator shall not be virtual.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
assignment_virtual	A user-defined assignment operator shall not be virtual.

AutosarC++19_03-A11.0.1

A non-POD type should be defined as class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
require_class_for_pod	Whether POD types should be classes as well	False
target		implementation

Possible Messages

Name	Message
cpp_struct	Use of struct in C++ unit.

AutosarC++19_03-A11.0.2

A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
no_function_member	Struct shall not provide any member functions or methods.
no_struct_as_base	A struct shall not be a base of another struct or class.
no_struct_inheritance	A struct shall not inherit from another struct or class.
non_public_member	Structs shall only have public data members.

AutosarC++19_03-A12.0.1

If a class declares a copy or move operation, or a destructor, either via "`=default`", "`=delete`", or via a user-provided declaration, then all others of these five special member functions shall be declared as well.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_defaulted_destructor_only	Allow classes with a defaulted destructor and no other special member functions.	False
allow_destructor_only	Allow all destructors without copy or move constructors	False
allow_empty_destructor	Allow empty destructors without copy or move constructors.	False
allow_missing_destructor	Suppress messages about missing destructors.	False
enforcement		automated
ignore_pod_classes	Whether POD classes should be checked at all	False
level		required
target		implementation

Possible Messages

Name	Message
missing_constructor_and_asgn	Class with destructor should also declare a copy or move constructor and assignment operator.
missing_copy_asgn	Class with copy constructor is missing copy assignment operator.
missing_copy_constructor	Class with copy assignment operator is missing copy constructor.
missing_destructor	Class with copy or move constructors or assignment operators should also declare a destructor.
missing_move_asgn	Class with move constructor is missing move assignment operator.
missing_move_constructor	Class with move assignment operator is missing move constructor.

AutosarC++19_03-A12.1.1

Constructors shall explicitly initialize all virtual base classes, all direct non-virtual base classes and all non-static data members.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
init_functions	Names of functions to be inspected as well when called directly from constructor.	{'Init', 'init'}
inspect_directly_called_methods	Inspect all methods directly called from constructor.	False
level		required
only_member_initializer_list	Only inspect member initializer list and not the constructor body/methods.	False
report_missing_field_constructors	Enables detection of constructors which rely on implicit field constructor calls.	True
target		implementation

Possible Messages

Name	Message
implicit_field_init	Field is only implicitly initialized in constructor.
missing_base_class_init	Base class is not explicitly initialized in constructor.
missing_field_init	Field is not initialized in constructor.

AutosarC++19_03-A12.1.2

Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_member_initializer_list	Only inspect member initializer list and not the constructor body/methods.	True
target		implementation

Possible Messages

Name	Message
nsdmi_mixed	Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type.

AutosarC++19_03-A12.1.3

If all user-defined constructors of a class initialize data members with constant values that are the same across all constructors, then data members shall be initialized using NSDMI instead.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_constructors_of_types	List of fully qualified type names which constructors are considered literals, without regard to the actual arguments.	[]
allow_literal_like_constructors	Whether to consider constructors that take only literals as arguments a literal. If set to true, this is checked recursively.	False
enforcement		automated
level		required
min_number_common_inits	Minimum number (inclusive) of common initializations to enforce use of NSDMI.	1
target		implementation

Possible Messages

Name	Message
use_nsmdi	Use NSDMI for common constant initializations ({}).

AutosarC++19_03-A12.1.4

All constructors that are callable with a single argument of fundamental type shall be declared explicit.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
consider_only_fundamental_types	Whether this check should be limited to single arguments of fundamental type or should also be applied to user defined types.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
constructor_missing_explicit	Constructor shall be declared explicit

AutosarC++19_03-A12.1.5

Common class initialization for non-constant members shall be done by a delegating constructor.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_copy_move_constructor	Whether to allow direct field initialization without a delegating constructor call for copy and move constructors.	False
enforcement		partially automated
level		required
target		implementation

Possible Messages

Name	Message
delegating_constructor	Use delegating constructor for common initialization.

AutosarC++19_03-A12.1.6

Derived classes that do not need further explicit initialization and require all the constructors from the base class shall use inheriting constructors.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_inheriting	Use inheriting constructors if possible.

AutosarC++19_03-A12.4.1

Destructor of a base class shall be public virtual, public override or protected non-virtual.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
base_class_destructor	Destructor of a base class shall be public virtual, public override or protected non-virtual.

AutosarC++19_03-A12.4.2

If a public destructor of a class is non-virtual, then the class should be declared final.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
non_final	If a public destructor of a class is non-virtual, then the class should be declared final.

AutosarC++19_03-A12.6.1

All class data members that are initialized by the constructor shall be initialized using member initializers.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
init_functions	Names of functions to be inspected as well when called directly from constructor.	('Init', 'init')
inspect_directly_called_methods	Inspect all methods directly called from constructor.	False
level		required
only_member_initializer_list	Only inspect member initializer list and not the constructor body/methods.	True
report_missing_field_constructors	Enables detection of constructors which rely on implicit field constructor calls.	True
target		implementation

Possible Messages

Name	Message
implicit_field_init	Field is only implicitly initialized in constructor.
missing_field_init	Field is not initialized in constructor.

AutosarC++19_03-A12.8.1

A copy constructor shall only initialize its base classes and the non-static members of the class of which it is a member.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_copy_constructor	Whether to report side effects on copy constructors.	True
report_move_constructor	Whether to report side effects on move constructors.	True
target		implementation

Possible Messages

Name	Message
copy_ctor_with_side_effect	Copy Constructor has side-effect
move_ctor_with_side_effect	Move Constructor has side-effect

AutosarC++19_03-A12.8.2

User-defined copy and move assignment operators should use user-defined no-throw swap function.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
asgn_no_swap	A move/copy assignment operator shall use a no-throw swap function.
asgn_not_nothrow	Used swap function is not no-throw.

AutosarC++19_03-A12.8.3

Moved-from object shall not be read-accessed.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		partially automated
inspect_class_field_moves	Detecting moved-from accesses on class fields requires heavy computation. This can be disabled by setting this option to False, but would result in false-positives in the context of class fields.	True
level		required
read_move_function_exceptions	Names of functions that are considered to leave the moved from objects in a well-specified state and are therefore except.	['std::unique_ptr::unique_ptr', 'std::unique_ptr::operator=', 'std::shared_ptr::shared_ptr', 'std::shared_ptr::operator=', 'std::weak_ptr::weak_ptr', 'std::weak_ptr::operator=', 'std::basic_filebuf::basic_filebuf', 'std::basic_filebuf::operator=', 'std::thread::thread', 'std::thread::operator=', 'std::unique_lock::unique_lock', 'std::unique_lock::operator=', 'std::shared_lock::shared_lock', 'std::shared_lock::operator=', 'std::promise::promise', 'std::promise::operator=', 'std::future::future', 'std::future::operator=', 'std::shared_future::shared_future', 'std::shared_future::operator=', 'std::packaged_task::packaged_task', 'std::packaged_task::operator=']
read_move_type_exceptions	Names of types that are considered to be left well-specified state after a move and are therefore except.	['std::basic_ios']
target		implementation

Possible Messages

Name	Message
moved_from_read	Don't read-access a moved-from object

AutosarC++19_03-A12.8.4

Move constructor shall not initialize its class members and base classes using copy semantics.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
move_constructor	Move constructor shall not initialize using copy semantics.

AutosarC++19_03-A12.8.6

Copy and move constructors and copy assignment and move assignment operators shall be declared protected or defined "=delete" in base class.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
special_asgn	Copy and move assignment operators shall be declared protected or defined "=delete" in base class.
special_ctor	Copy and move constructors shall be declared protected or defined "=delete" in base class.

AutosarC++19_03-A12.8.7

Assignment operators should be declared with the ref-qualifier &.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
asgn_missing	Assignment operators should be declared with the ref-qualifier &.

AutosarC++19_03-A13.1.2

User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
user_defined_literal_naming	User-defined literals shall have a suffix matching "_[a-zA-Z]+".

AutosarC++19_03-A13.1.3

User defined literals operators shall only perform conversion of passed parameters.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
literal_side_effects	User-defined literals shall not have side effects.

AutosarC++19_03-A13.2.1

An assignment operator shall return a reference to "this".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
asgn_ref_this	An assignment operator shall return a reference to "this".

AutosarC++19_03-A13.2.2

A binary arithmetic operator and a bitwise operator shall return a "prvalue".

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_bitwise_shift	Whether to allow non-basic values for operator>> or operator<<.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
arith_bitwise_basic_value	Binary arithmetic or bitwise operator shall return a basic value.

AutosarC++19_03-A13.2.3

A relational operator shall return a boolean value.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
relational_bool	A relational operator shall return a boolean value.

AutosarC++19_03-A13.3.1

A function that contains "forwarding reference" as its argument shall not be overloaded.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
overloaded_fref	Functions that contain forwarding reference parameters shall not be overloaded

AutosarC++19_03-A13.5.1

If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
array_access_non_const	No const version of operator[] implemented.

AutosarC++19_03-A13.5.2

All user-defined conversion operators shall be defined explicit.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
user_defined_conversion_explicit	All user-defined conversion operators shall be defined explicit.

AutosarC++19_03-A13.5.3

User-defined conversion operators should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
no_user_defined_conversion	User-defined conversion operators should not be used.

AutosarC++19_03-A13.5.4

If two opposite operators are defined, one shall be defined in terms of the other.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
implement_in_terms	Implement in terms of other operator

AutosarC++19_03-A13.6.1

Digit sequences separators ' shall only be used as follows: (1) for decimal, every 3 digits, (2) for hexadecimal, every 2 digits, (3) for binary, every 4 digits.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
digit_separator	Possibly unreadable digit separators.

AutosarC++19_03-A14.7.2

All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_type_declaration_file	Also allow specifications in the same file of a user-defined type for which the specialization is declared.	True
enforcement		automated
level		required
relax_if_template_only_declared	Allow specializations that are not declared in the header file if the template itself is only declared but not defined in the header.	False
target		implementation

Possible Messages

Name	Message
template_specialization_in_different_file	Specialization not declared in same file as primary template

AutosarC++19_03-A14.8.2

Overloaded function templates shall not be explicitly specialized.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
specialization_of_overloaded_template	Overloaded function templates shall not be explicitly specialized

AutosarC++19_03-A15.1.1

Only instances of types derived from std::exception shall be thrown.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throwing_std_exception	Only instances of types derived from std::exception shall be thrown.

AutosarC++19_03-A15.1.2

An exception object should not have pointer type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throwing_pointer	Exception object of pointer type

AutosarC++19_03-A15.1.3

All thrown exceptions should be unique.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		advisory
target		implementation

Possible Messages

Name	Message
throwing_duplicate	All thrown exceptions should be unique.

AutosarC++19_03-A15.2.1

Constructors that are not noexcept shall not be invoked before program startup.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
static_field_only_noexcept	Constructor called that may throw an exception in static field.
static_variable_only_noexcept	Constructor called that may throw an exception in static variable.

AutosarC++19_03-A15.3.3

Main function and a task main function shall catch at least: base class exceptions from all third-party libraries used, std::exception and all otherwise unhandled exceptions.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		partially-automated
external_base_exceptions	Sequence of external/third-party exception base-classes that should be caught	[]
inspect_thread_main	Whether to also inspect thread main functions.	True
level		required
std_base_exceptions	Sequence of cpp-std exception base-classes that should be caught	['std::exception']
target		implementation

Possible Messages

Name	Message
missing_catch_all	Catch-all required around main program body
missing_catch_handler	Handler for {} needed

AutosarC++19_03-A15.3.4

Catch-all (ellipsis and std::exception) handlers shall be used only in (a) main, (b) task main functions, (c) in functions that are supposed to isolate independent components and (d) when calling third-party code that uses exceptions not according to AUTOSAR C++14 guidelines.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_extern_c_functions	Whether to allow a catch-all in `extern C` functions.	False
allow_in_functions_matching	If not None, a `re.compile()` ed object where functions whose qualified name matches the regex are allowed to contain catch-olds.	None
allow_in_main	Whether to allow a catch-all in the main() function.	True
allow_in_thread_main	Whether to allow a catch-all in thread-main functions.	True
allow_rethrow	Whether to allow a catch-all with a re-throw.	False
autosar_exception_model	The default model for what counts as an checked / unchecked exception.	<bauhaus.ir.autosar.exceptions.autosar_exceptions.AutosarExceptionModel object at 0x7dfa83c4650>
disallow_std_exception	Whether to consider catching the literal std::exception as a catch-all.	True
enforcement		non-automated
level		required
target		implementation

Possible Messages

Name	Message
catch_all	Use of catch(...).
catch_std_exception	Catching std::exceptions is too general.

AutosarC++19_03-A15.3.5

A class type exception shall always be caught by reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_class_types	Whether all types should be caught by reference or only class types.	True
target		implementation

Possible Messages

Name	Message
catch_without_reference	A class type exception shall always be caught by reference.

AutosarC++19_03-A15.4.1

Dynamic exception-specification shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

AutosarC++19_03-A15.4.2

If a function is declared to be noexcept, noexcept(true) or noexcept(<true condition>), then it shall not exit with an exception.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
generateViolationPath	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignoreConstructorDestructor	Whether to ignore escaping exceptions from constructors and destructors.	False
ignoreThrowingFunctions	Whether to ignore noexcept specification violations on function that actually throw an exception.	False
ignoreUnknownRoutines	Whether to ignore extern or only declared routines.	False
level		required
reportNoexceptFalseViolations	Whether to report cases where the function is declared noexcept(false), but no exceptions are ever thrown (directly or indirectly).	False
target		implementation

Possible Messages

Name	Message
implicit_noexcept_specViolationWithout	Function implicitly declared noexcept(false) but no exceptions will be thrown.
noexceptSpecViolationWith	Exception violates function's noexcept-specification.
noexceptSpecViolationWithout	Function declared noexcept(false) but no exceptions will be thrown.

AutosarC++19_03-A15.4.4

A declaration of non-throwing function shall contain noexcept specification.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
generateViolationPath	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignoreConstructorDestructor	Whether to ignore escaping exceptions from constructors and destructors.	False
ignoreThrowingFunctions	Whether to ignore noexcept specification violations on function that actually throw an exception.	True
ignoreUnknownRoutines	Whether to ignore extern or only declared routines.	True
level		required
reportNoexceptFalseViolations	Whether to report cases where the function is declared noexcept(false), but no exceptions are ever thrown (directly or indirectly).	True
target		implementation

Possible Messages

Name	Message
implicit_noexcept_specViolationWithout	Function implicitly declared noexcept(false) but no exceptions will be thrown.
noexceptSpecMissing	Explicit noexcept-specification missing.
noexceptSpecViolationWith	Exception violates function's noexcept-specification.
noexceptSpecViolationWithout	Function declared noexcept(false) but no exceptions will be thrown.

AutosarC++19_03-A15.4.5

Checked exceptions that could be thrown from a function shall be specified together with the function declaration and they shall be identical in all function declarations and for all its overriders.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
autosar_exception_model	The default model for what counts as an checked / unchecked exception.	18_03
enforcement		automated
ignore_definitions_if_declaration_documented	Instead of requiring the documentation to be repeated for every declaration (including definition), with this option set, the rule only checks the non-defining declarations if at least one non-defining declaration exists.	False
level		required
match_qualified_name	Matches the fully-qualified name when comparing documented exceptions with what can actually occur. If set to 'False', this rule will accept any suffix of the qualified name of an exception class as the documentation string.	True
target		implementation
throw_marker	The command to document an exception to be thrown.	@throw

Possible Messages

Name	Message
differing_documented	Documented exceptions differ from overridden method.
document_exception	Document checked exception {} using {}.
superflous_documented	Documented exceptions {} probably never thrown.

AutosarC++19_03-A15.5.1

All user-provided class destructors, deallocation functions, move constructors, move assignment operators and swap functions shall not exit with an exception. A noexcept exception specification shall be added to these functions as appropriate.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
routine_may_except	Function must not exit with an exception.
routine_not_noexcept	Function shall be explicitly declared noexcept if appropriate.

AutosarC++19_03-A15.5.2

Program shall not be abruptly terminated. In particular, an implicit or explicit invocation of std::abort(), std::quick_exit(), std::_Exit(), std::terminate() shall not be done.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
allowed_exceptions	Exceptions that are allowed to escape from destructors.	['bad_alloc', 'bad_cast', 'failure', 'runtime_error', 'system_error']
blacklist	Dictionary of header globbing to [list of] function name globbing(s) of forbidden functions.	dict(...)
constructors	Whether to consider constructors.	False
destructors	Whether to consider destructors.	True
enforcement		partially automated
generateViolationPath	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignoreConstructorDestructor	Whether to ignore escaping exceptions from constructors and destructors.	False
ignoreUnknownRoutines	Whether to ignore extern or only declared routines.	False
inspectAtExitHandlers	Whether to also inspect at_exit() handlers-functions.	True
inspectThreadMain	Whether to also inspect thread main functions.	True
level		required
reportOnlyOneExceptionPerFunction	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
target		implementation

Possible Messages

Name	Message
exceptionEscapingConstructor	Escaping exception from constructor.
exceptionEscapingDestructor	Escaping exception from destructor.
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionEscapingMain	Uncaught exception escaping from main or additional entry point
exceptionSpecificationViolation	Exception violates function's exception-specification.
forbiddenLibfuncCall	Call to forbidden function.

AutosarC++19_03-A15.5.3

The terminate() function shall not be called implicitly.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
allowed_exceptions	Exceptions that are allowed to escape from destructors.	['bad_alloc', 'bad_cast', 'failure', 'runtime_error', 'system_error']
constructors	Whether to consider constructors.	False
destructors	Whether to consider destructors.	True
enforcement		automated
generateViolationPath	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignoreConstructorDestructor	Whether to ignore escaping exceptions from constructors and destructors.	True
ignoreThrowingFunctions	Whether to ignore noexcept specification violations on function that actually throw an exception.	False
ignoreUnknownRoutines	Whether to ignore extern or only declared routines.	False
inspectAtExitHandlers	Whether to also inspect at_exit() handlers-functions.	True
inspectThreadMain	Whether to also inspect thread main functions.	True
level		required
reportNoexceptFalseViolations	Whether to report cases where the function is declared noexcept(false), but no exceptions are ever thrown (directly or indirectly).	False
reportOnlyOneExceptionPerFunction	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
required	Dict which lists required operations per resource. The mapping gives each case a description which maps to a dict for key "Required_Functions", "Resource_Parameter_Empty".	dict(...)
resources	Configuration of resources and operations on them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)
target		implementation

Possible Messages

Name	Message
exceptionEscapingConstructor	Escaping exception from constructor.
exceptionEscapingDestructor	Escaping exception from destructor.
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionEscapingMain	Uncaught exception escaping from main or additional entry point
exceptionSpecificationViolation	Exception violates function's exception-specification.
implicitNoexceptSpecViolationWithout	Function implicitly declared noexcept(false) but no exceptions will be thrown.
noexceptSpecViolationWith	Exception violates function's noexcept-specification.
noexceptSpecViolationWithout	Function declared noexcept(false) but no exceptions will be thrown.
possiblyRequiredOperation	This thread is possibly joinable on destructor call
requiredOperation	This thread is joinable on destructor call

AutosarC++19_03-A16.0.1

The pre-processor shall only be used for file inclusion and include guards.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
accept_conditional_includes	Whether to accept #ifs for conditional includes.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_macro_definition	Macros are only allowed for include guards
line_directive	The pre-processor shall only be used for file inclusion and include guards.
object_macro_definition	Macros are only allowed for include guards
pp_if	Conditional compilation is only allowed for include guards
pragma	The pre-processor shall only be used for file inclusion and include guards.
undef	The pre-processor shall only be used for file inclusion and include guards.

AutosarC++19_03-A16.2.1

The ', ", /*, //, \ characters shall not occur in a header file name or in #include directive.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
check_quotation_mark	Whether system includes should be checked for containing a ".	True
enforcement		automated
forbidden	The substrings to check for. " will be added for system-includes.	set(["//", "\\", "", /*])
level		required
target		implementation

Possible Messages

Name	Message
nonstandard_include_character	Non-standard character in #include directive

AutosarC++19_03-A16.6.1

#error directive shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
error	Use of #error

AutosarC++19_03-A16.7.1

The #pragma directive shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pragma	Use of #pragma

AutosarC++19_03-A17.0.1

Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
additional_reserved_identifiers	Names listed here are seen as violations as well.	set(['assert', 'errno', 'defined'])
allow_reserved_identifier_as_include_guard	If True, macros used for include guards are not checked for being a reserved identifier.	False
check_for_keyword	Whether macro names being keywords should be reported as well.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
macro_having_reserved_name	Definition of reserved identifier or standard library element
undef_of_reserved_name	#undef of reserved identifier or standard library element

AutosarC++19_03-A17.6.1

Non-standard entities shall not be added to standard namespaces.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
std_specialization_blacklist		('std::integral_constant', 'std::is_void', 'std::is_null_pointer', 'std::is_integral', 'std::is_floating_point', 'std::is_array', 'std::is_enum', 'std::is_union', 'std::is_class', 'std::is_function', 'std::is_pointer', 'std::is_lvalue_reference', 'std::is_rvalue_reference', 'std::is_member_object_pointer', 'std::is_member_function_pointer', 'std::is_fundamental', 'std::is_arithmetic', 'std::is_scalar', 'std::is_object', 'std::is_compound', 'std::is_reference', 'std::is_member_pointer', 'std::is_const', 'std::is_volatile', 'std::is_trivial', 'std::is_trivially_copyable', 'std::is_standard_layout', 'std::is_pod', 'std::is_literal_type', 'std::has_unique_object_representations', 'std::is_empty', 'std::is_polymorphic', 'std::is_abstract', 'std::is_final', 'std::is_aggregate', 'std::is_signed', 'std::is_unsigned', 'std::is_constructible', 'std::is_trivially_constructible', 'std::is_nothrow_constructible', 'std::is_default_constructible', 'std::is_trivially_default_constructible', 'std::is_nothrow_default_constructible', 'std::is_copy_constructible', 'std::is_trivially_copy_constructible', 'std::is_nothrow_copy_constructible', 'std::is_move_constructible', 'std::is_trivially_move_constructible', 'std::is_nothrow_move_constructible', 'std::is_assignable', 'std::is_trivially_assignable', 'std::is_nothrowAssignable', 'std::is_copyAssignable', 'std::is_trivially_copyAssignable', 'std::is_nothrow_copyAssignable', 'std::is_moveAssignable', 'std::is_trivially_moveAssignable', 'std::is_nothrow_moveAssignable', 'std::is_destructible', 'std::is_trivially_destructible', 'std::is_nothrow_destructible', 'std::has_virtual_destructor', 'std::is_swappable_with', 'std::is_swappable', 'std::is_nothrow_swappable_with', 'std::is_nothrow_swappable', 'std::alignment_of', 'std::rank', 'std::extent', 'std::is_same', 'std::is_base_of', 'std::is_convertible', 'std::is_nothrow_convertible', 'std::is_invocable', 'std::is_invocable_r', 'std::is_nothrow_invocable', 'std::is_nothrow_invocable_r', 'std::remove_cv', 'std::remove_const', 'std::remove_volatile', 'std::add_cv', 'std::add_const', 'std::add_volatile', 'std::remove_reference', 'std::add_lvalue_reference', 'std::add_rvalue_reference', 'std::remove_pointer', 'std::add_pointer', 'std::make_signed', 'std::make_unsigned', 'std::remove_extent', 'std::remove_all_extents', 'std::aligned_storage', 'std::aligned_union', 'std::decay', 'std::remove_cvref', 'std::enable_if', 'std::conditional', 'std::common_type', 'std::underlying_type', 'std::result_of', 'std::invoke_result', 'std::void_t', 'std::conjunction', 'std::disjunction', 'std::negation', 'std::endian', 'std::unary_function', 'std::binary_function')
std_specialization_whitelist		('std::common_type',)
target		implementation

Possible Messages

Name	Message
std_extension	Invalid addition to std namespace
std_specialization	Invalid std template specialization

AutosarC++19_03-A18.0.1

The C library shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
c_lib_header	Include <{}> instead of <{}>.
cpp_lib_header_with_suffix	Include <{}> instead of <{}>.

AutosarC++19_03-A18.0.2

The error state of a conversion from string to a numeric value shall be checked.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
check_functions	Fully qualified names of functions to check the state of the input stream.	['std::basic_ios::fail', 'std::ios_base::fail']
enforcement		automated
functions_to_check	Fully qualified names of functions after which a call to one of the check functions is required	['std::basic_istream::operator>>']
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	stdlib
symbols	Names of symbols which are forbidden.	['atof', 'atoi', 'atol', 'atoll']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.
missing_basic_ios_fail	Use basic_ios::fail() after reading from input streams.

AutosarC++19_03-A18.0.3

The library <clocale> (locale.h) and the setlocale function shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	locale
symbols	Names of symbols which are forbidden.	['setlocale']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++19_03-A18.1.1

C-style arrays should not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
c_array_used	C-style arrays should not be used.

AutosarC++19_03-A18.1.2

The std::vector<bool> specialization shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
type_used	The type vector<bool> shall not be used.

AutosarC++19_03-A18.1.3

The std::auto_ptr type shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
type_used	The std::auto_ptr shall not be used.

AutosarC++19_03-A18.1.4

A pointer pointing to an element of an array of objects shall not be passed to a smart pointer of single object type.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
type_used	{ } shall not refer to an array type.

AutosarC++19_03-A18.1.6

All std::hash specializations for user-defined types shall have a noexcept function call operator.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
specialization_of_hash_except	std::hash specialization shall have noexcept call operator.

AutosarC++19_03-A18.5.1

Functions malloc, calloc, realloc and free shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
allow_in_user_new_delete_operator	Whether to allow the library functions inside user defined new/delete operator overloads.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
stdlib_memory_function_call	Dynamic memory management function from <stdlib.h> is called.

AutosarC++19_03-A18.5.3

The form of delete operator shall match the form of new operator used to allocate the memory.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. See the base class for more options.	dict(...)
target		implementation

Possible Messages

Name	Message
possible_wrong_release	Resource possibly released using wrong function [allocation used {0}]
wrong_release	Resource released using wrong function [allocation used {0}]

AutosarC++19_03-A18.5.4

If a project has sized or unsized version of operator "delete" globally defined, then both sized and unsized versions shall be defined.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
only_sized	Only the sized version of operator delete is defined.
only_unsized	Only the unsized version of operator delete is defined.

AutosarC++19_03-A18.5.8

Objects that do not outlive a function shall have automatic storage duration.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Configuration

Name	Explanation	Value
allow_smart_ptr_from_function_return	Whether to allow a smart pointer returned from a function as a local variable in a function. Calling the smart pointer constructors or the helper functions std::make_{shared,unique,..} will remain a violation.	True
enforcement		partially automated
level		required
target		implementation

Possible Messages

Name	Message
could_be_scoped	Local objects shall be allocated on the stack.

AutosarC++19_03-A18.9.1

The std::bind shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	functional
symbols	Names of symbols which are forbidden.	['std::bind']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	False

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity std::bind from <>.

AutosarC++19_03-A18.9.2

Forwarding values to other functions shall be done via: (1) std::move if the value is an rvalue reference, (2) std::forward if the value is forwarding reference.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
forwarding_forwarding_reference	Use std::forward if the value is a forwarding reference.
forwarding_rvalue_reference	Use std::move if the value is a rvalue reference.

AutosarC++19_03-A18.9.3

The std::move shall not be used on objects declared const or const&.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
std_move_const	Call to std::move with argument declared const/const&.

AutosarC++19_03-A21.8.1

Arguments to character-handling functions shall be representable as an unsigned char.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
functions	Names of functions that require an unsigned char cast.	['isalnum', 'isalpha', 'islower', 'isupper', 'isdigit', 'isxdigit', 'iscntrl', 'isgraph', 'isspace', 'isblank', 'isprint', 'ispunct', 'tolower', 'toupper']
level		required
target		implementation

Possible Messages

Name	Message
missing_uchar_cast	Missing explicit cast to unsigned char.
missing_uchar_cast_on_routine_literal	Missing explicit cast to unsigned char.

AutosarC++19_03-A23.0.1

An iterator shall not be implicitly converted to const_iterator.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
implicit_const_iterator	An iterator shall not be implicitly converted to const_iterator.

AutosarC++19_03-A26.5.1

Pseudorandom numbers shall not be generated using std::rand().

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
blacklist	Dictionary of header globbing to (list of) function name globbing(s) of forbidden functions.	dict(...)
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
forbidden_libfunc_call	Call to forbidden function.

AutosarC++19_03-A26.5.2

Random number engines shall not be default-initialized.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
additional_blacklist	Sequence of additional forbidden class names where calling the default constructor is forbidden.	[]
blacklist	Sequence of forbidden class names where calling the default constructor is forbidden.	('std::minstd_rand0', 'std::mt19937', 'std::mt19937_64', 'std::ranlux24_base<t>', 'std::ranlux48_base', 'std::ranlux24', 'std::ranlux48', 'std::knuth_b', 'std::default_random_engine', 'std::linear_congruential_engine', 'std::mersenne_twister_engine', 'std::subtract_with_carry_engine')
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
forbidden_default_constructor_call	Call to forbidden default constructor.

AutosarC++19_03-A27.0.4

C-style strings shall not be used.

Input: IR

Source languages: C++

Details

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
c_string_used	C-style strings should not be used.

AutosarC++19_03-M0.1.1

There shall be no unreachable code.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_dead_if_branch_with_const_literal_condition	Whether bodies of if-branches with constant literal conditions should be reported.	True
target		implementation

Possible Messages

Name	Message
unreachable_code	Unreachable code

AutosarC++19_03-M0.1.2

A project shall not contain infeasible paths.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_do_while_false	Whether do ... while(0) should be reported.	True
report_while_true	Whether while(1) ... should be reported.	True
target		implementation

Possible Messages

Name	Message
dead_false_branch	Condition is always true: Else branch is dead
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Parameter condition is always true: Else branch is dead
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Parameter comparison to NULL is always true: Else branch is dead
dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Parameter comparison to NULL is always false: Then branch is dead
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Parameter condition is always false: Then branch is dead
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Condition is always false: Then branch is dead
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Variable condition is always true: Else branch is dead
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Variable condition is always false: Then branch is dead
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead

AutosarC++19_03-M0.1.3

A project shall not contain unused variables.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
ignore_types	Variables of types named here are ignored in this check. Globbing patterns are supported.	[]
level		required
only_check_unit_locals	Whether only global static variables and local variables should be checked.	False
report_global_constants	Whether unused global constants should be reported.	False
report_undefined_variables	Whether only-declared variables should be reported.	True
target		implementation
treat_initialization_as_use	Whether an explicit initialization should be considered a use of the variable.	True
treat_side_effect_constructors_as_use	Whether variables should be seen as used if they are of a class type and initialized through a call to a constructor having a side-effect, e.g. std::lock_guard	False

Possible Messages

Name	Message
unused_field	Unused field
unused_variable	Unused variable

AutosarC++19_03-M0.1.4

A project shall not contain non-volatile POD variables having only one use.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_if_address_taken	Allow using a variable only once if that use involves taking the address of the variable.	False
enforcement		automated
level		required
report_fields	Select whether fields used only once should be reported as well.	True
target		implementation

Possible Messages

Name	Message
field_referenced_only_once	{ } referenced only once
unreferenced_initialized_field	{ } initialized but not referenced
unreferenced_initialized_variable	{ } initialized but not referenced
variable_used_only_once	{ } referenced only once

AutosarC++19_03-M0.1.8

All functions with void return type shall have external side effect(s).

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
assume_effect_when_undefined	Whether all calls to undefined [external] functions should be seen as having a side-effect.	True
enforcement		automated
exceptions	Names of functions that should be excluded from the check.	main
exclude_constructors	If True, tolerate constructors with no side-effect.	False
exclude_virtual_destructors	If True, tolerate virtual destructors with no side-effect.	False
level		required
target		implementation

Possible Messages

Name	Message
void_func_without_side_effect	Void function has no external side-effect

AutosarC++19_03-M0.1.9

There shall be no dead code.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
allow_void_var	Whether {void}var; should be allowed or reported.	True
assume_effect_for_all_calls	Whether all calls should be seen as having a side-effect.	False
assume_effect_when_undefined	Whether all calls to undefined (external) functions should be seen as having a side-effect.	True
enforcement		automated
level		required
report_dead_initializations	Whether initialisations may be reported as dead.	True
report_do_while_false	Whether do ... while[0] should be reported.	True
report_while_true	Whether while[1] ... should be reported.	True
target		implementation
tolerate_void_cast	Whether a {void} cast is accepted.	True

Possible Messages

Name	Message
conditional_unused_def	Result of assignment is not used along some path(s) (disabled)
dead_false_branch	Redundant code, condition is always true
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Redundant code, parameter condition is always true
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Redundant code, parameter comparison to NULL is always true
dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Redundant code, parameter comparison to NULL is always false
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Redundant code, parameter condition is always false
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Redundant code, condition is always false
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Redundant code, variable condition is always true
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Redundant code, variable condition is always false
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead
init_used_in_other_isr	Initialization is only used in some interrupt handler
no_effect	Non-null statement without side-effect
removable_declaration	Declaration can be removed
removable_statement	Statement can be removed
unused_def	Dead (redundant) code: result of assignment is not used
unused_init	Unused initialization
used_in_other_isr	Result of assignment is only used in some interrupt handler

AutosarC++19_03-M0.1.10

Every defined function shall be called at least once.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
ignore_copy_assignment_operators	Whether uncalled copy assignment operators are allowed. Other rules like the "rule of three" might require copy assignment operators as part of good class design, even if there are currently no callers of the copy assignment operator.	False
ignore_copy_constructors	Whether uncalled copy constructors are allowed. Other rules like the "rule of three" might require copy constructors as part of good class design, even if there are currently no callers of the copy constructor.	False
level		required
only_check_internal_or_private_methods	Whether only function with internal linkage or private functions should be checked.	False
only_check_static_or_private_methods	Whether only static functions or private functions should be checked.	False
only_check_unit_locals	Whether only global static functions should be checked.	False
target		implementation

Possible Messages

Name	Message
defined_function_not_called	Defined function is not called

AutosarC++19_03-M0.2.1

An object shall not be assigned to an overlapping object.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
assignment_to_overlapping	An object shall not be assigned to an overlapping object

AutosarC++19_03-M0.3.1

Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
abstract_interpretation_div_by_zero	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
abstract_interpretation_overflow	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
argument_checks	Configuration of [dis]allowed values in addition to what iranalysis.config provides. This is a dict key -> dict with keys 'Functions', 'Parameter', and 'Allowed'/'Disallowed'. The parameter number counting starts at 0. Arguments passed in for this parameters are checked against the value specification given with '[Dis]allowed'.	dict(...)
enforcement		automated
execute_checks_covered_in_other_rules	Whether iranalysis-based checks covered by other Misra-C++:2008 rules should be executed here as well.	False
forbidden	Dict which lists forbidden operations per resource. The mapping gives each case a description which maps to a dict for key "Forbidden_Functions", "Mode_Parameter", "Mode".	dict(...)
level		required
modes	Describes when an allocating call opens a file in read or write mode. This is a dict resource -> dict with keys "Resource_Parameter", "Check_Write_Write", and "Modes" (being a dict with key "Read" and "Write")	dict(...)
perform_global_analysis	Whether global analysis (using iranalysis) should be done, or just a local analysis.	True
report_dead_initializations	Whether initialisations may be reported as dead.	True
report_do_while_false	Whether do ... while[0] should be reported.	True
report_while_true	Whether while[1] ... should be reported.	True
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)
target		implementation

Possible Messages

Name	Message
argumentViolation	Argument not within allowed values
arithmetic_out_of_bounds	Pointer arithmetic on {1} creates pointer outside array bounds of {0}
asgn_not_an_enumerator	Assignment of value which does not correspond to an enumerator
asgn_out_of_enum_range	Assignment of value outside the range of the enumerators
cast_overflow	Cast on result of arithmetic computation may cause overflow
cast_truncate	Cast may truncate value
cast_underflow	Cast on result of arithmetic computation may cause underflow
conditional_unused_def	Result of assignment is not used along some path(s)
deadCatch	Dead exception handler
dead_false_branch	Condition is always true: Else branch is dead
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Parameter condition is always true: Else branch is dead
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Parameter comparison to NULL is always true: Else branch is dead

dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Parameter comparison to NULL is always false: Then branch is dead
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Parameter condition is always false: Then branch is dead
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Condition is always false: Then branch is dead
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Variable condition is always true: Else branch is dead
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Variable condition is always false: Then branch is dead
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead
division_by_zero	Division by zero
double_free	Dynamic memory released here was already released earlier
escaping_address	Escaping address of local variable
forbidden_operation	The way in which this resource was allocated forbids this operation
init_used_in_other_isr	Initialization is only used in some interrupt handler
memory_leak	Call allocates leaking memory
mode_conflict	Same file used for both reading and writing
modulo_by_zero	Modulo by zero
null_deref	Pointer is NULL at dereference
out_of_bounds	Access into array is out of bounds
overflow	Arithmetic computation may cause overflow
pass_as_pointer_to_const_param	Passing uninitialized variable by pointer as function parameter with pointer-to-const type
possible_argumentViolation	Argument possibly not within allowed values
possible_division_by_zero	Possible division by zero
possible_double_free	Dynamic memory released here possibly already released earlier
possible_indirect_out_of_bounds	Pointer-indirect access through {1} might be out of bounds accessing {0}
possible_memory_leak	Call allocates possibly leaking memory
possible_mode_conflict	Same file possibly used for both reading and writing
possible_modulo_by_zero	Possible modulo by zero
possible_null_deref	Pointer may be NULL at dereference
possible_out_of_bounds	Access into array might be out of bounds
possible_stack_free	{ } possibly released by call to { } is a stack object
possible_uninit	Use of possibly uninitialized variable
possible_unrelated_ptr_comparison	Comparing possibly unrelated pointers
possible_unrelated_ptr_subtraction	Subtracting possibly unrelated pointers
possible_use_after_free	Dynamic memory possibly used after it was previously released

possible_wrong_release	Resource possibly released using wrong function (allocation used {0})
possiblyEscapingAddress	Possibly escaping address of local variable (as target of {1})
possiblyForbiddenOperation	The way in which this resource was allocated possibly forbids this operation
stack_free	{ } released by call to { } is a stack object
underflow	Arithmetic computation may cause underflow
uninit	Use of uninitialized variable
unrelated_ptr_comparison	Comparing unrelated pointers
unrelated_ptr_subtraction	Subtracting unrelated pointers
unused_def	Result of assignment is not used
unused_init	Unused initialization
use_after_free	Dynamic memory used after it was previously released
used_in_other_isr	Result of assignment is only used in some interrupt handler
wrong_release	Resource released using wrong function (allocation used {0})

AutosarC++19_03-M0.3.2

If a function generates error information, then that error information shall be tested.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
inspect_template_instances	Whether calls in template instances should be reported.	False
level		required
relevant_functions	If provided, only calls to these functions are inspected.	[]
target		implementation

Possible Messages

Name	Message
discarded_error_with_entity	Integral return value (possibly error code) discarded.
discarded_error_without_entity	Integral return value (possibly error code) discarded.

AutosarC++19_03-M0.4.2

Use of floating-point arithmetic shall be documented.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
float_arithmetic	Use of floating-point arithmetic

AutosarC++19_03-M2.7.1

The character sequence /* shall not be used within a C-style comment.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_cpp_comments	Whether to look for /* in C++-style comments as well.	False
enforcement		automated
files_to_check	Files to apply this check to (Primary_File / User_Include_File / System_Include_File)	{'Primary_File', 'User_Include_File'}
level		required
target		implementation

Possible Messages

Name	Message
nested_c_comment	C-style comment containing /* sequence.

AutosarC++19_03-M2.10.1

Different identifiers shall be typographically unambiguous.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
distinguish_c_name_spaces	Whether to allow similar identifiers in different C name spaces (type tag/label/member/ordinary identifier)	True
distinguish_macro_names	Whether to allow macros to have similar names as other identifiers.	False
distinguish_type_names	Whether to allow types to have similar names as other identifiers. For example, a variable declaration "MyClass myClass;" is allowed by this option. Note: unlike the option 'distinguish_c_name_spaces', this option also distinguishes typedef names, not just struct/class tag names.	False
enforcement		automated
exclude_system_headers	Whether to exclude identifiers appearing in system headers	False
level		required
normalizations	Which pairs of characters should be seen as ambiguous	[('0', 'O'), ('1', 'l'), ('l', '1'), ('i', 'I'), ('5', 'S'), ('2', 'Z'), ('8', 'B'), ('rn', 'm'), ('n', 'h'), ('_', '_')]
target		architecture/design/implementation

Possible Messages

Name	Message
ambiguous	Identifiers are typographically ambiguous
casing	Identifiers only differ in casing

AutosarC++19_03-M2.13.2

Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
octal_escape_sequence	Use of octal escape sequence.
octal_literal	Use of octal literal.

AutosarC++19_03-M2.13.3

A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_only_hex_and_octal	Whether only hex and octal literals should be checked.	True
consider_conversion	Selects whether literals with an implicit conversion from signed to unsigned are reported.	False
consider_pp_expressions	Whether literals in #if conditions are checked.	False
enforcement		automated
level		required
report_all_unsuffixed_hex_and_octal	Whether hex and octal literals should always be reported when they are unsuffixed, independent of the environment	False
target		architecture/design/implementation

Possible Messages

Name	Message
missing_hex_suffix	Hexadecimal and octal constants require "U" suffix
missing_u_suffix	Constant of unsigned type requires "U" suffix

AutosarC++19_03-M2.13.4

Literal suffixes shall be upper case.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		architecture/design/implementation

Possible Messages

Name	Message
lowercase_suffix	Literal suffix should be upper case

AutosarC++19_03-M3.1.2

Functions shall not be declared at block scope.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
local_function_declaration	Functions shall not be declared at block scope.

AutosarC++19_03-M3.2.1

All declarations of an object or function shall have compatible types.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_parameter_type_qualifier_differences	If enabled, do not report a violation for different top-level type qualifiers in parameters: int f(int p); int f(const int p) {} Top-level qualifiers are only relevant for the implementation of the function and have no effect on callers, so it is not necessary to replicate them in the header file.	False
check_parameter_types	Whether parameter types should be compared	False
check_undefined	Whether only-declared routines and variables should also be checked.	True
enforcement		automated
level		required
require_exact_match	Whether to check for identical or compatible types (for routine return types).	False
target		implementation

Possible Messages

Name	Message
decl_return_type_mismatch	Return type at declarations differs
decl_type_mismatch	Type at declarations differs
parameter_type_mismatch	Parameter type at definition differs from type at declaration
return_type_mismatch	Return type at definition differs from type at declaration
type_mismatch	Type at definition differs from type at declaration

AutosarC++19_03-M3.2.2

The One Definition Rule shall not be violated.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
class_struct_difference	{}
different_enumerators	{}
different_field_types	{}
different_fields	{}
general_odrViolation	{}

AutosarC++19_03-M3.2.3

A type, object or function that is used in multiple translation units shall be declared in one and only one file.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_only_one_decl	Enables message if there are multiple declarations but all in same file.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_declared_in_multiple_files	Function declared in more than one file.
function_declared_multiple_times	Function declared more than once.
type_declared_in_multiple_files	Type declared in more than one file.
type_declared_multiple_times	Type declared more than once.
variable_declared_in_multiple_files	Variable declared in more than one file.
variable_declared_multiple_times	Variable declared more than once.

AutosarC++19_03-M3.2.4

An identifier with external linkage shall have exactly one definition.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_undefined_specialized_template	Select whether undefined templates should be reported if specializations of them exist.	True
allow_unselected_declarations	Select whether to report declarations with no definition if they are unused.	True
allowed_undefined_functions	Regular expressions for functions which are tolerated without a definition, e.g. compiler intrinsics.	[_.*]
allowed_undefined_types	Regular expressions for types which are tolerated without a definition.	[_.*]
allowed_undefined_variables	Regular expressions for variables which are tolerated without a definition.	[_.*]
check_composite_types	Check class/struct/union types for having no definition even if they have no external linkage.	False
check_types_only_with_external_linkage	Whether the types should only be checked when they have external linkage	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
composite_defined_multiple_times	Composite type has multiple definitions.
enum_defined_multiple_times	Enumeration type has multiple definitions.
function_defined_multiple_times	Type, object or function has multiple definitions.
undefined_function	Type, object or function without definition
undefined_type	Type without definition
undefined_variable	Type, object or function without definition
variable_defined_multiple_times	Type, object or function has multiple definitions.

AutosarC++19_03-M3.3.2

If a function has internal linkage then all re-declarations shall include the static storage class specifier.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
function_decl_missing_static	Declaration should use static storage class specifier
function_def_missing_static	Definition should use static storage class specifier
variable_decl_missing_static	Declaration should use static storage class specifier
variable_def_missing_static	Definition should use static storage class specifier

AutosarC++19_03-M3.4.1

An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_moving_to_other_primary_file	When a variable is only used in a single file X, but currently implemented in a different file Y, this controls whether the check suggests to move it into X and make it static there.	False
check_classes	Whether to report structs/classes/unions which are only used in a single function or file	True
check_loop_counter	Whether to report for-loop counters being declared before the loop that could be declared in the for-init part instead from C99 on	False
enforcement		automated
exclude_function_locals	If True, variables that could even be function-local are not reported.	True
exclude_undefined	Whether only-declared symbols should be reported as well	True
level		required
move_global_const_into_function	Controls suggestions for global constants that could be declared locally in a function.	True
only_check_unit_locals	Whether only global static variables should be checked.	False
target		implementation

Possible Messages

Name	Message
locality_block	{ } can be declared in a more local scope.
locality_file	{ } can be declared locally in primary file.
locality_function	Global { } can be declared inside function.
locality_loop_init	{ } can be declared in the for-loop's initialization.
var_file_static	{ } can be declared static in primary file.

AutosarC++19_03-M3.9.1

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_missing_qualifiers	If True, tolerate differences in the use of explicit namespace/class qualifiers.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
parameter_type_tokens_mismatch	Type of redeclaration is not token-for-token identical
return_type_tokens_mismatch	Type of redeclaration is not token-for-token identical
variable_type_tokens_mismatch	Type of redeclaration is not token-for-token identical

AutosarC++19_03-M3.9.3

The underlying bit representations of floating-point values shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
float_bit_representation	Use of bit representation of a float value.

AutosarC++19_03-M4.5.1

Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_sizeof	Allows the use of sizeof(bool_var). The Misra rule forbids all uses of expressions of type bool except for those explicitly allowed; and does not allow sizeof. However this is likely an oversight; there is no good reason to prevent the use of sizeof(bool_var). This option also allows alignof(bool_var) in C++11.	False
allow_unevaluated_operator	Allows the use of boolean expressions in contexts where the expression is not evaluated. As of C++17, these are: typeid(bool_var), noexcept(bool_var) and decltype(bool_var). sizeof(bool_var) is controlled by the separate option 'allow_sizeof'.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bool_operand_in_bad_operator	Use of boolean operand in '{}' operator
bool_operand_outside_logical_and_relational_op	Use of boolean operand with integral promotion

AutosarC++19_03-M4.5.3

Expressions with type [plain] char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
char_operand_outside_comparison	Use of character operand in forbidden context

AutosarC++19_03-M4.10.1

NULL shall not be used as an integer value.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
null_as_int	Use of NULL as integer value

AutosarC++19_03-M4.10.2

Literal zero (0) shall not be used as the null-pointer-constant.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
require_nullptr	Require C++11 nullptr instead of macro NULL or literal zero (0).	False
target		implementation

Possible Messages

Name	Message
zero_as_null	Use of literal zero (0) as null-pointer-constant, use {} instead

AutosarC++19_03-M5.0.2

Limited dependence should be placed on C++ operator precedence rules in expressions.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_algebraic_order	Whether parens can be omitted for multiplicative operators as operands in additive operators	False
allow_redundant_parentheses_in_logical_operator	Allow redundant parentheses around unary operators when they appear as operand of '&&' or ' '. This option can be used to avoid a conflict with MisraC++-5.2.1, which requires parentheses in these instances.	True
enforcement		automated
level		required
report_unnecessary_parentheses	Controls whether unnecessary use of parentheses on the right side of assignments or around unary operators are reported.	True
target		implementation

Possible Messages

Name	Message
assignment_rhs_in_parens	No parentheses required for right-hand side of assignment
missing_parens_depends_on_precedence	Parentheses required to avoid dependence on precedence rules
unary_op_in_parens	No parentheses required for unary operator

AutosarC++19_03-M5.0.3

A value expression shall not be implicitly converted to a different underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes], [SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
target		implementation
type_category	Selection of type categories to consider for type changes inside same category.	[SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Result of cvalue expression implicitly converted to different underlying type
cast_changes_type_inside_category	Result of cvalue expression implicitly converted to different underlying type

AutosarC++19_03-M5.0.4

An implicit integral conversion shall not change the signedness of the underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_constant_conversions	If this option is enabled, the rule is relaxed to allow implicit conversions of constant integer expressions whenever the constant value fits into the target type.	True
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
report_signed_to_unsigned	Whether conversion from signed to unsigned shall be reported.	True
report_unsigned_to_signed	Whether conversion from unsigned to signed shall be reported.	True
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_from_signed_to_unsigned	Implicit integral conversion changes signedness of underlying type
cast_from_unsigned_to_signed	Implicit integral conversion changes signedness of underlying type

AutosarC++19_03-M5.0.5

There shall be no implicit floating-integral conversions.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes], [FloatingTypes]], [[FloatingTypes], [SignedTypes, UnsignedTypes]]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Implicit floating-integral conversion

AutosarC++19_03-M5.0.6

An implicit integral or floating-point conversion shall not reduce the size of the underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
report_narrowing	Whether narrowing casts should be reported.	True
report_widening	Whether widening casts should be reported.	False
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
size_source_category	Selection of type categories of the operand being cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
size_target_category	Selection of type categories appearing as target types of the cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
narrowing_cast	Implicit conversion reduces size of underlying type
widening_cast	Conversion to larger type

AutosarC++19_03-M5.0.7

There shall be no explicit floating-integral conversions of a cvalue expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, CharacterTypes], [FloatingTypes]], [[FloatingTypes], [SignedTypes, UnsignedTypes, CharacterTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Explicit floating-integral conversion of cvalue expression

AutosarC++19_03-M5.0.8

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
report_narrowing	Whether narrowing casts should be reported.	False
report_widening	Whether widening casts should be reported.	True
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
size_source_category	Selection of type categories of the operand being cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
size_target_category	Selection of type categories appearing as target types of the cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
narrowing_cast	Conversion to smaller type
widening_cast	Explicit conversion increases size of underlying type of cvalue expression

AutosarC++19_03-M5.0.9

An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
report_signed_to_unsigned	Whether conversion from signed to unsigned shall be reported.	True
report_unsigned_to_signed	Whether conversion from unsigned to signed shall be reported.	True
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_from_signed_to_unsigned	Explicit conversion changes signedness of underlying type of cvalue expression
cast_from_unsigned_to_signed	Explicit conversion changes signedness of underlying type of cvalue expression

AutosarC++19_03-M5.0.10

If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitop_small_without_cast	Bitwise operator requires cast to underlying type on result

AutosarC++19_03-M5.0.11

The plain char type shall only be used for the storage and use of character values.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
char_used_numerically	Plain char datatype used for non-character data.

AutosarC++19_03-M5.0.12

signed char and unsigned char type shall only be used for the storage and use of numeric values.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
numeric_char_used_as_character	Signed/unsigned char datatype used for character data.

AutosarC++19_03-M5.0.14

The first operand of a conditional-operator shall have type bool.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
nonbool_conditional_operator_condition	Condition must have type bool

AutosarC++19_03-M5.0.15

Array indexing shall be the only form of pointer arithmetic.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
indexing_only_on_identifiers	Report array indexing on pointers only for variables (`ptr[i]`), not for other pointer expressions (e.g. `get_ptr()[i]`). This option is meant to suppress the violations introduced by the BAUHAUS-12021 bugfix in version 6.9.6.	False
level		required
target		implementation

Possible Messages

Name	Message
array_indexing_on_pointer	Array indexing only allowed for arrays
pointer_arithmetic	Pointer arithmetic not allowed
pointer_increment_decrement	Pointer arithmetic not allowed

AutosarC++19_03-M5.0.16

A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
arithmetic_on_nonarray	Pointer arithmetic on {1} operates on non-array target {0}
arithmetic_out_of_bounds	Pointer arithmetic on {1} creates pointer outside array bounds of {0}
out_of_bounds	Access into array is out of bounds
possible_indirect_out_of_bounds	Pointer-indirect access through {1} might be out of bounds accessing {0}
possible_out_of_bounds	Access into array might be out of bounds

AutosarC++19_03-M5.0.17

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
possible_unrelated_ptr_subtraction	Subtracting possibly unrelated pointers
unrelated_ptr_subtraction	Subtracting unrelated pointers

AutosarC++19_03-M5.0.18

`>`, `>=`, `<`, `<=` shall not be applied to objects of pointer type, except where they point to the same array.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
possible_unrelated_ptr_comparison	Comparing possibly unrelated pointers
unrelated_ptr_comparison	Comparing unrelated pointers

AutosarC++19_03-M5.0.20

Non-constant operands to a binary bitwise operator shall have the same underlying type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitop_with_different_operand_types	Non-constant operands of bitwise operator differ in underlying type
shortcut_bitop_with_different_operand_types	Non-constant operands of bitwise operator differ in underlying type

AutosarC++19_03-M5.0.21

Bitwise operators shall only be applied to operands of unsigned underlying type.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitop_on_enum	Bitwise operator on enum underlying type
bitop_on_signed	Bitwise operator on signed underlying type

AutosarC++19_03-M5.2.2

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cannot_cast_virtual_base	Cannot convert pointer to base class {} to pointer to derived class {} -- base class is virtual
missing_dynamic_cast_on_virtual_base	Use dynamic_cast on virtual base class

AutosarC++19_03-M5.2.3

Casts from a base class to a derived class should not be performed on polymorphic types.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
cast_from_polybase_to_derived	Cast from polymorphic base class to derived class

AutosarC++19_03-M5.2.6

A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[FunctionPointerTypes], [VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes, NullPointerTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_category	Selection of type categories to consider for type changes inside same category.	[FunctionPointerTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion of function pointer to other type
cast_changes_type_inside_category	Conversion of function pointer to other function pointer type

AutosarC++19_03-M5.2.8

An object with integer type or pointer to void type shall not be converted to an object with pointer type.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, VoidPointerTypes], [ObjectPointerTypes, IncompletePointerTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, [T*](void*)x will not be reported.	True
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion from void* or integer to pointer type

AutosarC++19_03-M5.2.9

A cast should not convert a pointer type to an integral type.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes], [SignedTypes, UnsignedTypes, BoolTypes, EnumTypes, CharacterTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
enforcement		automated
level		required
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
target		implementation
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion between pointer and integral type

AutosarC++19_03-M5.2.10

The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
forbid_all_operators	If True, forbids mixing with any kind of operator; else only with arithmetic operators.	False
level		required
target		implementation

Possible Messages

Name	Message
increment_mixed_with_operator	Increment or decrement mixed with other operators

AutosarC++19_03-M5.2.11

The comma operator, && operator and the || operator shall not be overloaded.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
invalid	Selection of disallowed operator overloads by name.	['operator&&', 'operator ', 'operator,']
invalid_kinds	Selection of disallowed operator overloads by operator kind.	[]
level		required
target		implementation

Possible Messages

Name	Message
forbidden_operator_overload	Overloaded version of comma operator or && or

AutosarC++19_03-M5.2.12

An identifier with array type passed as a function argument shall not decay to a pointer.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
decay	Array to pointer decay

AutosarC++19_03-M5.3.1

Each operand of the ! operator, the logical && or the logical || operators shall have type bool.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
only_in_conditions	If True, only logical operators inside conditions are checked.	False
target		implementation

Possible Messages

Name	Message
nonbool_logical_operator_operand	Operand of logical operator shall be of type bool

AutosarC++19_03-M5.3.2

The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_literals	If True, integer literals are also disallowed as operands.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unary_minus_on_unsigned	Unary minus applied to unsigned

AutosarC++19_03-M5.3.3

The unary & operator shall not be overloaded.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
invalid	Selection of disallowed operator overloads by name.	[]
invalid_kinds	Selection of disallowed operator overloads by operator kind.	[7]
level		required
target		implementation

Possible Messages

Name	Message
forbidden_operator_overload	Overloaded version of unary &

AutosarC++19_03-M5.3.4

Evaluation of the operand to the sizeof operator shall not contain side effects.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
side_effect_in_sizeof	Operand of "sizeof" shall not contain side effects

AutosarC++19_03-M5.8.1

The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_nonconst_cases	Whether a message should be printed for non-constant cases missing a range-checking if statement	False
target		implementation
use_essential_type	Whether essential type should be computed for allowed value range.	False

Possible Messages

Name	Message
excessive_shift	Right operand of shift operator has value {}, but should be in range 0 to {}
possible_excessive_shift	Right operand of shift operator should be checked to be in range 0 to {}

AutosarC++19_03-M5.14.1

The right-hand operand of a logical `&&` or `||` operator shall not contain side effects.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_volatile_access	Whether access to a volatile variable should be reported as side effect.	False
assume_effect_when_undefined	Whether all calls to undefined (external) functions should be seen as having a side-effect.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
call_with_side_effect	Call in right-hand operand of <code>&&</code> or <code> </code> may have side-effect
modifies_local_var	Right-hand operand of <code>&&</code> or <code> </code> modifies ' <code>{}</code> '
side_effect	Right-hand operand of <code>&&</code> or <code> </code> has side-effect

AutosarC++19_03-M5.17.1

The semantic equivalence between a binary operator and its assignment operator form shall be preserved.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_call_relation	If True, checks whether there is a call relation between binary and assignment version.	True
enforcement		automated
ignore_stream_operators	If True, allows definitions of operator <code><<()</code> and operator <code>>>()</code> without the corresponding assignment operator. Note this will also allow operator <code><<=()</code> and operator <code>>>=()</code> as they no longer have an equivalent binary form.	False
level		required
target		implementation

Possible Messages

Name	Message
missing_assignment_version	Missing overload for corresponding assignment version of operator
missing_binary_version	Missing overload for corresponding binary version of operator
missing_call_to_assignment_version	There is no call relation between this operator and its assignment version to ensure semantic equivalence
missing_call_to_binary_version	There is no call relation between this operator and its binary version to ensure semantic equivalence

AutosarC++19_03-M5.18.1

The comma operator shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_comma	Use of comma sequence operator.

AutosarC++19_03-M5.19.1

Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
const_expression_overflow	Overflow during evaluation of constant expression.

AutosarC++19_03-M6.2.1

Assignment operators shall not be used in sub-expressions.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
assignment_result_used	Assignment inside sub-expression.

AutosarC++19_03-M6.2.2

Floating-point expressions shall not be directly or indirectly tested for equality or inequality.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
float_equality_comparison	Testing a float for (in)equality.
float_indirect_equality_comparison	Indirectly testing a float for (in)equality.

AutosarC++19_03-M6.2.3

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_empty_macros	Whether a macro invocation before the ; is allowed if it expands to nothing.	False
allow_nonempty_macros	Whether a non-empty macro invocation before the ; is allowed.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
null_statement_not_isolated	Null statement not on a line by itself

AutosarC++19_03-M6.3.1

The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
loop_body_not_compound	Loop body shall be a statement sequence.
switch_body_not_compound	switch body shall be a statement sequence.

AutosarC++19_03-M6.4.1

An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if

statement.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
dangerous_else	Block or if statement required as else branch.
dangerous_then	Block required as then branch.

AutosarC++19_03-M6.4.2

All if ... else if constructs shall be terminated with an else clause.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
defensive_else	Controls if the else branch of if-cascade's should contain a suitable comment or not be empty.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
non_defensive_else	Last else in the if-cascade must not be empty or contain a comment.
unterminated_if_cascade	Last if in the if-cascade has no else.

AutosarC++19_03-M6.4.3

A switch statement shall be a well-formed switch statement.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
acceptable_end_items	This set characterizes allowed last items at the end of a switch case	('Exit_Switch', 'Throw_Expression')
acceptable_middle_items	This set characterizes allowed last items in the middle of a switch case	('Exit_Switch', 'Nondefault_Case_Label', 'Throw_Expression')
allow_as_first	Whether to allow the "default" case as the first case in a switch.	False
allow_complete_enum	Whether a switch having a case for all enumerators is tolerated without a default.	True
allow_empty_statements	Whether empty statements at the end of switch-case should be ignored.	False
allow_fallthrough_comment	Whether a comment with text "fall" and "through" is allowed to mark fallthrough.	True
allow_fallthrough_macro	Whether a macro called __fallthrough is allowed to mark fallthrough.	True
allow_if	Whether an if containing acceptable items at the end of both branches is accepted.	False
allow_noreturn_function_calls	Whether a function call to a [[noreturn]] function is considered a valid termination.	False
comment	The comment that indicates an intentional fallthrough. Can be a list of comments.	[]
disallow_multiple_compound_statements	Whether there can be more than one toplevel compound statement.	True
disallowed_nested_statements	Statement types not allowed inside compound statements within case-clause body.	None
disallowed_toplevel_statements	Statement types not allowed in case-clause body on toplevel.	('Named_Label', 'Unconditional_Branch', 'Declaration_Statement')
enforcement		automated
level		required
minimum_switch_cases	The number of cases a switch statement should at least have.	1
target		implementation

Possible Messages

Name	Message
case_label_nested	A switch label shall only be used in a switch.
default_not_last	"default" clause is not last in switch.
empty_default	Default clause should contain a statement or comment.
empty_switch	Switch has no "case" clause.
forbidden_compound_statement_in_case	At most one compound statement is allowed on toplevel in case-clauses.
forbidden_nested_statement_in_case	Statement is not allowed inside compound statement in case-clauses.
forbidden_toplevel_statement_in_case	Statement is not allowed outside a compound statement in case-clauses.
minimum_switch_cases	Switch has too little "case" clauses.
missing_default	Switch has no "default" clause.
switch_body_not_compound	A switch statement shall be a well-formed switch statement.
switch_not_well_formed	Switch has a statement before the first case label.
switch_over_bool	Switch expression is effectively Boolean.
unterminated_case	This switch-case is not terminated with {}, last statement is {}.
unterminated_empty_case	This switch-case is empty, not terminated with {}.

AutosarC++19_03-M6.4.4

A switch label shall only be used when the most closely-enclosing compound-statement is the body of a switch-statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
case_label_nested	A switch label shall only be used in a switch.

AutosarC++19_03-M6.4.5

An unconditional throw or break statement shall terminate every non-empty switch-clause.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
acceptable_end_items	This set characterizes allowed last items at the end of a switch case	('Exit_Switch', 'Throw_Expression')
acceptable_middle_items	This set characterizes allowed last items in the middle of a switch case	('Exit_Switch', 'Nondefault_Case_Label', 'Throw_Expression')
allow_empty_statements	Whether empty statements at the end of switch-case should be ignored.	False
allow_fallthrough_comment	Whether a comment with text "fall" and "through" is allowed to mark fallthrough.	True
allow_fallthrough_macro	Whether a macro called __fallthrough is allowed to mark fallthrough.	True
allow_if	Whether an if containing acceptable items at the end of both branches is accepted.	False
allow_noreturn_function_calls	Whether a function call to a [[noreturn]] function is considered a valid termination.	False
comment	The comment that indicates an intentional fallthrough. Can be a list of comments.	[]
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unterminated_case	This switch-case is not terminated with {}, last statement is {}.
unterminated_empty_case	This switch-case is empty, not terminated with {}.

AutosarC++19_03-M6.4.6

The final clause of a switch statement shall be the default clause.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_as_first	Whether to allow the "default" case as the first case in a switch.	False
allow_complete_enum	Whether a switch having a case for all enumerators is tolerated without a default.	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
default_not_last	"default" clause is not last in switch.
empty_default	Default clause should contain a statement or comment.
missing_default	Switch has no "default" clause.

AutosarC++19_03-M6.4.7

The condition of a switch statement shall not have bool type.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
switch_over_bool	Switch condition shall not have bool type.

AutosarC++19_03-M6.5.2

If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
stepping_loop_uses_equality_check	Loop-counter shall not be tested with equality operator if not modified by -- or ++

AutosarC++19_03-M6.5.3

The loop-counter shall not be modified within condition or statement.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
loop_counter_modified_in_condition	Loop-counter shall not be modified within condition
modified_loop_counter	Loop-counter shall not be modified within loop body

AutosarC++19_03-M6.5.4

The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_constexpr	Allow constexpr as a constant <n>.	True
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
nonconst_loop_increment	Loop-counter shall be modified by one of: --, ++, -=n, or +=n (with constant n)

AutosarC++19_03-M6.5.5

A loop-control-variable other than the loop-counter shall not be modified within condition or expression.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
modified_loop_control_variable	Loop-control variable (other than counter) shall not be modified within condition or expression

AutosarC++19_03-M6.5.6

A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
target		implementation

Possible Messages

Name	Message
nonbool_loop_control_variable	Loop-control variable (other than counter) shall have type bool

AutosarC++19_03-M6.6.1

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
goto_label_block	Label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

AutosarC++19_03-M6.6.2

The goto statement shall jump to a label declared later in the same function body.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
backwards_goto	Label referenced by a goto statement shall be declared later in same function.

AutosarC++19_03-M6.6.3

The continue statement shall only be used within a well-formed for loop.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
loop_counter_model		MisraCppLoopCounters
report_outside_for_loops	Whether to report continue statements in do..while/while loops	True
target		implementation

Possible Messages

Name	Message
continue_in_bad_loop	The continue statement shall only be used within a well-formed for loop

AutosarC++19_03-M7.1.2

A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_all_variables	Whether non-parameter variables should be checked as well.	False
enforcement		automated
ignore_forwarding_references	Whether to allow C++11 forwarding references (universal references) even if all template instances would also work with a const lvalue reference.	True
ignore_move_constructors	Whether to allow non-const rvalue references on constructors.	True
level		required
report_overrides	Whether to report violations on overrides. For virtual methods, this rule will report a violation only if the parameter can be marked const in all related methods in the inheritance hierarchy. Thus, the same violation is reported for each method in the inheritance hierarchy. You can set this option to False to report such violations only for the base methods.	True
target		implementation

Possible Messages

Name	Message
parameter_can_point_to_const	{ } can be declared as pointer/reference to const.

AutosarC++19_03-M7.3.1

The global namespace shall only contain main, namespace declarations and extern "C" declarations.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
global_symbol	Symbol not allowed in global namespace.

AutosarC++19_03-M7.3.2

The identifier main shall not be used for a function other than the global function main.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
nonglobal_function_named_main	The identifier main shall not be used for a function other than the global function main

AutosarC++19_03-M7.3.3

There shall be no unnamed namespaces in header files.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unnamed_namespace_in_header	Unnamed namespaces in header file

AutosarC++19_03-M7.3.4

Using-directives shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
using_directive	Using-directives shall not be used

AutosarC++19_03-M7.3.6

using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
using_declaration_in_header	Using-declaration in header file
using_namespace_in_header	Using-directive in header file

AutosarC++19_03-M7.4.1

All usage of assembler shall be documented.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
use_of_assembler	Usage of assembler shall be documented

AutosarC++19_03-M7.4.2

Assembler instructions shall only be introduced using the asm declaration.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pragma_asm	Assembler instructions shall only be introduced using the asm declaration

AutosarC++19_03-M7.4.3

Assembly language shall be encapsulated and isolated.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
asm_not_encapsulated	Assembly language shall be encapsulated and isolated.

AutosarC++19_03-M7.5.1

A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
global_reference_to_local_var	Address of local variable escapes via global variable.
parameter_reference_to_local_var	Address of local variable escapes via parameter.
returning_reference_to_local_var	Returning reference/pointer to local variable.

AutosarC++19_03-M7.5.2

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_longer_living_local	Whether assignment to a longer-living local variable should be accepted.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
possibly_leaking_reference_to_local_variable	Address of local variable is assigned to longer-living object.

AutosarC++19_03-M8.0.1

An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_in_for_loop_init	Whether a multi-declaration is allowed in the for-init statement	False
allow_uninitialized_simple_type	Whether a multi-declaration of simple type is allowed when there is no initialization	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
multi_declaration	Multiple declarators in single declaration

AutosarC++19_03-M8.3.1

Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
redefinition_uses_different_default_argument	Default argument differs from the one in redefined method

AutosarC++19_03-M8.4.2

The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
ignore_casing	Whether to ignore casing for the check.	False
include_redefinitions	Whether declarations of redefining methods should be considered together with the redefined ones.	True
inspect_only_defined_routines	If False, re-declarations of undefined functions are compared to earlier declarations as well.	False
level		required
target		implementation

Possible Messages

Name	Message
parameter_name_mismatch	Different name used for parameter

AutosarC++19_03-M8.4.4

A function identifier shall either be used to call the function or it shall be preceded by &.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value	
allowed	Qualified names of functions of which it is allowed to take the address implicitly, e.g. C++ I/O manipulators	['std::endl', 'std::flush', 'std::boolalpha', 'std::noboolalpha', 'std::showbase', 'std::noshowbase', 'std::showpoint', 'std::noshowpoint', 'std::showpos', 'std::noshowpos', 'std::skipws', 'std::noskipws', 'std::uppercase', 'std::nouppercase', 'std::unitbuf', 'std::nounitbuf', 'std::internal', 'std::left', 'std::right', 'std::dec', 'std::hex', 'std::oct', 'std::fixed', 'std::scientific', 'std::hexfloat', 'std::defaultfloat', 'std::ws', 'std::ends', 'std::resetiosflag', 'std::setiosflag', 'std::setbase', 'std::setfill', 'std::setprecision', 'std::setw', 'std::setw', 'std::get_money', 'std::put_money', 'std::get_time', 'std::put_time', 'std::quoted']	
enforcement		automated	
level		required	
target		implementation	

Possible Messages

Name	Message
implicit_routine_address	Taking address of function without &

AutosarC++19_03-M8.5.2

Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_any_zero_initialization	Whether {0} is allowed in nested aggregates as well.	False
allow_toplevel_zero_initialization	Whether {0} is allowed as top-level initialization for the complete aggregate	True
allowed_classes_without_braces	For the listed class types, don't require braces. This is useful for classes that transparently wrap a single contained value. For example, with this option "std::array<int, 3> a = {1,2,3};" is OK even though there could be two sets of braces (one for "std::array", the other and for its inner "int[3]" array).	['std::array']
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
incomplete_aggregate_initialization	Provide explicit initializer for each part of initialized entity
missing_aggregate_initialization_braces	Use braces to indicate structure of initialized entity

AutosarC++19_03-M9.3.1

const member functions shall not return non-const pointers or references to class-data.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
include_copies_of_pointer_fields	Whether returning a pointer field should be reported as well.	True
inspect_only_const_methods	Whether all methods or only const methods should be checked.	True
level		required
only_report_references	Whether pointer and reference to field should be reported, or just references.	False
smart_pointer_names	Names that are considered smart pointers.	['unique_ptr', 'shared_ptr', 'weak_ptr']
target		implementation

Possible Messages

Name	Message
returning_nonconst_member_reference	Returning non-const reference/pointer to class data.

AutosarC++19_03-M9.3.3

If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_candidates_for_const	If False, avoid reporting methods that can be made const.	True
report_candidates_for_static	If False, avoid reporting methods that can be made static.	True
target		implementation
test_operators_for_static	If True, check whether a method can be made static is also applied to operator methods	False

Possible Messages

Name	Message
method_can_be_const	Method can be declared const.
method_can_be_static	Method can be declared static.

AutosarC++19_03-M9.6.1

When the absolute positioning of bits representing a bit-field is required, then the behaviour and packing of bit-fields shall be documented.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
bitfield	Usage of bit-fields shall be documented

AutosarC++19_03-M9.6.4

Named bit-fields with signed integer type shall have a length of more than one bit.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
signed_single_bitfield	Signed bit field shall be at least 2 bits long.

AutosarC++19_03-M10.1.1

Classes should not be derived from virtual bases.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
virtual_inheritance	Classes should not be derived from virtual bases.

AutosarC++19_03-M10.1.2

A base class shall only be declared virtual if it is used in a diamond hierarchy.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
virtual_inheritance_outside_diamond	A base class shall only be declared virtual if it is used in a diamond hierarchy.

AutosarC++19_03-M10.1.3

An accessible base class shall not be both virtual and non-virtual in the same hierarchy.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
base_class_being_virtual_and_nonvirtual	Has base class which is both virtual and non-virtual.

AutosarC++19_03-M10.2.1

All accessible entity names within a multiple inheritance hierarchy should be unique.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
ambiguous_member	All accessible entity names within a multiple inheritance hierarchy should be unique
use_of_ambiguous_name	{ } is ambiguous

AutosarC++19_03-M10.3.3

A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pure_redefinition	Pure redefinition of non-pure virtual function.

AutosarC++19_03-M11.0.1

Member data in non-POD class types shall be private.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_protected_members	If True, protected members are tolerated.	False
allowed	Specifies allowed fields as pairs (class name pattern, field name pattern). Example: {re.compile('.*'), re.compile('x')} to allow x in all classes.	[]
enforcement		automated
ignore_const_members	If True, non-private const members are tolerated.	False
ignore_pod	Whether fields in POD classes should be reported.	True
ignore_structs	Whether fields in structs should be reported.	False
ignore_templates	Whether fields in generic templates should be reported.	True
level		required
target		implementation

Possible Messages

Name	Message
protected_field	Member data in non-POD class types shall be private.
public_field	Member data in non-POD class types shall be private.

AutosarC++19_03-M12.1.1

An object's dynamic type shall not be used from the body of its constructor or destructor.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
constructor_using_dynamic_cast	Dynamic cast used in constructor/destructor.
constructor_using_typeid	Typeid on polymorphic class used in constructor/destructor.
constructor_using_virtual_call	Virtual call used in constructor/destructor.

AutosarC++19_03-M14.5.3

A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_copy_asgn_for_template	Class has template assignment operator but no copy assignment operator

AutosarC++19_03-M14.6.1

In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unqualified_dependent_member_access	Use qualifiers or this-> to select name that may be found in that dependent base

AutosarC++19_03-M15.0.3

Control shall not be transferred into a try or catch block using a goto or a switch statement.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
goto_into_try	Goto jumps into try or catch block
switch_into_try	Switch statement jumps into try or catch block

AutosarC++19_03-M15.1.1

The assignment-expression of a throw statement shall not itself cause an exception to be thrown.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throw_expression_raises_exception	Expression of throw may itself raise an exception

AutosarC++19_03-M15.1.2

NULL shall not be thrown explicitly.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
throwing_null	NULL shall not be thrown explicitly

AutosarC++19_03-M15.1.3

An empty throw (throw;) shall only be used in the compound-statement of a catch handler.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
rethrow_outside_catch	Rethrow outside any catch block

AutosarC++19_03-M15.3.1

Exceptions shall be raised only after start-up and before termination of the program.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_only_one_exception_per_function	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
report_only_uncaught	Whether the check shall report all throws or just those not caught.	False
target		implementation

Possible Messages

Name	Message
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionRaisedInInitialization	Exception raised in initialization or finalization

AutosarC++19_03-M15.3.3

Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
handlerUsesField	Handler of a function-try-block shall not reference non-static members from this class or its bases

AutosarC++19_03-M15.3.4

Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
enforcement		automated
inspect_at_exit_handlers	Whether to also inspect at_exit() handlers-functions.	True
inspect_thread_main	Whether to also inspect thread main functions.	True
level		required
report_only_one_exception_per_function	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
target		implementation

Possible Messages

Name	Message
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionEscapingMain	Uncaught exception escaping from main or additional entry point

AutosarC++19_03-M15.3.6

Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
wrongCatchOrder	Catch handlers in wrong order.

AutosarC++19_03-M15.3.7

Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
catchAllNotLast	Catch-all shall occur as last handler.

AutosarC++19_03-M16.0.1

#include directives in a file shall only be preceded by other preprocessor directives or comments.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_extern_blocks	Whether to allow includes in `extern` blocks and not report a violation.	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
include_after_code	#include should only be preceded by comments or preprocessor directives

AutosarC++19_03-M16.0.2

Macros shall only be #define'd or #undef'd in the global namespace.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
local_macro	#define or #undef not in global namespace

AutosarC++19_03-M16.0.5

Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
directives	Preprocessing directives to check for in macro arguments.	('#if', '#ifdef', '#ifndef', '#elif', '#else', '#endif', '#pragma', '#warning', '#error', '#line', '#include', '#include_next', '#ident', '#region', '#endregion', '#asm', '#endasm', '#define', '#undef')
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
pp_directive_as_macro_arg	Preprocessing directive used in macro argument.

AutosarC++19_03-M16.0.6

In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
unsafe_macro_param	Macro parameter not enclosed in parentheses

AutosarC++19_03-M16.0.7

Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
undefined_macro_used	Zero used for undefined preprocessing identifier

AutosarC++19_03-M16.0.8

If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
malformed_pp_directive	Malformed or unknown preprocessing directive

AutosarC++19_03-M16.1.1

The defined preprocessor operator shall only be used in one of the two standard forms.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
nonstandard_defined	Non-standard use of defined operator

AutosarC++19_03-M16.1.2

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
missing_endif	The #endif for this directive is missing
missing_if	The #if for this directive is missing

AutosarC++19_03-M16.2.3

Include guards shall be provided.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
macro_name_restrictions	Python iterable of functions with parameters {file, define, macro} to perform additional checks on the macro used as include guard in the given file. Should return None if macro is accepted, else a message to print for the violation. The parameters are: PIR File node for the header, token for the macro #define or #pragma once, and the name of the macro or the string 'pragma once'.	None
target		implementation

Possible Messages

Name	Message
include_guard_missing	Missing include guard.
include_guard_partial	Include guard does not cover complete file.
include_guard_with_wrong_define	Bad include guard: #define uses wrong macro, should be {}.
include_guard_without_define	Incomplete include guard: missing #define {}.

AutosarC++19_03-M16.3.1

There shall be at most one occurrence of the # or ## operators in a single macro definition.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
report_only_concatenated_stringized	If True, only macros where a parameter is stringized and the result used for token concatenation are reported.	False
target		implementation

Possible Messages

Name	Message
concatenated_stringized	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator
more_than_one_macro_hash	Only one # or ## allowed in macro definition

AutosarC++19_03-M16.3.2

The # and ## operators should not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
hash_in_macro	The # and ## operators should not be used.

AutosarC++19_03-M17.0.2

The names of standard library macros and objects shall not be reused.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
check_locals	Whether parameters and local variables should also be checked	True
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
reused_macro_object_libname	The names of standard library macros and objects shall not be reused.

AutosarC++19_03-M17.0.3

The names of standard library functions shall not be overridden.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_function_declarations	Whether nondefining function declarations with library names are allowed	False
enforcement		automated
level		required
target		implementation

Possible Messages

Name	Message
reused_routine_libname	The names of standard library functions shall not be overridden.

AutosarC++19_03-M17.0.5

The setjmp macro and the longjmp function shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	setjmp
symbols	Names of symbols which are forbidden.	['setjmp', 'longjmp']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++19_03-M18.0.3

The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	stdlib
symbols	Names of symbols which are forbidden.	['abort', 'exit', 'getenv', 'system']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++19_03-M18.0.4

The time handling functions of library <ctime> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	time
symbols	Names of symbols which are forbidden.	['clock', 'difftime', 'mktime', 'time', 'asctime', 'ctime', 'gmtime', 'localtime', 'strftime']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h} -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++19_03-M18.0.5

The unbounded functions of library <cstring> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	string
symbols	Names of symbols which are forbidden.	['strcpy', 'strcmp', 'strcat', 'strchr', 'strspn', 'strcspn', 'strpbrk', 'strrchr', 'strstr', 'strtok', 'strlen']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h} -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++19_03-M18.2.1

The macro offsetof shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	stddef
symbols	Names of symbols which are forbidden.	['offsetof']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

AutosarC++19_03-M18.7.1

The signal handling facilities of <csignal> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header (used for symbols like NULL and size_t which are defined in multiple headers)	['NULL', 'size_t']
enforcement		automated
header	Name of the header file which should not be used.	signal
level		required
target		implementation

Possible Messages

Name	Message
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.

AutosarC++19_03-M19.3.1

The error indicator errno shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
enforcement		automated
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level		required
symbol_header	Name of the header file of which the symbols should not be used.	['errno', 'stdlib', 'stddef']
symbols	Names of symbols which are forbidden.	['errno']
target		implementation
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <cerrno>.

AutosarC++19_03-M27.0.1

The stream input/output library <cstdio> shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header (used for symbols like NULL and size_t which are defined in multiple headers)	['NULL', 'size_t']
enforcement		automated
header	Name of the header file which should not be used.	stdio
level		required
target		implementation

Possible Messages

Name	Message
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.

Rules in Group C#

C#-BitfieldHasNoPlurals

An enum used as bitfield requires a plural "s".

Input: IR

Source languages: C#

Details

Enums with "Flags" attribute should have a plural "s" at the end of their names.

Configuration

Name	Explanation	Value

C#-BracesForSingleStatementBodies

Bodies of control statements like if, for and while require curly braces.

Input: IR
Source languages: C#

Details

Bodies of control statements like if, for and while should be enclosed by curly braces.

Configuration

Name	Explanation	Value

C#-CSharpCommentsConvention

Checks for correctly used comment style in C# files.

Input: IR
Source languages: C#

Details

This rule reports the use of delimited comments (starting with /* and ending with */) in C# source code.

Configuration

Name	Explanation	Value
exception	Start of a valid comment, even if it has an invalid prefix.	/**
invalid	Start of an invalid comment.	/*
skip_file_header_comment	Ignore comments at the very beginning of a file.	True

Possible Messages

Name	Message
csharp_comment_style	Use of invalid comment style in C# source file.

C#-CSharpNamingConvention

Naming conventions for C#, inspired partly by <http://msdn.microsoft.com/de-de/library/vstudio/ms229043%28v=vs.100%29.aspx>

Input: IR
Source languages: C#

Details

This rule checks naming conventions for C#, inspired partly by the guidelines specified [here](#).

Configuration

Name	Explanation	Value
naming	Dictionary to map entity -> naming convention.	dict(...)

C#-CSharpNoBackwardGotoJumps

A goto statement should not have a target which is defined before the actual goto statement.

Input: IR
Source languages: C#

Details

This rule reports goto statements that have a target which is defined before the goto statement.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
csharp_backwards_goto	Label referenced by a goto statement shall be declared later in same function.

C#-CSharpNoCatchAllExceptionsClause

Do not use a single clause for catching all exceptions.

Input: IR

Source languages: C#

Details

This rule checks that a `try/catch` block does not have a single catch clause catching `System.Exception`.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
single_catch_all	Single clause for catching all exceptions used.

C#-CSharpNoRefAndOutParameters

Parameters with modifiers `ref` and `out` should not be used.

Input: IR

Source languages: C#

Details

This rule reports parameters passed by reference (i.e., decorated with `ref` or `out`) in public or protected methods.

Configuration

Name	Explanation	Value
exclude_name_prefixes	List of name prefixes of methods that should not be reported.	['Try']

Possible Messages

Name	Message
out_parameter	Parameter definition in publicly accessible methods should not use the "out" modifier.
ref_parameter	Parameter definition in publicly accessible methods should not use the "ref" modifier.

C#-CSharpReferencedLabelInSameBlockAsGoto

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

Input: IR

Source languages: C#

Details

This rule reports labels that are referenced by a goto statement, but not declared in the same block as the goto and also not declared in a block enclosing the `goto` statement.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
csharp_goto_label_block	Label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

C#-EnumeratorRequiresInitialization

Every enumerator has to be initialized.

Input: IR

Source languages: C#

Details

Every enumerator has to be initialized.

Configuration

Name	Explanation	Value

C#-FieldNotPrivateNorProtected

A field should be either private or protected.

Input: IR, RFG

Source languages: C#

Details

Fields should be declared private or protected.

Configuration

Name	Explanation	Value
report_const_fields		False
report_fields_in_struct_layout_types		False
report_fields_in_structs		False
report_READONLY_fields		False

C#-InvalidDependencyPropertyDeclaration

Follow the guidelines for creating dependency properties.

Input: IR, RFG

Source languages: C#

Details

This rule checks whether a dependency property has been properly declared (see also this [guideline](#)).

- A regular CLR property (having the name given in the register invocation) has to be provided.
- The dependency property static field has to be named by suffixing the name of the property with "Property".

Configuration

Name	Explanation	Value
check_name_of_static_field	Check whether the declaring static field has the recommended name.	True
dependency_property_class	The name of the DependencyProperty class including namespaces.	System.Windows.DependencyProperty
register_methods	List of method names that are used for registering dependency properties.	['Register']

C#-MethodShouldBeDeclaredStatic

Instance methods that do not access instance members '' or other instance methods should be declared static.

Input: IR, RFG

Source languages: C#

Details

Instance Methods that do not use the actual object instance they are invoked with should be made static. In this way the independence from the actual object instance is clearly expressed.

Configuration

Name	Explanation	Value

C#-NoHidingOfBaseClassMethods

Do not hide a base method by a method in a derived type.

Input: IR

Source languages: C#

Details

This rule reports the following scenario: A type T derives (directly or indirectly) from type TP , and an instance method m is declared in T with the same name and number of parameters as an instance method in TP . The types of the parameter match except for one or more parameters, where the type of $T.m$ is more common than the type of $TP.m$.

If the option `report_only_more_specific_hidings` (default: True) is set to False, also hidings are reported where *all* parameter types coincide. If the option `report_if_parameters_have_generic_parameter_type` (default: True) is set to true, and the parameter types of at least one pair of corresponding parameters are both generic parameters, the method is reported even if all other parameter types coincide.

Configuration

Name	Explanation	Value
report_if_parameters_have_generic_parameter_type		True
report_only_more_specific_hidings		True

C#-VirtualCallInConstructor

No virtual calls in constructors or destructors.

Input: IR

Source languages: C#

Details

Virtual calls should not occur in constructors or destructors

Configuration

Name	Explanation	Value
exclude_accessors	Do not report calls to virtual event/property accessors.	False

Possible Messages

Name	Message
constructor_using_virtual_call	Virtual call used in constructor/destructor.

Rules in Group CertC

CertC-PRE00

Prefer inline or static functions to function-like macros.

Input: IR

Source languages: C, C++

Details

Macros are dangerous because their use resembles that of real functions, but they have different semantics. The inline function-specifier was introduced to the C programming language in the C99 standard. Inline functions should be preferred over macros when they can be used interchangeably. Making a function an inline function suggests that calls to the function be as fast as possible by using, for example, an alternative to the usual function call mechanism, such as *inline substitution*. [See also [PRE31-C. Avoid side effects in arguments to unsafe macros](#), [PRE01-C. Use parentheses within macros around parameter names](#), and [PRE02-C. Macro replacement lists should be parenthesized](#).]

Inline substitution is not textual substitution, nor does it create a new function. For example, the expansion of a macro used within the body of the function uses the definition it had at the point the function body appeared, not where the function is called; and identifiers refer to the declarations in scope where the body occurs.

Arguably, a decision to inline a function is a low-level optimization detail that the compiler should make without programmer input. The use of inline functions should be evaluated on the basis of (a) how well they are supported by targeted compilers, (b) what (if any) impact they have on the performance characteristics of your system, and (c) portability concerns. Static functions are often as good as inline functions and are supported in C.

Noncompliant Code Example

In this noncompliant code example, the macro `CUBE()` has [undefined behavior](#) when passed an expression that contains side effects:

```
#define CUBE(x) ((x) * (x) * (x))

void func(void) {
    int i = 2;
    int a = 81 / CUBE(++i);
    /* ... */
}
```

For this example, the initialization for `a` expands to

```
int a = 81 / ((++i) * (++i) * (++i));
```

which is undefined (see [EXP30-C. Do not depend on the order of evaluation for side effects](#)).

Compliant Solution

When the macro definition is replaced by an inline function, the [side effect](#) is executed only once before the function is called:

```
inline int cube(int i) {
    return i * i * i;
}

void func(void) {
    int i = 2;
    int a = 81 / cube(++i);
    /* ... */
}
```

Noncompliant Code Example

In this noncompliant code example, the programmer has written a macro called `EXEC_BUMP()` to call a specified function and increment a global counter [[Dewhurst 2002](#)]. When the expansion of a macro is used within the body of a function, as in this example, identifiers refer to the declarations in scope where the body occurs. As a result, when the macro is called in the `aFunc()` function, it inadvertently increments a local counter with the same name as the global variable. Note that this example also violates [DCL01-C. Do not reuse variable names in subscopes](#).

```
size_t count = 0;

#define EXEC_BUMP(func) (func(), ++count)

void g(void) {
    printf("Called g, count = %zu.\n", count);
}

void aFunc(void) {
    size_t count = 0;
    while (count++ < 10) {
        EXEC_BUMP(g);
    }
}
```

The result is that invoking `aFunc()` (incorrectly) prints out the following line five times:

```
Called g, count = 0.
```

Compliant Solution

In this compliant solution, the `EXEC_BUMP()` macro is replaced by the inline function `exec_bump()`. Invoking `aFunc()` now (correctly) prints the value of `count` ranging from 0 to 9:

```
size_t count = 0;

void g(void) {
    printf("Called g, count = %zu.\n", count);
}

typedef void (*exec_func)(void);
inline void exec_bump(exec_func f) {
    f();
    ++count;
}

void aFunc(void) {
    size_t count = 0;
    while (count++ < 10) {
        exec_bump(g);
    }
}
```

The use of the inline function binds the identifier `count` to the global variable when the function body is compiled. The name cannot be re-bound to a different variable (with the same name) when the function is called.

Noncompliant Code Example

Unlike functions, the execution of macros can interleave. Consequently, two macros that are harmless in isolation can cause [undefined behavior](#) when combined in the same expression. In this example, `F()` and `G()` both increment the global variable `operations`, which causes problems when the two macros are used together:

```
int operations = 0, calls_to_F = 0, calls_to_G = 0;

#define F(x) (++operations, ++calls_to_F, 2 * x)
#define G(x) (++operations, ++calls_to_G, x + 1)

void func(int x) {
    int y = F(x) + G(x);
}
```

The variable `operations` is both read and modified twice in the same expression, so it can receive the wrong value if, for example, the following ordering occurs:

```
read operations into register 0
read operations into register 1
increment register 0
increment register 1
store register 0 into operations
store register 1 into operations
```

This noncompliant code example also violates [EXP30-C. Do not depend on the order of evaluation for side effects.](#)

Compliant Solution

The execution of functions, including inline functions, cannot be interleaved, so problematic orderings are not possible:

```
int operations = 0, calls_to_F = 0, calls_to_G = 0;

inline int f(int x) {
    ++operations;
    ++calls_to_F;
    return 2 * x;
}

inline int g(int x) {
    ++operations;
    ++calls_to_G;
    return x + 1;
}

void func(int x) {
    int y = f(x) + g(x);
}
```

Platform-Specific Details

GNU C (and some other compilers) supported inline functions before they were added to the C Standard and, as a result, have significantly different semantics. Richard Kettlewell provides a good explanation of differences between the C99 and GNU C rules [[Kettlewell 2003](#)].

Exceptions

PRE00-C-EX1: Macros can be used to implement *local functions* (repetitive blocks of code that have access to automatic variables from the enclosing scope) that cannot be achieved with inline functions.

PRE00-C-EX2: Macros can be used for concatenating tokens or performing stringification. For example,

```
enum Color { Color_Red, Color_Green, Color_Blue };
static const struct {
    enum Color color;
    const char *name;
} colors[] = {
#define COLOR(color) { Color_##color, #color }
    COLOR(Red), COLOR(Green), COLOR(Blue)
};
```

calculates only one of the two expressions depending on the selector's value. See [PRE05-C. Understand macro replacement when concatenating tokens or performing stringification](#) for more information.

PRE00-C-EX3: Macros can be used to yield a compile-time constant. This is not always possible using inline functions, as shown by the following example:

```
#define ADD_M(a, b) ((a) + (b))
static inline int add_f(int a, int b) {
    return a + b;
}
```

In this example, the `ADD_M(3, 4)` macro invocation yields a constant expression, but the `add_f(3, 4)` function invocation does not.

PRE00-C-EX4: Macros can be used to implement type-generic functions that cannot be implemented in the C language without the aid of a mechanism such as C++ templates.

An example of the use of [function-like macros](#) to create type-generic functions is shown in [MEM02-C. Immediately cast the result of a memory allocation function call into a pointer to the allocated type.](#)

Type-generic macros may also be used, for example, to swap two variables of any type, provided they are of the same type.

PRE00-C-EX5: Macro parameters exhibit call-by-name semantics, whereas functions are call by value. Macros must be used in cases where call-by-name semantics are required.

Risk Assessment

Improper use of macros may result in [undefined behavior](#).

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
PRE00-C	Medium	Unlikely	Medium	P4	L3

Related Guidelines

SEI CERT C++ Coding Standard	PRE00-CPP. Avoid defining macros
ISO/IEC TR 24772:2013	Pre-processor Directives [NMP]
MISRA C:2012	Directive 4.9 (advisory)

Bibliography

[Dewhurst 2002]	Gotcha #26, "#define Pseudofunctions"
[FSF 2005]	Section 5.34, " An Inline Function Is as Fast as a Macro "
[Kettlewell 2003]	
[Summit 2005]	Question 10.4

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/Vlbu>], Copyright [C] 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
function_macro_definition	Prefer function over function-like macro

CertC-PRE01

Use parentheses within macros around parameter names.

Input: IR

Source languages: C, C++

Details

Parenthesize all parameter names in macro definitions. See also [PRE00-C. Prefer inline or static functions to function-like macros](#) and [PRE02-C. Macro replacement lists should be parenthesized](#).

Noncompliant Code Example

This `CUBE()` macro definition is noncompliant because it fails to parenthesize the parameter names:

```
#define CUBE(I) (I * I * I)
```

As a result, the invocation

```
int a = 81 / CUBE(2 + 1);
```

expands to

```
int a = 81 / (2 + 1 * 2 + 1 * 2 + 1); /* Evaluates to 11 */
```

which is clearly not the desired result.

Compliant Solution

Parenthesizing all parameter names in the `CUBE()` macro allows it to expand correctly (when invoked in this manner):

```
#define CUBE(I) ( (I) * (I) * (I) )
int a = 81 / CUBE(2 + 1);
```

Exceptions

PRE01-C-EX1: When the parameter names are surrounded by commas in the replacement text, regardless of how complicated the actual arguments are, there is no need for parenthesizing the macro parameters. Because commas have lower precedence than any other operator, there is no chance of the actual arguments being parsed in a surprising way. Comma separators, which separate arguments in a function call, also have lower precedence than other operators, although they are technically different from comma operators.

```
#define FOO(a, b, c) bar(a, b, c)
/* ... */
FOO(arg1, arg2, arg3);
```

PRE01-C-EX2: Macro parameters cannot be individually parenthesized when concatenating tokens using the `##` operator, converting macro parameters to strings using the `#` operator, or concatenating adjacent string literals. The following `JOIN()` macro concatenates both arguments to form a new token. The `SHOW()` macro converts the single argument into a string literal, which is then passed as a parameter to `printf()` and as a string and as a parameter to the `%d` specifier. For example, if `SHOW()` is invoked as `SHOW(66)`, the macro would be expanded to `printf("66" " = %d\n", 66);`.

```
#define JOIN(a, b) (a ## b)
#define SHOW(a) printf(#a " = %d\n", a)
```

See [PRE05-C. Understand macro replacement when concatenating tokens or performing stringification](#) for more information on using the `##` operator to concatenate tokens.

Risk Assessment

Failing to parenthesize the parameter names in a macro can result in unintended program behavior.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
PRE01-C	Medium	Probable	Low	P12	L1

Related Guidelines

SEI CERT C++ Coding Standard	PRE01-CPP. Use parentheses within macros around parameter names
ISO/IEC TR 24772:2013	Operator Precedence/Order of Evaluation [JCW] Pre-processor Directives [NMP]
MISRA C:2012	Rule 20.7 (required)

Bibliography

[Plum 1985]	
[Summit 2005]	Question 10.1

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/CgU>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
unsafe_macro_param	Macro parameter not enclosed in parentheses

CertC-PRE02

Macro replacement lists should be parenthesized.

Input: IR

Source languages: C, C++

Details

Macro replacement lists should be parenthesized to protect any lower-precedence operators from the surrounding expression. See also [PRE00-C. Prefer inline or static functions to function-like macros](#) and [PRE01-C. Use parentheses within macros around parameter names](#).

Noncompliant Code Example

This `CUBE()` macro definition is noncompliant because it fails to parenthesize the replacement list:

```
#define CUBE(X) (X) * (X) * (X)
int i = 3;
int a = 81 / CUBE(i);
```

As a result, the invocation

```
int a = 81 / CUBE(i);
```

expands to

```
int a = 81 / i * i * i;
```

which evaluates as

```
int a = ((81 / i) * i) * i; /* Evaluates to 243 */
```

which is not the desired behavior.

Compliant Solution

With its replacement list parenthesized, the `CUBE()` macro expands correctly for this type of invocation.

```
#define CUBE(X) ((X) * (X) * (X))
int i = 3;
int a = 81 / CUBE(i);
```

This compliant solution violates [PRE00-C. Prefer inline or static functions to function-like macros](#). Consequently, this solution would be better implemented as an inline function.

Noncompliant Code Example

In this noncompliant code example, `END_OF_FILE` is defined as `-1`. The macro replacement list consists of a unary negation operator followed by an integer literal 1:

```
#define END_OF_FILE -1
/* ... */
if (getchar() EOF) {
/* ... */
```

In this example, the programmer has mistakenly omitted the comparison operator from the conditional statement, which should be `getchar() != END_OF_FILE`. (See [void MSC02-C. Avoid errors of omission](#).) After macro expansion, the conditional expression is incorrectly evaluated as a binary operation: `getchar() -1`. This statement is syntactically correct, even though it is certainly not what the programmer intended. Note that this example also violates [DCL00-C. Const-qualify immutable objects](#).

Parenthesizing the `-1` in the declaration of `END_OF_FILE` ensures that the macro expansion is evaluated correctly:

```
#define END_OF_FILE (-1)
```

Once this modification is made, the noncompliant code example no longer compiles because the macro expansion results in the conditional expression `getchar() (-1)`, which is no longer syntactically valid. Note that there must be a space after `END_OF_FILE` because, otherwise, it becomes a [function-like macro](#) (and one that is incorrectly formed because `-1` cannot be a formal parameter).

Compliant Solution

In this compliant solution, the macro definition is replaced with an enumeration constant in compliance with [DCL00-C. Const-qualify immutable objects](#). In addition, because `EOF` is a reserved macro defined in the `<stdio.h>` header, the compliant solution must also use a different identifier in order to comply with [DCL37-C. Do not declare or define a reserved identifier](#).

```
enum { END_OF_FILE = -1 };
/* ... */
if (getchar() != END_OF_FILE) {
/* ... */
```

Exceptions

PRE02-C-EX1: A macro that expands to a single identifier or function call is not affected by the precedence of any operators in the surrounding expression, so its replacement list need not be parenthesized.

```
#define MY_PID getpid()
```

PRE02-C-EX2: A macro that expands to an array reference using the array-subscript operator [] , or an expression designating a member of a structure or union object using either the member-access . or -> operators is not affected by the precedence of any operators in the surrounding expression, so its replacement list need not be parenthesized.

```
#define NEXT_FREE block->next_free  
#define CID customer_record.account.cid  
#define TOO_FAR array[MAX_ARRAY_SIZE]
```

Risk Assessment

Failing to parenthesize macro replacement lists can cause unexpected results.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
PRE02-C	Medium	Probable	Low	P12	L1

Related Guidelines

SEI CERT C++ Coding Standard	PRE02-CPP. Macro replacement lists should be parenthesized
ISO/IEC TR 24772:2013	Operator Precedence/Order of Evaluation [JCW] Pre-processor Directives [NMP]

Bibliography

[Plum 1985]	Rule 1-1
[Summit 2005]	Question 10.1

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/HAs>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
unsafe_macro_body	Macro replacement text potentially must be parenthesized.

CertC-PRE03

Prefer `typedefs` to `defines` for encoding non-pointer types.

Input: IR

Source languages: C, C++

Details

Prefer type definitions (`typedef`) to macro definitions (`#define`) when encoding types. Type definitions obey scope rules; macro definitions do not. Textual substitution is inferior to using the type system. While type definitions for non-pointer types have similar advantages [[Summit 2005](#)], can make it more difficult to write `const`-correct code (see [DCL05-C. Use `typedefs` of non-pointer types only](#)).

Noncompliant Code Example

This noncompliant code example will not compile, because macros use textual substitution and not the type system:

```
#define MATRIX double matrix[4] [4]
MATRIX matrix_a;
```

After preprocessing, this code example is translated to the following invalid declaration:

```
#define MATRIX double matrix[4] [4]
double matrix[4] [4] matrix_a;
```

Compliant Solution

Using type definitions instead of macro definitions in this compliant solution results in a valid declaration:

```
typedef double matrix[4] [4];
matrix matrix_a;
```

Noncompliant Code Example

I don't actually know what is wrong with this:

```
#define uchar unsigned char
```

Compliant Solution

Use type definitions to encode all non-pointer types.

```
typedef unsigned char uchar;
```

Risk Assessment

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
PRE03-C	Low	Unlikely	Medium	P2	L3

Related Guidelines

SEI CERT C++ Coding Standard	PRE03-CPP. Prefer typedefs to defines for encoding types
ISO/IEC TR 24772:2013	Pre-processor Directives [NMP]

Bibliography

[Saks 1999]	
[Summit 2005]	Question 1.13 Question 11.11

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/mgk>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
nonpointer_type_macro	Macro is used to define non-pointer type, prefer typedef instead

Do not reuse a standard header file name.

Input: IR

Source languages: C, C++

Details

If a file with the same name as a standard header is placed in the search path for included source files, the behavior is [undefined](#).

The following table from the C Standard, subclause 7.1.2 [[ISO/IEC 9899:2011](#)], lists these standard headers:

<assert.h>	<float.h>	<math.h>	<stdatomic.h>	<stdlib.h>	<time.h>
<complex.h>	<inttypes.h>	<setjmp.h>	<stdbool.h>	<stdnoreturn.h>	<uchar.h>
<ctype.h>	<iso646.h>	<signal.h>	<stddef.h>	<string.h>	<wchar.h>
<errno.h>	<limits.h>	<stdalign.h>	<stdint.h>	<tgmath.h>	<wctype.h>
<fenv.h>	<locale.h>	<stdarg.h>	<stdio.h>	<threads.h>	

Do not reuse standard header file names, system-specific header file names, or other header file names.

Noncompliant Code Example

In this noncompliant code example, the programmer chooses to use a local version of the standard library but does not make the change clear:

```
#include "stdio.h" /* Confusing, distinct from <stdio.h> */
/* ... */
```

Compliant Solution

The solution addresses the problem by giving the local library a unique name (per [PRE08-C. Guarantee that header file names are unique](#)), which makes it apparent that the library used is not the original:

```
/* Using a local version of stdio.h */
#include "mystdio.h"
/* ... */
```

Risk Assessment

Using header file names that conflict with other header file names can result in an incorrect file being included.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
PRE04-C	Low	Unlikely	Medium	P2	L3

Related Guidelines

SEI CERT C++ Coding Standard	PRE04-CPP. Do not reuse a standard header file name
CERT Oracle Secure Coding Standard for Java	DCL01-J. Do not reuse public identifiers from the Java Standard Library

Bibliography

[ISO/IEC 9899:2011]	Subclause 7.1.2, "Standard Headers"
-------------------------------------	-------------------------------------

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/7lAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium
standard_names	The forbidden names for user-include files	[<code>'assert.h'</code> , <code>'complex.h'</code> , <code>'ctype.h'</code> , <code>'errno.h'</code> , <code>'fenv.h'</code> , <code>'float.h'</code> , <code>'inttypes.h'</code> , <code>'iso646.h'</code> , <code>'limits.h'</code> , <code>'locale.h'</code> , <code>'math.h'</code> , <code>'setjmp.h'</code> , <code>'signal.h'</code> , <code>'stdalign.h'</code> , <code>'stdarg.h'</code> , <code>'stdatomic.h'</code> , <code>'stdbool.h'</code> , <code>'stddef.h'</code> , <code>'stdint.h'</code> , <code>'stdio.h'</code> , <code>'stdlib.h'</code> , <code>'stdnoreturn.h'</code> , <code>'string.h'</code> , <code>'tgmath.h'</code> , <code>'threads.h'</code> , <code>'time.h'</code> , <code>'uchar.h'</code> , <code>'wchar.h'</code> , <code>'wctype.h'</code>]

Possible Messages

Name	Message
reused_std_name	User include files should not reuse the names of standard headers.

CertC-PRE05

Understand macro replacement when concatenating tokens or performing stringification.

Input: IR

Source languages: C, C++

Details

It is necessary to understand how macro replacement works in C, particularly in the context of concatenating tokens using the `##` operator and converting macro parameters to strings using the `#` operator.

Concatenating Tokens

The `##` preprocessing operator is used to merge two tokens into one while expanding macros, which is called *token pasting* or *token concatenation*. When a macro is expanded, the two tokens on either side of each `##` operator are combined into a single token that replaces the `##` and the two original tokens in the macro expansion [[FSF 2005](#)].

Token pasting is most useful when one or both of the tokens come from a macro argument. If either of the tokens next to a `##` is a parameter name, it is replaced by its actual argument before `##` executes. The actual argument is not macro expanded first.

Stringification

Parameters are not replaced inside string constants, but the `#` preprocessing operator can be used instead. When a macro parameter is used with a leading `#`, the preprocessor replaces it with the literal text of the actual argument converted to a string constant [[FSF 2005](#)].

Noncompliant Code Example

The following definition for `static_assert()` from [DCL03-C. Use a static assertion to test the value of a constant expression](#) uses the `JOIN()` macro to concatenate the token `assertion_failed_at_line_` with the value of `__LINE__`:

```
#define static_assert(e) \
    typedef char JOIN(assertion_failed_at_line_, __LINE__) \
        [(e) ? 1 : -1]
```

`__LINE__` is a predefined macro name that expands to an integer constant representing the presumed line number of the current source line within the current source file. If the intention is to expand the `__LINE__` macro, which is likely the case here, the following definition for `JOIN()` is noncompliant because the `__LINE__` is not expanded, and the character array is subsequently named `assertion_failed_at_line____LINE__`:

```
#define JOIN(x, y) x ## y
```

Compliant Solution

To get the macro to expand, a second level of indirection is required, as shown by this compliant solution:

```
#define JOIN(x, y) JOIN AGAIN(x, y)
#define JOIN AGAIN(x, y) x ## y
```

JOIN(x, y) calls JOIN AGAIN(x, y) so that if x or y is a macro, it is expanded before the ## operator pastes them together.

Note also that macro parameters cannot be individually parenthesized when concatenating tokens using the ## operator, converting macro parameters to strings using the # operator, or concatenating adjacent string literals. This is an exception, PRE01-C-EX2, to [PRE01-C. Use parentheses within macros around parameter names](#).

Noncompliant Code Example

This example is noncompliant if the programmer's intent is to expand the macro before stringification:

```
#define str(s) #
#define foo 4
str(foo)
```

The macro invocation str(foo) expands to foo.

Compliant Solution

To stringify the result of expansion of a macro argument, two levels of macros must be used:

```
#define xstr(s) str(s)
#define str(s) #
#define foo 4
```

The macro invocation xstr(foo) expands to 4 because s is stringified when it is used in str(), so it is not macro expanded first. However, s is an ordinary argument to xstr(), so it is completely macro expanded before xstr() is expanded. Consequently, by the time str() gets to its argument, it has already been macro expanded.

Risk Assessment

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
PRE05-C	Low	Unlikely	Medium	P2	L3

Related Guidelines

SEI CERT C++ Coding Standard	PRE05-CPP. Understand macro replacement when concatenating tokens or performing stringification
--	---

Bibliography

[FSF 2005]	Section 3.4, " Stringification " Section 3.5, " Concatenation "
[Saks 2008]	

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/LQBi>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium
report_only_mixed_uses	Whether to report macros that only use the parameter with # or ##	False

Possible Messages

Name	Message
mixed_macro_arg_substitution	Macro parameter {} is used both normally and with # or ## (where macro expansion is suppressed for the macro passed as argument for it at the positions listed below)
unexpanded_macro_as_argument	Macro parameter {} is used with # or ##, so macro expansion is suppressed for the macro passed as argument for it at the positions listed below

CertC-PRE06

Enclose header files in an inclusion guard.

Input: IR

Source languages: C, C++

Details

Until the early 1980s, large software development projects had a continual problem with the inclusion of headers. One group might have produced a `graphics.h`, for example, which started by including `io.h`. Another group might have produced `keyboard.h`, which also included `io.h`. If `io.h` could not safely be included several times, arguments would break out about which header should include it. Sometimes an agreement was reached that each header should include no other headers, and as a result, some application programs started with dozens of `#include` lines, and sometimes they got the ordering wrong or forgot a required header.

Compliant Solution

All these complications disappeared with the discovery of a simple technique: each header should `#define` a symbol that means "I have already been included." The entire header is then enclosed in an inclusion guard:

```
#ifndef HEADER_H
#define HEADER_H

/* ... Contents of <header.h> ... */

#endif /* HEADER_H */
```

Consequently, the first time `header.h` is `#include`'d, all of its contents are included. If the header file is subsequently `#include`'d again, its contents are bypassed.

Because solutions such as this one make it possible to create a header file that can be included more than once, the C Standard guarantees that the standard headers are safe for multiple inclusion.

Note that it is a common mistake to choose a reserved name for the name of the macro used in the inclusion guard. See [DCL37-C. Do not declare or define a reserved identifier](#).

Risk Assessment

Failure to include header files in an inclusion guard can result in [unexpected behavior](#).

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
PRE06-C	Low	Unlikely	Low	P3	L3

Related Guidelines

SEI CERT C++ Coding Standard	PRE06-CPP. Enclose header files in an inclusion guard
MISRA C:2012	Directive 4.10 (required)

Bibliography

[Plum 1985]	Rule 1-14
-----------------------------	-----------

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/WgBj>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
macro_name_restrictions	Python iterable of functions with parameters {file, define, macro} to perform additional checks on the macro used as include guard in the given file. Should return None if macro is accepted, else a message to print for the violation. The parameters are: PIR File node for the header, token for the macro #define or #pragma once, and the name of the macro or the string 'pragma once'.	()
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
include_guard_missing	Missing include guard.
include_guard_partial	Include guard does not cover complete file.
include_guard_with_wrong_define	Bad include guard: #define uses wrong macro, should be {}.
include_guard_without_define	Incomplete include guard: missing #define {}.

CertC-PRE07

Avoid using repeated question marks.

Input: IR

Source languages: C, C++

Details

Two consecutive question marks signify the start of a trigraph sequence. According to the C Standard, subclause 5.2.1.1 [[ISO/IEC 9899:2011](#)],

All occurrences in a source file of the following sequences of three characters (that is, *trigraph sequences*) are replaced with the corresponding single character.

??=	#		??)]		??!	
??([??*	^		??>	}
??/	\		??<	{		??-	~

Noncompliant Code Example

In this noncompliant code example, `a++` is not executed because the trigraph sequence `??/` is replaced by `\`, logically putting `a++` on the same line as the comment:

```
// What is the value of a now?/
a++;
```

Compliant Solution

This compliant solution eliminates the accidental introduction of the trigraph by separating the question marks:

```
// What is the value of a now? ?
a++;
```

Noncompliant Code Example

This noncompliant code example includes the trigraph sequence `??!`, which is replaced by the character `|`:

```
size_t i = /* Some initial value */;
if (i > 9000) {
    if (puts("Over 9000!??!") == EOF) {
        /* Handle error */
    }
}
```

This example prints `Over 9000!|` if a C-compliant compiler is used.

Compliant Solution

This compliant solution uses string concatenation to concatenate the two question marks; otherwise, they are interpreted as beginning a trigraph sequence:

```
size_t i = /* Some initial value */;
/* Assignment of i */
if (i > 9000) {
    if (puts("Over 9000!??!") == EOF) {
        /* Handle error */
    }
}
```

This code prints Over 9000!??!, as intended.

Risk Assessment

Inadvertent trigraphs can result in unexpected behavior. Some compilers provide options to warn when trigraphs are encountered or to disable trigraph expansion. Use the warning options, and ensure your code compiles cleanly. (See [MSC00-C. Compile cleanly at high warning levels](#).)

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
PRE07-C	Low	Unlikely	Medium	P2	L3

Related Guidelines

SEI CERT C++ Coding Standard	PRE07-CPP. Avoid using repeated question marks
MISRA C:2012	Rule 4.2 (advisory)

Bibliography

[ISO/IEC 9899:2011]	Subclause 5.2.1.1, "Trigraph Sequences"
-------------------------------------	---

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [https://www.securecoding.cert.org/confluence/x/nAE_/], Copyright [C] 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
allow_in_comments	Allow the use of trigraphs in comments	True
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
trigraph_use	Use of trigraph.

CertC-PRE08

Guarantee that header file names are unique.

Input: IR

Source languages: C, C++

Details

Make sure that included header file names are unique. According to the C Standard, subclause 6.10.2, paragraph 5 [[ISO/IEC 9899:2011](#)],

The [implementation](#) shall provide unique mappings for sequences consisting of one or more nondigits or digits (6.4.2.1) followed by a period (.) and a single nondigit. The first character shall not be a digit. The implementation may ignore distinctions of alphabetical case and restrict the mapping to eight significant characters before the period.

This means that

- Only the first eight characters in the file name are guaranteed to be significant.

- The file has only one nondigit character after the period in the file name.
- The case of the characters in the file name is not guaranteed to be significant.

To guarantee that header file names are unique, all included files should differ (in a case-insensitive manner) in their first eight characters or in their (one-character) file extension.

Note that compliance with this recommendation does not require that short file names are used, only that the file names are unique.

Noncompliant Code Example

This noncompliant code example contains references to headers that may exist independently in various environments but can be ambiguously interpreted by a C-compliant compiler:

```
#include "Library.h"
#include <stdio.h>
#include <stdlib.h>
#include "library.h"

#include "utilities_math.h"
#include "utilities_physics.h"

#include "my_library.h"

/* ... */
```

`Library.h` and `library.h` may refer to the same file. Also, because only the first eight characters are guaranteed to be significant, it is unclear whether `utilities_math.h` and `utilities_physics.h` are parsed. Finally, if a file such as `my_libraryOLD.h` exists, it may inadvertently be included instead of `my_library.h`.

Compliant Solution

This compliant solution avoids the ambiguity by renaming the associated files to be unique under the preceding constraints:

```
#include "Lib_main.h"
#include <stdio.h>
#include <stdlib.h>
#include "lib_2.h"

#include "util_math.h"
#include "util_physics.h"

#include "my_library.h"

/* ... */
```

The only solution for mitigating ambiguity of a file, such as `my_libraryOLD.h`, is to rename old files with either a prefix (that would fall within the first eight characters) or add an extension (such as `my_library.h.old`).

Exceptions

PRE08-C-EX1: Although the C Standard requires only the first eight characters in the file name to be significant, most modern systems have long file names, and compilers on such systems can typically differentiate them. Consequently, long file names in headers may be used, provided that all the implementations to which the code is ported can distinguish between these file names.

Risk Assessment

Failing to guarantee uniqueness of header files may result in the inclusion of an older version of a header file, which may include incorrect macro definitions or obsolete function prototypes or result in other errors that may or may not be detected by the compiler. Portability issues may also stem from the use of header names that are not guaranteed to be unique.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
PRE08-C	Low	Unlikely	Medium	P2	L3

Related Guidelines

SEI CERT C++ Coding Standard	PRE08-CPP. Guarantee that header file names are unique
--	--

Bibliography

[ISO/IEC 9899:2011]	Subclause 6.10.2, "Source File Inclusion"
-------------------------------------	---

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/GABB>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium
significant_file_name_characters	If provided, file names must be unique within this many initial characters	None

Possible Messages

Name	Message
duplicate_file_name	Names of included files should be unique (case-insensitive).
shared_file_name_prefix	Names of included files should be unique (case-insensitive) within the first {} characters
wrong_include_casing	#include should use casing of target file (which is {}).

CertC-PRE09

Do not replace secure functions with deprecated or obsolescent functions.

Input: IR

Source languages: C, C++

Details

Macros are frequently used in the remediation of existing code to globally replace one identifier with another, for example, when an existing API changes. Although some risk is always involved, this practice becomes particularly dangerous if a function name is replaced with the function name of a deprecated or obsolescent function. Deprecated functions are defined by the C Standard and Technical Corrigenda. Obsolescent functions are defined by [MSC24-C. Do not use deprecated or obsolescent functions](#).

Although compliance with rule [MSC24-C. Do not use deprecated or obsolescent functions](#) guarantees compliance with this recommendation, the emphasis of this recommendation is the extremely risky and deceptive practice of replacing functions with less secure alternatives.

Noncompliant Code Example

The Internet Systems Consortium's (ISC) Dynamic Host Configuration Protocol (DHCP) contained a vulnerability that introduced several potential buffer overflow conditions [\[VU#654390\]](#). ISC DHCP makes use of the `vsnprintf()` function for writing various log file strings; `vsnprintf()` is defined in the Portable Operating System Interface (POSIX^(R)), Base Specifications, Issue 7 [[IEEE Std 1003.1:2013](#)] as well as in the C Standard. For systems that do not support `vsnprintf()`, a C include file was created that defines the `vsnprintf()` function to `vsprintf()`, as shown in this noncompliant code example:

```
#define vsnprintf(buf, size, fmt, list) \
vsprintf(buf, fmt, list)
```

The `vsprintf()` function does not check bounds. Consequently, `size` is discarded, creating the potential for a buffer overflow when untrusted data is used.

Compliant Solution

The solution is to include an implementation of the missing function `vsnprintf()` to eliminate the dependency on external library functions when they are not available. This compliant solution assumes that `__USE_ISOC11` is not defined on systems that fail to provide a `vsnprintf()` implementation:

```
#include <stdio.h>
#ifndef __USE_ISOC11
/* Reimplements vsnprintf() */
#include "my_stdio.h"
#endif
```

Risk Assessment

Replacing secure functions with less secure functions is a very risky practice because developers can be easily fooled into trusting the function to perform a security check that is absent. This may be a concern, for example, as developers attempt to adopt more secure functions, such as the C11 Annex K functions, that might not be available on all platforms. (See [STR07-C. Use the bounds-checking interfaces for string manipulation](#).)

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
PRE09-C	High	Likely	Medium	P18	L1

Related Guidelines

SEI CERT C++ Coding Standard	PRE09-CPP. Do not replace secure functions with less secure functions
ISO/IEC TR 24772:2013	Executing or Loading Untrusted Code [XYS]
MITRE CWE	CWE-684 , Failure to provide specified functionality

Bibliography

[IEEE Std 1003.1:2013]	XSH, System Interfaces, vsnprintf, vsprintf
[Seacord 2013]	Chapter 6, "Formatted Output"
[VU#654390]	

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/iwD3>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
forbidden	Names of functions that should not be renamed with a macro.	set(['vsnprintf', 'vsscanf', 'sscanf', 'snprintf', 'wmemmove', 'wprintf', 'wctomb', 'strcat', 'strerror', 'fwprintf', 'swprintf', 'fprintf', 'vswscanf', 'printf', 'wcrtomb', 'fopen', 'strncpy', 'atoll', 'strncat', 'wmemcpy', 'fwscanf', 'qsort', 'vfwprintf', 'wcsncat', 'sprintf', 'vscanf', 'gets', 'mbstowcs', 'localtime', 'vprintf', 'vwprintf', 'wcsncpy', 'rewind', 'gmtime', 'getenv', 'wcscat', 'mbsrtowcs', 'vswprintf', 'vfprintf', 'atol', 'wscanf', 'setbuf', 'atof', 'memcpy', 'vfwscanf', 'wcsrtombs', 'ctime', 'fscanf', 'vfscanf', 'wcstombs', 'memmove', 'asctime', 'swscanf', 'wcscpy', 'wcstok', 'bsearch', 'freopen', 'strcpy', 'strtok', 'atoi', 'vwscanf', 'vsprintf'])
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
forbidden_function_renaming	Forbidden renaming/wrapping with macro.

CertC-PRE10

Wrap multistatement macros in a do-while loop.

Input: IR

Source languages: C, C++

Details

Macros are often used to execute a sequence of multiple statements as a group.

Inline functions are, in general, more suitable for this task (see [PRE00-C. Prefer inline or static functions to function-like macros](#)). Occasionally, however,

they are not feasible (when macros are expected to operate on variables of different types, for example).

When multiple statements are used in a macro, they should be bound together in a `do-while` loop syntactically, so the macro can appear safely inside `if` clauses or other places that expect a single statement or a statement block. (Alternatively, when an `if`, `for`, or `while` statement uses braces even for a single body statement, then multiple statements in a macro will expand correctly even without a `do-while` loop (see [EXP19-C. Use braces for the body of an if, for, or while statement](#)).

Noncompliant Code Example

This noncompliant code example contains multiple, unbound statements:

```
/*
 * Swaps two values and requires
 * tmp variable to be defined.
 */
#define SWAP(x, y) \
    tmp = x; \
    x = y; \
    y = tmp
```

This macro expands correctly in a normal sequence of statements but not as the `then` clause in an `if` statement:

```
int x, y, z, tmp;
if (z == 0)
    SWAP(x, y);
```

It expands to the following, which is certainly not what the programmer intended:

```
int x, y, z, tmp;
if (z == 0)
    tmp = x;
x = y;
y = tmp;
```

Noncompliant Code Example

This noncompliant code example inadequately bounds multiple statements:

```
/*
 * Swaps two values and requires
 * tmp variable to be defined.
 */
#define SWAP(x, y) { tmp = x; x = y; y = tmp; }
```

This macro fails to expand correctly in some case, such as the following example, which is meant to be an `if` statement with two branches:

```
if (x > y)
    SWAP(x, y); /* Branch 1 */
else
    do_something(); /* Branch 2 */
```

Following macro expansion, however, this code is interpreted as an `if` statement with only one branch:

```
if (x > y) { /* Single-branch if-statement!!! */

    tmp = x; /* The one and only branch consists */
    x = y; /* of the block. */
    y = tmp;
}
; /* Empty statement */
else /* ERROR!!! "parse error before else" */
    do_something();
```

The problem is the semicolon (`;`) following the block.

Compliant Solution

Wrapping the macro inside a `do-while` loop mitigates the problem:

```
/*
 * Swaps two values and requires
 * tmp variable to be defined.
 */
#define SWAP(x, y) \
    do { \
        tmp = x; \
        x = y; \
        y = tmp; } \
    while (0)
```

The `do-while` loop will always be executed exactly once.

Risk Assessment

Improperly wrapped statement macros can result in unexpected and difficult to diagnose behavior.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
PRE10-C	Medium	Probable	Low	P12	L1

Related Guidelines

ISO/IEC TR 24772:2013	Pre-processor Directives [NMP]
---------------------------------------	--------------------------------

Bibliography

Linux Kernel Newbies FAQ	FAQ/DoWhile0
--	------------------------------

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/jgL7>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
unwrapped_multi_statement_macro	Multi-statement macro should be wrapped in a do .. while {0} loop

CertC-PRE11

Do not conclude macro definitions with a semicolon.

Input: IR

Source languages: C, C++

Details

Macros are frequently used to make source code more readable. Macro definitions, regardless of whether they expand to a single or multiple statements, should not conclude with a semicolon. (See [PRE10-C. Wrap multistatement macros in a do-while loop](#).) If required, the semicolon should be included following the macro expansion. Inadvertently inserting a semicolon at the end of the macro definition can unexpectedly change the control flow of the program.

Another way to avoid this problem is to prefer inline or static functions over function-like macros. (See also [PRE00-C. Prefer inline or static functions to function-like macros](#).)

In general, the programmer should ensure that there is no semicolon at the end of a macro definition. The responsibility for having a semicolon where needed during the use of such a macro should be delegated to the person invoking the macro.

Noncompliant Code Example

This noncompliant code example creates a macro definition for a `for` loop in the program. A `for` loop should require braces, even if it contains only a single body statement. (See [EXP19-C. Use braces for the body of an if, for, or while statement](#).) This macro takes an integer argument, which is the number of times the loop should run. The programmer has inserted a semicolon at the end of the macro definition by mistake.

```
#define FOR_LOOP(n)  for(i=0; i<(n); i++)
int i;
FOR_LOOP(3)
{
    puts("Inside for loop\n");
}
```

The programmer expects to get the following output from the code:

```
Inside for loop
Inside for loop
Inside for loop
```

But because of the semicolon at the end of the macro definition, the `for` loop in the program has a null statement, so the statement "Inside for loop" gets printed just once. Essentially, the semicolon at the end of the macro definition changes the program control flow.

Although this example might not actually be used in code, it shows the effect a semicolon in a macro definition can have.

Compliant Solution

The compliant solution is to write the macro definition without the semicolon at the end, leaving the decision whether or not to have a semicolon up to the person who is using the macro:

```
#define FOR_LOOP(n)  for(i=0; i<(n); i++)
int i;
FOR_LOOP(3)
{
    puts("Inside for loop\n");
}
```

Noncompliant Code Example

In this noncompliant code example, the programmer defines a macro that increments the value of the first argument, `x`, by 1 and modulates it with the value of the second argument, `max`:

```
#define INCREMOD(x, max) ((x) = ((x) + 1) % (max));
int index = 0;
int value;
value = INCREMOD(index, 10) + 2;
/* ... */
```

In this case, the programmer intends to increment `index` and then use that as a value by adding 2 to it. Unfortunately, the value is equal to the incremented value of `index` because of the semicolon present at the end of the macro. The `+ 2;` is treated as a separate statement by the compiler. The user will not get any compilation errors. If the user has not enabled warnings while compiling, the effect of the semicolon in the macro cannot be detected at an early stage.

Compliant Solution

The compliant solution is to write the macro definition without the semicolon at the end, leaving the decision whether or not to have a semicolon up to the person who is using the macro:

```
#define INCREMOD(x, max) ((x) = ((x) + 1) % (max))
```

Compliant Solution

This compliant solution uses an inline function as recommended by [PRE00-C. Prefer inline or static functions to function-like macros](#).

```
inline int incremod(int *x, int max) {*x = (*x + 1) % max;}
```

Risk Assessment

Using a semicolon at the end of a macro definition can result in the change of program control flow and thus unintended program behavior.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
PRE11-C	Medium	Probable	Low	P12	L1

Related Guidelines

SEI CERT C++ Coding Standard	PRE11-CPP. Do not conclude macro definitions with a semicolon
--	---

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/wgBIAQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
macro_ends_with_semicolon	Macro ends with a semicolon

CertC-PRE12

Do not define unsafe macros.

Input: IR

Source languages: C, C++

Details

An [unsafe function-like macro](#) is one that, when expanded, evaluates its argument more than once or does not evaluate it at all. Contrasted with function calls, which always evaluate each of their arguments exactly once, unsafe function-like macros often have unexpected and surprising effects and lead to subtle, hard-to-find defects (see [PRE31-C. Avoid side effects in arguments to unsafe macros](#)). Consequently, every [function-like macro](#) should evaluate each of its arguments exactly once. Alternatively and preferably, defining function-like macros should be avoided in favor of inline functions (see [PRE00-C. Prefer inline or static functions to function-like macros](#)).

Noncompliant Code Example (Multiple Argument Evaluation)

The most severe problem with [unsafe function-like macros](#) is side effects of macro arguments, as shown in this noncompliant code example:

```
#define ABS(x) (((x) < 0) ? -(x) : (x))

void f(int n) {
    int m;
    m = ABS (++n);
    /* ... */
}
```

The invocation of the `ABS()` macro in this noncompliant code example expands to the following code. The resulting code has well-defined behavior but causes `n` to be incremented twice rather than once, which may be surprising to those unfamiliar with the [implementation](#) of the macro or unaware that they are using a macro in the first place.

```
m = (((++n) < 0) ? -(++n) : (++n));
```

Compliant Solution (Inline Function)

A possible and preferable compliant solution is to define an inline function with equivalent but unsurprising semantics:

```
inline int Abs(int x) {
    return x < 0 ? -x : x;
}
```

Compliant Solution (Language Extension)

Some implementations provide language extensions that make it possible to define safe function-like macros, such as the macro `ABS()`, that would otherwise require evaluating their arguments more than once. For example, the GCC extension [Statements and Declarations in Expressions](#) makes it possible to implement the macro `ABS()` in a safe way. Note, however, that because relying on implementation-defined extensions introduces undesirable platform dependencies that may make the resulting code nonportable, such solutions should be avoided in favor of portable ones wherever possible (see [MSC14-C. Do not introduce unnecessary platform dependencies](#)).

Another GCC extension known as *statement expression* makes it possible for the block statement to appear where an expression is expected. The statement expression extension establishes a scope (note the curly braces) and any declarations in it are distinct from those in enclosing scopes.

```
#define ABS(x) __extension__ ({ __typeof__(x) __tmp = x; __tmp < 0 ? - __tmp : __tmp; })
```

Risk Assessment

Defining an unsafe macro leads to invocations of the macro with an argument that has [side effects](#), causing those side effects to occur more than once. [Unexpected](#) or [undefined](#) program behavior can result.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
PRE12-C	Low	Probable	Low	P6	L2

Related Guidelines

SEI CERT C++ Coding Standard	PRE10-CPP. Do not define unsafe macros
--	--

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/Tf3Ag>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
parameter_multiply_evaluated	Macro parameter is evaluated more than once
parameter_not_evaluated	Macro parameter is not evaluated

CertC-PRE13

Use the Standard predefined macros to test for versions and features.

Input: IR

Source languages: C, C++

Details

The C Standard defines a set of predefined macros (see subclause 6.10.8) to help the user determine if the [implementation](#) being used is a [conforming](#) implementation, and if so, to which version of the C Standard it conforms. These macros can also help the user to determine which of the standard features are implemented.

The following tables list these macros and indicate in which version of the C Standard they were introduced. The following macros are required:

Macro Name	C90	C99	C11
__STDC__	x	x	x
__STDC_HOSTED__		x	x
__STDC_VERSION__ ¹		x	x
__DATE__	x	x	x
__FILE__	x	x	x
__LINE__	x	x	x
__TIME__	x	x	x

1) __STDC_VERSION__ was introduced by an Amendment to C90, this version of the C Standard is commonly call C94

The following are optional environment macros:

Macro Name	C90	C99	C11
__STDC_ISO_10646__		x	x
__STDC_MB_MIGHT_NEQ_WC__		x	x
__STDC_UTF_16__			x
__STDC_UTF_32__			x

The following are optional feature macros:

Macro Name	C90	C99	C11
__STDC_ANALYZABLE__			x
__STDC_IEC_559__		x	x
__STDC_IEC_559_COMPLEX__		x	x
__STDC_LIB_EXT1__			x
__STDC_NO_ATOMICS__			x
__STDC_NO_COMPLEX__			x
__STDC_NO_THREADS__			x
__STDC_NO_VLA__			x

The following is optional and is defined by the user:

Macro Name	C90	C99	C11
__STDC_WANT_LIB_EXT1__			x

Noncompliant Code Example (Checking Value of Predefined Macro)

C Standard predefined macros should never be tested for a value before the macro is tested for definition, as shown in this noncompliant code example:

```
#include <stdio.h>

int main(void) {
    #if (__STDC__ == 1)
        printf("Implementation is ISO-conforming.\n");
    #else
        printf("Implementation is not ISO-conforming.\n");
    #endif
    /* ... */

    return 0;
}
```

Compliant Solution (Testing for Definition of Macro)

In this compliant solution, the definition of the predefined macro __STDC__ is tested before the value of the macro is tested:

```
#include <stdio.h>

int main(void) {
    #if defined(__STDC__)
        #if (__STDC__ == 1)
            printf("Implementation is ISO-conforming.\n");
        #else
            printf("Implementation is not ISO-conforming.\n");
        #endif
    #else /* !defined(__STDC__) */
        printf("__STDC__ is not defined.\n");
    #endif
    /* ... */

    return 0;
}
```

Compliant Solution (Test for Optional Feature)

This compliant solution tests to see if the C11 predefined macro __STDC_ANALYZABLE__ is defined and what value the implementation has given the macro:

```
#include <stdio.h>

int main(void) {
    #if defined(__STDC__)
        #if defined(__STDC_VERSION__) && (__STDC_VERSION__ >= 201112L) /* C11 */
            #if defined(__STDC_ANALYZABLE__)
                #if (__STDC_ANALYZABLE__ == 1)
                    printf("Compiler conforms to Annex L (Analyzability).\n");
                #else
                    printf("Compiler does not support Annex L (Analyzability).\n");
                #endif
            #else
                printf("__STDC_ANALYZABLE__ is not defined.\n");
            #endif
        #else
            printf("Compiler not C11.\n");
        #endif
    #else
        printf("Compiler not Standard C.\n");
    #endif

    return 0;
}
```

Compliant Solution [Optional Language Features]

This compliant solution checks for the C11 optional language features in Annex K. If Annex K is supported by the implementation, the functions defined in Annex K are used; if Annex K is not supported, then the standard library functions are used. (See [DCL09-C. Declare functions that return errno with a return type of errno_t](#).)

```
#if defined(__STDC_LIB_EXT1__)
#ifndef __STDC_LIB_EXT1__ >= 201112L
#define USE_EXT1 1
#define __STDC_WANT_LIB_EXT1__ 1 /* Want the ext1 functions */
#endif
#endif

#include <string.h>
#include <stdlib.h>

int main(void) {
    char source_msg[] = "This is a test.";
    char *msg = malloc(sizeof(source_msg) + 1);

    if (msg != NULL) {
        #if defined(USE_EXT1)
        strcpy_s(msg, sizeof msg, source_msg);
        #else
        strcpy(msg, source_msg);
        #endif
    }
    else {
        return EXIT_FAILURE;
    }
    return 0;
}
```

Compliant Solution [Optional Language Features]

The previous compliant solution comes close to violating [PRE09-C. Do not replace secure functions with deprecated or obsolescent functions](#), and would if a function-like macro were defined which called either `strcpy_s()` or `strcpy()` depending on if `USE_EXT1` were defined. This compliant solution solves the problem by including a custom library that implements the optional language feature, which in this case is the Safe C Library available from [SourceForge](#).

```
#if defined(__STDC_LIB_EXT1__)
#ifndef __STDC_LIB_EXT1__ >= 201112L
#define USE_EXT1 1
#define __STDC_WANT_LIB_EXT1__ 1 /* Want the ext1 functions */
#endif
#endif

#include <string.h>
#include <stdlib.h>

#ifndef USE_EXT1
#include "safe_str_lib.h"
#endif

int main(void) {
    char source_msg[] = "This is a test.";
    char *msg = malloc(sizeof(source_msg) + 1);

    if (msg != NULL) {
        strcpy_s(msg, sizeof msg, source_msg);
    }
    else {
        return EXIT_FAILURE;
    }
    return 0;
}
```

Risk Assessment

Not testing for language features or the version of the [implementation](#) being used can lead to unexpected or [undefined program behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
PRE13-C	Low	Probable	Low	P6	L2

Related Guidelines

ISO/IEC TR 24772:2013	Pre-processor Directives [NMP]
ISO/IEC 9899:2011	6.10.8, "Predefined macro names" K.3.7.1, "Copying functions"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/GYE4Bw>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
macros	Optional list of macros to restrict the check to	['__STDC__', '__STDC_HOSTED__', '__STDC_VERSION__', '__DATE__', '__FILE__', '__LINE__', '__TIME__', '__STDC_ISO_10646__', '__STDC_MB_MIGHT_NEQ_WC__', '__STDC_UTF_16__', '__STDC_UTF_32__', '__STDC_ANALYZABLE__', '__STDC_IEC_559__', '__STDC_IEC_559_COMPLEX__', '__STDC_LIB_EXT1__', '__STDC_NO_ATOMICS__', '__STDC_NO_COMPLEX__', '__STDC_NO_THREADS__', '__STDC_NO_VLA__', '__STDC_WANT_LIB_EXT1__']
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
macro_value_test_without_definition_check	Check that a macro is defined before testing its value in an #if

CertC-PRE30

Do not create a universal character name through concatenation.

Input: IR

Source languages: C, C++

Details

The C Standard supports universal character names that may be used in identifiers, character constants, and string literals to designate characters that are not in the basic character set. The universal character name \unnnnnnnn designates the character whose 8-digit short identifier (as specified by ISO/IEC 10646) is nnnnnnnn. Similarly, the universal character name \unnnn designates the character whose 4-digit short identifier is nnnn (and whose 8-digit short identifier is 0000nnnn).

The C Standard, 5.1.1.2, paragraph 4 [ISO/IEC 9899:2011], says,

If a character sequence that matches the syntax of a universal character name is produced by token concatenation (6.10.3.3), the behavior is undefined.

See also [undefined behavior 3](#).

In general, avoid universal character names in identifiers unless absolutely necessary.

Noncompliant Code Example

This code example is noncompliant because it produces a universal character name by token concatenation:

```
#define assign(uc1, uc2, val) uc1##uc2 = val

void func(void) {
    int \u0401;
    /* ... */
    assign(\u04, 01, 4);
    /* ... */
}
```

Implementation Details

This code compiles and runs with Microsoft Visual Studio 2013, assigning 4 to the variable as expected.

GCC 4.8.1 on Linux refuses to compile this code; it emits a diagnostic reading, "stray '\' in program," referring to the universal character fragment in the invocation of the `assign` macro.

Compliant Solution

This compliant solution uses a universal character name but does not create it by using token concatenation:

```
#define assign(ucn, val) ucn = val

void func(void) {
    int \u0401;
    /* ... */
    assign(\u0401, 4);
    /* ... */
}
```

Risk Assessment

Creating a universal character name through token concatenation results in undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
PRE30-C	Low	Unlikely	Medium	P2	L3

Bibliography

[ISO/IEC 10646-2003]	
[ISO/IEC 9899:2011]	Subclause 5.1.1.2, "Translation Phases"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/Zg4>], Copyright [C] 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
universal_name_by_concat	Do not create a universal character name through concatenation

CertC-PRE31

Avoid side effects in arguments to unsafe macros.

Input: IR

Source languages: C, C++

Details

An [unsafe function-like macro](#) is one whose expansion results in evaluating one of its parameters more than once or not at all. Never invoke an unsafe macro with arguments containing an assignment, increment, decrement, volatile access, input/output, or other expressions with side effects (including function calls, which may cause side effects).

The documentation for unsafe macros should warn against invoking them with arguments with side effects, but the responsibility is on the programmer using the macro. Because of the risks associated with their use, it is recommended that the creation of unsafe function-like macros be avoided. (See [PRE00-C. Prefer inline or static functions to function-like macros](#).)

This rule is similar to [EXP44-C. Do not rely on side effects in operands to sizeof, _Alignof, or _Generic](#).

Noncompliant Code Example

One problem with unsafe macros is [side effects](#) on macro arguments, as shown by this noncompliant code example:

```
#define ABS(x) ((x) < 0) ? -(x) : (x)
```

```

void func(int n) {
    /* Validate that n is within the desired range */
    int m = ABS(++n);

    /* ... */
}

```

The invocation of the `ABS()` macro in this example expands to

```
m = (((++n) < 0) ? -(++n) : (++n));
```

The resulting code is well defined but causes `n` to be incremented twice rather than once.

Compliant Solution

In this compliant solution, the increment operation `++n` is performed before the call to the unsafe macro.

```

#define ABS(x) (((x) < 0) ? -(x) : (x)) /* UNSAFE */

void func(int n) {
    /* Validate that n is within the desired range */
    ++n;
    int m = ABS(n);

    /* ... */
}

```

Note the comment warning programmers that the macro is unsafe. The macro can also be renamed `ABS_UNSAFE()` to make it clear that the macro is unsafe. This compliant solution, like all the compliant solutions for this rule, has undefined behavior if the argument to `ABS()` is equal to the minimum (most negative) value for the signed integer type. (See [INT32-C. Ensure that operations on signed integers do not result in overflow](#) for more information.)

Compliant Solution

This compliant solution follows the guidance of [PRE00-C. Prefer inline or static functions to function-like macros](#) by defining an inline function `iabs()` to replace the `ABS()` macro. Unlike the `ABS()` macro, which operates on operands of any type, the `iabs()` function will truncate arguments of types wider than `int` whose value is not in range of the latter type.

```

#include <complex.h>
#include <math.h>

static inline int iabs(int x) {
    return ((x) < 0) ? -(x) : (x);
}

void func(int n) {
    /* Validate that n is within the desired range */

    int m = iabs(++n);

    /* ... */
}

```

Compliant Solution

A more flexible compliant solution is to declare the `ABS()` macro using a `_Generic` selection. To support all arithmetic data types, this solution also makes use of inline functions to compute integer absolute values. (See [PRE00-C. Prefer inline or static functions to function-like macros](#) and [PRE12-C. Do not define unsafe macros](#).)

According to the C Standard, 6.5.1.1, paragraph 3 [[ISO/IEC 9899:2011](#)]:

The controlling expression of a generic selection is not evaluated. If a generic selection has a generic association with a type name that is compatible with the type of the controlling expression, then the result expression of the generic selection is the expression in that generic association. Otherwise, the result expression of the generic selection is the expression in the `default` generic association. None of the expressions from any other generic association of the generic selection is evaluated.

Because the expression is not evaluated as part of the generic selection, the use of a macro in this solution is guaranteed to evaluate the macro parameter `v` only once.

```

#include <complex.h>
#include <math.h>

static inline long long llabs(long long v) {
    return v < 0 ? -v : v;
}
static inline long labs(long v) {
    return v < 0 ? -v : v;
}
static inline int iabs(int v) {
    return v < 0 ? -v : v;
}
static inline int sabs(short v) {
    return v < 0 ? -v : v;
}
static inline int scabs(signed char v) {
    return v < 0 ? -v : v;
}

#define ABS(v) _Generic(v, signed char : scabs, \
                      short : sabs, \
                      int : iabs, \
                      long : labs, \

```

```

long long : llabs, \
float : fabsf, \
double : fabs, \
long double : fabsl, \
double complex : cabs, \
float complex : cabsf, \
long double complex : cabsl)(v)

void func(int n) {
    /* Validate that n is within the desired range */
    int m = ABS(++n);
    /* ... */
}

```

Generic selections were introduced in C11 and are not available in C99 and earlier editions of the C Standard.

Compliant Solution (GCC)

GCC's `__typeof__` extension makes it possible to declare and assign the value of the macro operand to a temporary of the same type and perform the computation on the temporary, consequently guaranteeing that the operand will be evaluated exactly once. Another GCC extension, known as `statement expression`, makes it possible for the block statement to appear where an expression is expected:

```
#define ABS(x) __extension__ ({ __typeof__ (x) tmp = x; \
                           tmp < 0 ? -tmp : tmp; })
```

Note that relying on such extensions makes code nonportable and violates [MSC14-C. Do not introduce unnecessary platform dependencies](#).

Noncompliant Code Example (`assert()`)

The `assert()` macro is a convenient mechanism for incorporating diagnostic tests in code. (See [MSC11-C. Incorporate diagnostic tests using assertions](#).) Expressions used as arguments to the standard `assert()` macro should not have side effects. The behavior of the `assert()` macro depends on the definition of the object-like macro `NDEBUG`. If the macro `NDEBUG` is undefined, the `assert()` macro is defined to evaluate its expression argument and, if the result of the expression compares equal to 0, call the `abort()` function. If `NDEBUG` is defined, `assert` is defined to expand to `((void)0)`. Consequently, the expression in the assertion is not evaluated, and no side effects it may have had otherwise take place in non-debugging executions of the code.

This noncompliant code example includes an `assert()` macro containing an expression (`index++`) that has a side effect:

```
#include <assert.h>
#include <stddef.h>

void process(size_t index) {
    assert(index++ > 0); /* Side effect */
    /* ... */
}
```

Compliant Solution (`assert()`)

This compliant solution avoids the possibility of side effects in assertions by moving the expression containing the side effect outside of the `assert()` macro.

```
#include <assert.h>
#include <stddef.h>

void process(size_t index) {
    assert(index > 0); /* No side effect */
    ++index;
    /* ... */
}
```

Exceptions

PRE31-C-EX1: An exception can be made for invoking an `unsafe macro` with a function call argument provided that the function has no `side effects`. However, it is easy to forget about obscure side effects that a function might have, especially library functions for which source code is not available; even changing `errno` is a side effect. Unless the function is user-written and does nothing but perform a computation and return its result without calling any other functions, it is likely that many developers will forget about some side effect. Consequently, this exception must be used with great care.

Risk Assessment

Invoking an unsafe macro with an argument that has side effects may cause those side effects to occur more than once. This practice can lead to [unexpected program behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
PRE31-C	Low	Unlikely	Low	P3	L3

Related Guidelines

SEI CERT C Coding Standard	PRE00-C. Prefer inline or static functions to function-like macros PRE12-C. Do not define unsafe macros MSC14-C. Do not introduce unnecessary platform dependencies DCL37-C. Do not declare or define a reserved identifier
SEI CERT C++ Coding Standard	PRE31-CPP. Avoid side-effects in arguments to unsafe macros
CERT Oracle Secure Coding Standard for Java	EXP06-J. Expressions used in assertions must not produce side effects
ISO/IEC TR 24772:2013	Pre-processor Directives [NMP]
MISRA C:2012	Rule 20.5 (advisory)

Bibliography

[Dewhurst 2002]	Gotcha #28, "Side Effects in Assertions"
[ISO/IEC 9899:2011]	Subclause 6.5.1.1, "Generic Selection"
[Plum 1985]	Rule 1-11

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/agBi>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
side_effect_in_unsafe_macro_call	Side-effect in call to unsafe macro

CertC-PRE32

Do not use preprocessor directives in invocations of function-like macros.

Input: IR

Source languages: C, C++

Details

The arguments to a macro must not include preprocessor directives, such as `#define`, `#ifdef`, and `#include`. Doing so results in [undefined behavior](#), according to the C Standard, 6.10.3, paragraph 11 [[ISO/IEC 9899:2011](#)]:

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, the behavior is undefined.

See also [undefined behavior 93](#).

This rule also applies to the use of preprocessor directives in arguments to a function where it is unknown whether or not the function is implemented using a macro. For example, standard library functions, such as `memcpy()`, `printf()`, and `assert()`, may be implemented as macros.

Noncompliant Code Example

In this noncompliant code example [[GCC Bugs](#)], the programmer uses preprocessor directives to specify platform-specific arguments to `memcpy()`. However, if `memcpy()` is implemented using a macro, the code results in undefined behavior.

```
#include <string.h>

void func(const char *src) {
    /* Validate the source string; calculate size */
    char *dest;
    /* malloc() destination string */
```

```

memcpy(dest, src,
#ifndef PLATFORM1
    12
#else
    24
#endif
);
/* ... */
);

```

Compliant Solution

In this compliant solution [[GCC Bugs](#)], the appropriate call to `memcpy()` is determined outside the function call:

```

#include <string.h>

void func(const char *src) {
    /* Validate the source string; calculate size */
    char *dest;
    /* malloc() destination string */
#ifndef PLATFORM1
    memcpy(dest, src, 12);
#else
    memcpy(dest, src, 24);
#endif
    /* ... */
}

```

Risk Assessment

Including preprocessor directives in macro arguments is undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
PRE32-C	Low	Unlikely	Medium	P2	L3

Bibliography

[GCC Bugs]	"Non-bugs"
[ISO/IEC 9899:2011]	6.10.3, "Macro Replacement"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/JYC2AQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
directives	Preprocessing directives to check for in macro arguments.	{'#if', '#ifdef', '#ifndef', '#elif', '#else', '#endif', '#pragma', '#warning', '#error', '#line', '#include', '#include_next', '#ident', '#region', '#endregion', '#asm', '#endasm', '#define', '#undef'}
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
pp_directive_as_macro_arg	Macro invocation argument looks like preprocessing directive

CertC-DCL00

Const-qualify immutable objects.

Input: IR

Source languages: C, C++

Details

Immutable objects should be `const`-qualified. Enforcing object immutability using `const` qualification helps ensure the correctness and security of applications. ISO/IEC TR 24772, for example, recommends labeling parameters as constant to avoid the unintentional modification of function arguments [[ISO/IEC TR 24772](#)]. [STR05-C. Use pointers to const when referring to string literals](#) describes a specialized case of this recommendation.

Adding `const` qualification may propagate through a program; as you add `const`, qualifiers become still more necessary. This phenomenon is sometimes called *const poisoning*, which can frequently lead to violations of [EXP05-C. Do not cast away a const qualification](#). Although `const` qualification is a good idea, the costs may outweigh the value in the remediation of existing code.

A macro or an enumeration constant may also be used instead of a `const`-qualified object. [DCL06-C. Use meaningful symbolic constants to represent literal values](#) describes the relative merits of using `const`-qualified objects, enumeration constants, and object-like macros. However, adding a `const` qualifier to an existing variable is a better first step than replacing the variable with an enumeration constant or macro because the compiler will issue warnings on any code that changes your `const`-qualified variable. Once you have verified that a `const`-qualified variable is not changed by any code, you may consider changing it to an enumeration constant or macro, as best fits your design.

Noncompliant Code Example

In this noncompliant code, `pi` is declared as a `float`. Although `pi` is a mathematical constant, its value is not protected from accidental modification.

```
float pi = 3.14159f;
float degrees;
float radians;
/* ... */
radians = degrees * pi / 180;
```

Compliant Solution

In this compliant solution, `pi` is declared as a `const`-qualified object:

```
const float pi = 3.14159f;
float degrees;
float radians;
/* ... */
radians = degrees * pi / 180;
```

Risk Assessment

Failing to `const`-qualify immutable objects can result in a constant being modified at runtime.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL00-C	Low	Unlikely	High	P1	L3

Related Guidelines

[SEI CERT C++ Coding Standard](#) | [DCL00-CPP. Const-qualify immutable objects](#)

Bibliography

[Dewhurst 2002]	Gotcha #25, "#define Literals"
[Saks 2000]	

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/vQ4>], Copyright [C] 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
ignore_pointer_variables	Whether variables of pointer type should be ignored.	False
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
only_check_unit_locals	Whether only local variables and global static variables should be checked.	False
only_immutable_data	Whether only declarations with immutable data should be checked.	False
priority	Priority based on the combination of severity, likelihood and remediation cost	1
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	high
report_only_at_definition	Report violations for non-const parameters only at the function definition, not the function declaration.	False

Possible Messages

Name	Message
parameter_missing_const	A parameter which is not modified shall be const qualified.
variable_missing_const	A variable which is not modified shall be const qualified.

CertC-DCL01

Do not reuse variable names in subscopes.

Input: IR

Source languages: C, C++

Details

Do not use the same variable name in two scopes where one scope is contained in another. For example,

- No other variable should share the name of a global variable if the other variable is in a subscope of the global variable.
- A block should not declare a variable with the same name as a variable declared in any block that contains it.

Reusing variable names leads to programmer confusion about which variable is being modified. Additionally, if variable names are reused, generally one or both of the variable names are too generic.

Noncompliant Code Example

This noncompliant code example declares the `msg` identifier at file scope and reuses the same identifier to declare a character array local to the `report_error()` function. The programmer may unintentionally copy the function argument to the locally declared `msg` array within the `report_error()` function. Depending on the programmer's intention, it either fails to initialize the global variable `msg` or allows the local `msg` buffer to overflow by using the global value `msgsize` as a bounds for the local buffer.

```
#include <stdio.h>

static char msg[100];
static const size_t msgsize = sizeof( msg);

void report_error(const char *str) {
    char msg[80];
    snprintf(msg, msgsize, "Error: %s\n", str);
    /* ... */
}

int main(void) {
    /* ... */
    report_error("some error");

    return 0;
}
```

Compliant Solution

This compliant solution uses different, more descriptive variable names:

```
#include <stdio.h>

static char message[100];
static const size_t message_size = sizeof( message);

void report_error(const char *str) {
    char msg[80];
    snprintf(msg, message_size, "Error: %s\n", str);
    /* ... */
}

int main(void) {
    /* ... */
    report_error("some error");

    return 0;
}
```

When the block is small, the danger of reusing variable names is mitigated by the visibility of the immediate declaration. Even in this case, however, variable name reuse is not desirable. In general, the larger the declarative region of an identifier, the more descriptive and verbose should be the name of the identifier.

By using different variable names globally and locally, the compiler forces the developer to be more precise and descriptive with variable names.

Noncompliant Code Example

This noncompliant code example declares two variables with the same identifier, but in slightly different scopes. The scope of the identifier `i` declared in the `for` loop's initial clause terminates after the closing curly brace of the `for` loop. The scope of the identifier `i` declared in the `for` loop's compound statement terminates before the closing curly brace. Thus, the inner declaration of `i` hides the outer declaration of `i`, which can lead to unintentionally referencing the wrong object.

```

void f(void) {
    for (int i = 0; i < 10; i++) {
        long i;
        /* ... */
    }
}

```

Compliant Solution

This compliant solution uses a unique identifier for the variable declared within the `for` loop.

```

void f(void) {
    for (int i = 0; i < 10; i++) {
        long j;
        /* ... */
    }
}

```

Exceptions

DCL01-C-EX1: A function argument in a function declaration may clash with a variable in a containing scope provided that when the function is defined, the argument has a name that clashes with no variables in any containing scopes.

```

extern int name;
void f(char *name); /* Declaration: no problem here */
/* ... */
void f(char *arg) { /* Definition: no problem; arg doesn't hide name */
    /* Use arg */
}

```

DCL01-C-EX2: A temporary variable within a new scope inside of a macro can override a surrounding identifier.

```

#define SWAP(type, a, b) do { type tmp = a; a = b; b = tmp; } while(0)

void func(void) {
    int tmp = 100;
    int a = 10, b = 20;
    SWAP(int, a, b); /* Hidden redeclaration of tmp is acceptable */
}

```

Risk Assessment

Reusing a variable name in a subspace can lead to unintentionally referencing an incorrect variable.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL01-C	Low	Unlikely	Medium	P2	L3

Related Guidelines

SEI CERT C++ Coding Standard	DCL01-CPP. Do not reuse variable names in subsscopes
MISRA C:2012	Rule 5.3 (required)

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/VwE>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
allow_in_classes	Whether identifiers in local classes can hide surrounding ones	False
allow_in_lambdas	Whether identifiers in lambdas can hide surrounding ones	False
allow_in_local_classes	Whether identifiers in local classes can hide surrounding ones	False
allow_in_namespaces	Whether identifiers in namespaces can hide surrounding ones	False
allow_parameter_in_function_declaration	Whether identifiers of parameters in function declarations may hide surrounding ones, given the function declaration uses a different identifier for the same parameter that does not hide a surrounding one.	True
check_inheritance_hiding	Whether to check for hiding by inheritance.	False
check_nested_classes	Whether symbols from inner classes should be reported as hiding symbols of outer classes.	False
distinguish_name_spaces	Whether C name spaces should be distinguished.	False
hiding_entities	LIR node types to consider symbols that hide other symbols.	{'Variable', 'Typedef_Type', 'Field', 'Parameter'}
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
maxlen	Number of significant characters (or None)	None
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium
tolerate_constructor_parameters_hiding_fields	Allow constructor parameters to hide fields	True
tolerate_function_prototype_argument_hiding	Whether to tolerate arguments of a function prototype, that itself is a function argument, that have the same identifier as a argument of the outer function. If 'True' is given, the rule will still report cases, where argument identifiers match up and the function prototype refers to arguments of the outer function (e.g. by `decltype()`).	False
tolerate_macro_local_identifier	Whether an identifier being local to a macro is allowed to hide identifiers from outside the macro	True
unchecked_types		{'Routine_Call', 'Stack_Object_Definition'}

Possible Messages

Name	Message
hiding	{} hides {}

CertC-DCL02

Use visually distinct identifiers.

Input: IR

Source languages: C, C++

Details

Use visually distinct identifiers with meaningful names to eliminate errors resulting from misreading the spelling of an identifier during the development and review of code. An identifier can denote an object; a function; a tag or a member of a structure, union, or enumeration; a typedef name; a label name; a macro name; or a macro parameter.

Depending on the fonts used, certain characters appear visually similar or even identical:

Character	Similar Characters
0 (zero)	o (capital o), Q (capital q), D (capital a)
1 (one)	I (capital i), l (lowercase L)
2 (two)	z (capital z)
5 (five)	s (capital s)
8 (eight)	B (capital b)
n (lowercase n)	h (lowercase H)
m (lowercase M)	rn (lowercase R, lowercase N)

Do not define multiple identifiers that vary only with respect to one or more visually similar characters.

Make the initial portions of long identifiers unique for easier recognition and to help prevent errors resulting from nonunique identifiers. (See [DCL23-C. Guarantee that mutually visible identifiers are unique](#).)

In addition, the larger the scope of an identifier, the more descriptive its name should be. It may be perfectly appropriate to name a loop control variable `i`, but the same name would likely be confusing if it named a file scope object or a variable local to a function more than a few lines long. See also [DCL01-C. Do not reuse variable names in subscopes](#) and [DCL19-C. Use as minimal a scope as possible for all variables and functions](#).

Noncompliant Code Example (Source Character Set)

DCL02-C implicitly assumes *global scope*, which can be confused with *scope within the same file*. Although it may not generate any errors, a possible violation of the rule may occur, as in the following example. Note this example does not violate [DCL23-C. Guarantee that mutually visible identifiers are unique](#).

In file `foo.h`:

```
int id_0; /* (Capital letter O) */
```

In file `bar.h`:

```
int id_0; /* (Numeric digit zero) */
```

If a file `foobar.c` includes both `foo.h` and `bar.h`, then both `id_0` and `id_0` come in the same scope, violating this rule.

Compliant Solution (Source Character Set)

In a compliant solution, use of visually similar identifiers should be avoided in the same project scope.

In file `foo.h`:

```
int id_a;
```

In file `bar.h`:

```
int id_b;
```

Risk Assessment

Failing to use visually distinct identifiers can result in referencing the wrong object or function, causing unintended program behavior.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL02-C	Low	Unlikely	Medium	P2	L3

Related Guidelines

SEI CERT C++ Coding Standard	DCL02-CPP. Use visually distinct identifiers
ISO/IEC TR 24772:2013	Choice of Clear Names [NAI]
MISRA C:2012	Directive 4.5 (advisory)

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/SQU>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
distinguish_c_name_spaces	Whether to allow similar identifiers in different C name spaces (type tag/label/member/ordinary identifier)	True
distinguish_macro_names	Whether to allow macros to have similar names as other identifiers.	False
distinguish_type_names	Whether to allow types to have similar names as other identifiers. For example, a variable declaration "MyClass myClass;" is allowed by this option. Note: unlike the option 'distinguish_c_name_spaces', this option also distinguishes typedef names, not just struct/class tag names.	False
exclude_system_headers	Whether to exclude identifiers appearing in system headers	False
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
normalizations	Which pairs of characters should be seen as ambiguous	[('0', '0'), ('Q', '0'), ('D', '0'), ('1', 'l'), ('I', 'l'), ('5', 'S'), ('2', 'Z'), ('8', 'B'), ('rn', 'm'), ('n', 'h')]
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
ambiguous	Identifiers are typographically ambiguous
casing	Identifiers only differ in casing

CertC-DCL03

Use a static assertion to test the value of a constant expression.

Input: IR

Source languages: C, C++

Details

Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in [vulnerabilities](#) (see [MSC11-C. Incorporate diagnostic tests using assertions](#)). The runtime `assert()` macro has some limitations, however, in that it incurs a runtime overhead and because it calls `abort()`. Consequently, the runtime `assert()` macro is useful only for identifying incorrect assumptions and not for runtime error checking. As a result, runtime assertions are generally unsuitable for server programs or embedded systems.

Static assertion is a new facility in the C Standard. It takes the form

```
static_assert(constant-expression, string-literal);
```

Subclause 6.7.10 of the C Standard [[ISO/IEC 9899:2011](#)] states:

The constant expression shall be an integer constant expression. If the value of the constant expression compares unequal to 0, the declaration has no effect. Otherwise, the constraint is violated and the implementation shall produce a diagnostic message that includes the text of the string literal, except that characters not in the basic source character set are not required to appear in the message.

It means that if `constant-expression` is true, nothing will happen. However, if `constant-expression` is false, an error message containing `string-literal` will be output at compile time.

```
/* Passes */
static_assert(
    sizeof(int) <= sizeof(void*),
    "sizeof(int) <= sizeof(void*)"
);

/* Fails */
static_assert(
    sizeof(double) <= sizeof(int),
    "sizeof(double) <= sizeof(int)"
);
```

Static assertion is not available in C99.

Noncompliant Code Example

This noncompliant code uses the `assert()` macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly:

```
#include <assert.h>

struct timer {
    unsigned char MODE;
    unsigned int DATA;
    unsigned int COUNT;
};

int func(void) {
    assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));
}
```

Although the use of the runtime assertion is better than nothing, it needs to be placed in a function and executed. This means that it is usually far away from the definition of the actual structure to which it refers. The diagnostic occurs only at runtime and only if the code path containing the assertion is executed.

Compliant Solution

For assertions involving only constant expressions, a preprocessor conditional statement may be used, as in this compliant solution:

```
struct timer {
    unsigned char MODE;
    unsigned int DATA;
    unsigned int COUNT;
};

#if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))
#error "Structure must not have any padding"
#endif
```

Using `#error` directives allows for clear diagnostic messages. Because this approach evaluates assertions at compile time, there is no runtime penalty.

Compliant Solution

This portable compliant solution uses `static_assert`:

```
#include <assert.h>

struct timer {
    unsigned char MODE;
    unsigned int DATA;
    unsigned int COUNT;
};

static_assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int),
             "Structure must not have any padding");
```

Static assertions allow incorrect assumptions to be diagnosed at compile time instead of resulting in a silent malfunction or runtime error. Because the assertion is performed at compile time, no runtime cost in space or time is incurred. An assertion can be used at file or block scope, and failure results in a meaningful and informative diagnostic error message.

Other uses of static assertion are shown in [STR07-C. Use the bounds-checking interfaces for string manipulation](#) and [FIO34-C. Distinguish between characters read from a file and EOF or WEOF](#).

Risk Assessment

Static assertion is a valuable diagnostic tool for finding and eliminating software defects that may result in [vulnerabilities](#) at compile time. The absence of static assertions, however, does not mean that code is incorrect.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL03-C	Low	Unlikely	High	P1	L3

Related Guidelines

[C++ Secure Coding Standard](#) | [DCL03-CPP. Use a static assertion to test the value of a constant expression](#)

Bibliography

[Becker 2008]	
[Eckel 2007]	
[ISO/IEC 9899:2011]	Subclause 6.7.10, "Static Assertions"
[Jones 2010]	
[Klarer 2004]	
[Saks 2005]	
[Saks 2008]	

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/tgCs>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
macro_names	Specifies the names of the assertion macros.	set(['assert'])
priority	Priority based on the combination of severity, likelihood and remediation cost	1
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
runtime_assert_could_be_static	The condition of this runtime assert is constant -- use a static assertion instead

CertC-DCL04

Do not declare more than one variable per declaration.

Input: IR

Source languages: C, C++

Details

Every declaration should be for a single variable, on its own line, with an explanatory comment about the role of the variable. Declaring multiple variables in a single declaration can cause confusion regarding the types of the variables and their initial values. If more than one variable is declared in a declaration, care must be taken that the type and initialized value of the variable are handled correctly.

Noncompliant Code Example

In this noncompliant code example, a programmer or code reviewer might mistakenly believe that the two variables `src` and `c` are declared as `char *`. In fact, `src` has a type of `char *`, whereas `c` has a type of `char`.

```
char *src = 0, c = 0;
```

Compliant Solution

In this compliant solution, each variable is declared on a separate line:

```
char *src; /* Source string */
char c; /* Character being tested */
```

Although this change has no effect on compilation, the programmer's intent is clearer.

Noncompliant Code Example

In this noncompliant code example, a programmer or code reviewer might mistakenly believe that both `i` and `j` have been initialized to 1. In fact, only `j` has been initialized, and `i` remains uninitialized.

```
int i, j = 1;
```

Compliant Solution

In this compliant solution, it is readily apparent that both `i` and `j` have been initialized to 1:

```
int i = 1;
int j = 1;
```

Exceptions

DCL04-C-EX1: Multiple loop control variables can be declared in the same `for` statement, as shown in the following function:

```
#include <limits.h> /* For CHAR_BIT */
#include <stddef.h> /* For size_t */

extern size_t popcount(uintmax_t);
#define PRECISION(umax_value) popcount(umax_value)

size_t bitcount(size_t n) {
    const size_t limit = PRECISION(SIZE_MAX);
    size_t count = 0;
    for (size_t i = 0, j = 1; i < limit; ++i, j <= 1) {
        if (n & j)
            ++count;
    }
    return count;
}
```

The `PRECISION()` macro provides the correct precision for any integer type and is defined in [INT35-C. Use correct integer precisions](#) see that rule for more information.

DCL04-C-EX2: Multiple, simple variable declarations can be declared on the same line given that there are no initializations. A simple variable declaration is one that is not a pointer or array.

```
int i, j, k;
```

Risk Assessment

Declaring no more than one variable per declaration can make code easier to read and eliminate confusion.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL04-C	Low	Unlikely	Low	P3	L3

Related Guidelines

[SEI CERT C++ Coding Standard](#) [DCL04-CPP. Do not declare more than one variable per declaration](#)

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/VgU>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
allow_in_for_loop_init	Whether a multi-declaration is allowed in the for-init statement	True
allow_uninitialized_simple_type	Whether a multi-declaration of simple type is allowed when there is no initialization	True
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
multi_declaration	Multiple declarators in single declaration

CertC-DCL05

Use typedefs of non-pointer types only.

Input: IR

Source languages: C, C++

Details

Using type definitions (`typedef`) can often improve code readability. However, type definitions to pointer types can make it more difficult to write `const`-correct code because the `const` qualifier will be applied to the pointer type, not to the underlying declared type.

Noncompliant Code Example

The following type definition improves readability at the expense of introducing a `const`-correctness issue. In this example, the `const` qualifier applies to the `typedef` instead of to the underlying object type. Consequently, `func` does not take a pointer to a `const` struct `obj` but instead takes a `const` pointer to a struct `obj`.

```
struct obj {
    int i;
    float f;
};

typedef struct obj *ObjectPtr;

void func(const ObjectPtr o) {
    /* Can actually modify o's contents, against expectations */
}
```

Compliant Solution

This compliant solution makes use of type definitions but does not declare a pointer type and so cannot be used in a `const`-incorrect manner:

```
struct obj {
    int i;
    float f;
};

typedef struct obj Object;

void func(const Object *o) {
    /* Cannot modify o's contents */
}
```

Noncompliant Code Example (Windows)

The Win32 SDK headers make use of type definitions for most of the types involved in Win32 APIs, but this noncompliant code example demonstrates a `const`-correctness bug:

```
#include <Windows.h>
/* typedef char *LPSTR; */

void func(const LPSTR str) {
    /* Can mutate str's contents, against expectations */
}
```

Compliant Solution (Windows)

This compliant solution demonstrates a common naming convention found in the Win32 APIs, using the proper `const` type:

```
#include <Windows.h>
/* typedef const char *LPCSTR; */

void func(LPCSTR str) {
    /* Cannot modify str's contents */
}
```

Noncompliant Code Example (Windows)

Note that many structures in the Win32 API are declared with pointer type definitions but not pointer-to-`const` type definitions (`LPPOINT`, `LPSIZE`, and others). In these cases, it is suggested that you create your own type definition from the base structure type.

```
#include <Windows.h>
/*
typedef struct tagPOINT {
    long x, y;
} POINT, *LPOINT;
*/

void func(const LPOINT pt) {
    /* Can modify pt's contents, against expectations */
}
```

Compliant Solution (Windows)

```
#include <Windows.h>
/*
typedef struct tagPOINT {
    long x, y;
} POINT, *LPOINT;
*/

typedef const POINT *LPCPOINT;
void func(LPCPOINT pt) {
    /* Cannot modify pt's contents */
}
```

Noncompliant Code Example

In this noncompliant code example, the declaration of the `signal()` function is difficult to read and comprehend:

```
void (*signal(int, void (*)(int)))(int);
```

Compliant Solution

This compliant solution makes use of type definitions to specify the same type as in the noncompliant code example:

```
typedef void SighandlerType(int signum);
extern SighandlerType *signal(
    int signum,
    SighandlerType *handler
);
```

Exceptions

Function pointer types are an exception to this recommendation.

Risk Assessment

Code readability is important for discovering and eliminating [vulnerabilities](#).

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL05-C	Low	Unlikely	Medium	P2	L3

Related Guidelines

SEI CERT C Coding Standard	DCL12-C. Implement abstract data types using opaque types
SEI CERT C++ Coding Standard	DCL05-CPP. Use typedefs to improve code readability

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/14At>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
function_pointer_missing_typedef	Use typedefs for function pointers to improve readability
pointer_typedef	Typedef for non-const pointer should not be used

CertC-DCL06

Use meaningful symbolic constants to represent literal values.

Input: IR

Source languages: C, C++

Details

The C language provides several different kinds of constants: *integer* constants, such as `10` and `0x1c`; *floating* constants, such as `1.0` and `6.022e+23`; and *character* constants, such as `'a'` and `'\x10'`. C also provides string literals, such as `"hello, world"` and `"\n"`. These constants can all be referred to as *literals*.

When used in program logic, literals can reduce the readability of source code. As a result, literals, in general, and integer constants, in particular, are frequently called *magic numbers* because their purpose is often obscured. Magic numbers can be constant values that represent either an arbitrary value (such as a determined appropriate buffer size) or a malleable concept (such as the age at which a person is considered an adult, which can change between geopolitical boundaries). Rather than embed literals in program logic, use appropriately named symbolic constants to clarify the intent of the code. In addition, if a specific value needs to be changed, reassigning a symbolic constant once is more efficient and less error prone than replacing every instance of the value [[Saks 2002](#)].

The C programming language has several mechanisms for creating named, symbolic constants: `const`-qualified objects, enumeration constants, and [object-like macro](#) definitions. Each of these mechanisms has associated advantages and disadvantages.

const-Qualified Objects

Objects that are `const`-qualified have scope and can be type-checked by the compiler. Because they are named objects (unlike macro definitions), some debugging tools can show the name of the object. The object also consumes memory.

A `const`-qualified object allows you to specify the exact type of the constant. For example,

```
const unsigned int buffer_size = 256;
```

defines `buffer_size` as a constant whose type is `unsigned int`.

Unfortunately, `const`-qualified objects cannot be used where compile-time integer constants are required, namely to define the

- Size of a bit-field member of a structure.
- Size of an array (except in the case of variable length arrays).
- Value of an enumeration constant.
- Value of a `case` constant.

If any of these are required, then an integer constant (which would be an [rvalue](#)) must be used.

`const`-qualified objects allow the programmer to take the address of the object:

```
const int max = 15;
int a[max]; /* Invalid declaration outside of a function */
const int *p;

/* A const-qualified object can have its address taken */
p = &max;
```

`const`-qualified objects are likely to incur some runtime overhead [[Saks 2001b](#)]. Most C compilers, for example, allocate memory for `const`-qualified objects. `const`-qualified objects declared inside a function body can have automatic storage duration. If so, the compiler will allocate storage for the object, and it will be on the stack. As a result, this storage will need to be allocated and initialized each time the containing function is invoked.

Enumeration Constants

Enumeration constants can be used to represent an integer constant expression that has a value representable as an `int`. Unlike `const`-qualified objects, enumeration constants do not consume memory. No storage is allocated for the value, so it is not possible to take the address of an enumeration constant.

```
enum { max = 15 };
int a[max]; /* OK outside function */
const int *p;

p = &max; /* Error: "&" on enum constant */
```

Enumeration constants do not allow the type of the value to be specified. An enumeration constant whose value can be represented as an `int` is always an `int`.

Object-like Macros

A preprocessing directive of the form

```
# define identifier replacement-list
```

defines an [object-like macro](#) that causes each subsequent instance of the macro name to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive.

C programmers frequently define symbolic constants as object-like macros. For example, the code

```
#define buffer_size 256
```

defines `buffer_size` as a macro whose value is 256. The preprocessor substitutes macros before the compiler does any other symbol processing. Later compilation phases never see macro symbols, such as `buffer_size`; they see only the source text after macro substitution. As a result, many compilers do not preserve macro names among the symbols they pass on to their debuggers.

Macro names do not observe the scope rules that apply to other names. Therefore, macros could substitute in unanticipated places with unexpected results.

Object-like macros do not consume memory; consequently, it is not possible to create a pointer to one. Macros do not provide for type checking because they are textually replaced by the preprocessor.

Macros can be passed as compile-time arguments.

Summary

The following table summarizes some of the differences between `const`-qualified objects, enumeration constants, and [object-like macro](#) definitions.

Method	Evaluated At	Consumes Memory	Viewable by Debuggers	Type Checking	Compile-Time Constant Expression
Enumerations	Compile time	No	Yes	Yes	Yes
<code>const</code> -qualified	Runtime	Yes	Yes	Yes	No
Macros	Preprocessor	No	No	No	Yes

Noncompliant Code Example

The meaning of the integer literal 18 is not clear in this example:

```
/* ... */
if (age >= 18) {
    /* Take action */
}
else {
    /* Take a different action */
}
/* ... */
```

Compliant Solution

This compliant solution replaces the integer literal 18 with the symbolic constant `ADULT_AGE` to clarify the meaning of the code:

```
enum { ADULT_AGE=18 };
/* ... */
if (age >= ADULT_AGE) {
    /* Take action */
}
else {
    /* Take a different action */
}
/* ... */
```

Noncompliant Code Example

Integer literals are frequently used when referring to array dimensions, as shown in this noncompliant code example:

```
char buffer[256];
/* ... */
fgets(buffer, 256, stdin);
```

This use of integer literals can easily result in buffer overflows if, for example, the buffer size is reduced but the integer literal used in the call to `fgets()` is not.

Compliant Solution (enum)

In this compliant solution, the integer literal is replaced with an enumeration constant. (See [DCL00-C. Const-qualify immutable objects](#).)

```
enum { BUFFER_SIZE=256 };
char buffer[BUFFER_SIZE];
/* ... */
fgets(buffer, BUFFER_SIZE, stdin);
```

Enumeration constants can safely be used anywhere a constant expression is required.

Compliant Solution (sizeof)

Frequently, it is possible to obtain the desired readability by using a symbolic expression composed of existing symbols rather than by defining a new symbol. For example, a `sizeof` expression can work just as well as an enumeration constant. (See [EXP09-C. Use sizeof to determine the size of a type or variable](#).)

```
char buffer[256];
/* ... */
fgets(buffer, sizeof(buffer), stdin);
```

Using the `sizeof` expression in this example reduces the total number of names declared in the program, which is generally a good idea [[Saks 2002](#)]. The `sizeof` operator is almost always evaluated at compile time (except in the case of variable-length arrays).

When working with `sizeof()`, keep in mind [ARR01-C. Do not apply the sizeof operator to a pointer when taking the size of an array](#).

Noncompliant Code Example

In this noncompliant code example, the string literal "localhost" and integer constant 1234 are embedded directly in program logic and are consequently difficult to change:

```
LDAP *ld = ldap_init("localhost", 1234);
if (ld == NULL) {
    perror("ldap_init");
    return(1);
}
```

Compliant Solution

In this compliant solution, the host name and port number are both defined as [object-like macros](#), so they can be passed as compile-time arguments:

```
#ifndef PORTNUMBER      /* Might be passed on compile line */
# define PORTNUMBER 1234
#endif

#ifndef HOSTNAME        /* Might be passed on compile line */
# define HOSTNAME "localhost"
#endif

/* ... */
```

```

LDAP *ld = ldap_init(HOSTNAME, PORTNUMBER);
if (ld == NULL) {
    perror("ldap_init");
    return(1);
}

```

Exceptions

DCL06-C-EX1: Although replacing numeric constants with a symbolic constant is often a good practice, it can be taken too far. Remember that the goal is to improve readability. Exceptions can be made for constants that are themselves the abstraction you want to represent, as in this compliant solution.

```
x = (-b + sqrt(b*b - 4*a*c)) / (2*a);
```

Replacing numeric constants with symbolic constants in this example does nothing to improve the readability of the code and can actually make the code more difficult to read.

```

enum { TWO = 2 };      /* A scalar */
enum { FOUR = 4 };     /* A scalar */
enum { SQUARE = 2 };   /* An exponent */
x = (-b + sqrt(pow(b, SQUARE) - FOUR*a*c))/ (TWO * a);

```

When implementing recommendations, it is always necessary to use sound judgment.

Note that this example does not check for invalid operations (taking the `sqrt()` of a negative number). See [FLP32-C. Prevent or detect domain and range errors in math functions](#) for more information on detecting domain and range errors in math functions.

Risk Assessment

Using numeric literals makes code more difficult to read and understand. Buffer overruns are frequently a consequence of a magic number being changed in one place (such as in an array declaration) but not elsewhere (such as in a loop through an array).

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL06-C	Low	Unlikely	Medium	P2	L3

Related Guidelines

SEI CERT C++ Coding Standard	DCL06-CPP. Use meaningful symbolic constants to represent literal values in program logic
MITRE CWE	CWE-547. Use of hard-coded, security-relevant constants

Bibliography

[Henricson 1992]	Chapter 10, " Constants "
[Saks 2001a]	
[Saks 2001b]	
[Saks 2002]	
[Summit 2005]	Question 10.5b

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/hYAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
allow_nonconst_variable_initialization	If set to `True`, allow the initialization of a non-const variable with a string literal in addition to the initialization of a const variable. A reassignment of such a variable is still a violation.	False
allowed	Literal values that are ok.	{0.0, 1.0, 2.0}
allowed_contexts	Optional set of PIR classes or functions ([node] -> bool) for allowed contexts, e.g. Case_Label.	set[[]]
allowed_logging_contexts	List of fully qualified function types that are considered logging contexts (using operator<<). If this is non-empty, `std::throw()` is considered a valid logging context as well.	[]
allowed_string_contexts	Optional set of PIR classes or functions ([node] -> bool) for allowed contexts.	set[[]]
allowed_strings	Literal values that are ok.	["'"]
exceptions	Optional predicate to filter out cases that should be allowed as an exception. The predicate takes a literal node and returns True if the given node is such an exception.	None
exclude_pp_literals	If True, literals in conditions of #if are ignored.	True
exclude_single_uses	If True, report only literals used more than once.	False
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
magic_number	Use of magic literal.
magic_number_without_token	Use of magic literal.
magic_string	Use of magic string literal.
possible_magic_number	Potential use of magic literal.

CertC-DCL07

Include the appropriate type information in function declarators.

Input: IR

Source languages: C, C++

Details

Function declarators must be declared with the appropriate type information, including a return type and parameter list. If type information is not properly specified in a function declarator, the compiler cannot properly check function type information. When using standard library calls, the easiest (and preferred) way to obtain function declarators with appropriate type information is to include the appropriate header file.

Attempting to compile a program with a function declarator that does not include the appropriate type information typically generates a warning but does not prevent program compilation. These warnings should be resolved. (See [MSC00-C. Compile cleanly at high warning levels](#).)

Noncompliant Code Example (Non-Prototype-Format Declarators)

This noncompliant code example uses the *identifier-list* form for parameter declarations:

```
int max(a, b)
int a, b;
{
    return a > b ? a : b;
}
```

Subclause 6.11.7 of the C Standard [ISO/IEC 9899:2011](#) states that "the use of function definitions with separate parameter identifier and declaration lists [not prototype-format parameter type and identifier declarators] is an obsolescent feature."

Compliant Solution (Non-Prototype-Format Declarators)

In this compliant solution, `int` is the type specifier, `max(int a, int b)` is the function declarator, and the block within the curly braces is the function body:

```
int max(int a, int b) {
    return a > b ? a : b;
}
```

Noncompliant Code Example [Function Prototypes]

Declaring a function without any prototype forces the compiler to assume that the correct number and type of parameters have been supplied to a function. This practice can result in unintended and [undefined behavior](#).

In this noncompliant code example, the definition of `func()` in `file_a.c` expects three parameters but is supplied only two:

```
/* file_a.c source file */
int func(int one, int two, int three) {
    printf("%d %d %d", one, two, three);
    return 1;
}
```

However, because there is no prototype for `func()` in `file_b.c`, the compiler assumes that the correct number of arguments has been supplied and uses the next value on the program stack as the missing third argument:

```
/* file_b.c source file */
func(1, 2);
```

C99 eliminated implicit function declarations from the C language. However, many compilers still allow the compilation of programs containing implicitly declared functions, although they may issue a warning message. These warnings should be resolved. (See [MSC00-C. Compile cleanly at high warning levels](#).)

Compliant Solution [Function Prototypes]

This compliant solution correctly includes the function prototype for `func()` in the compilation unit in which it is invoked, and the function invocation has been corrected to pass the right number of arguments:

```
/* file_b.c source file */
int func(int, int, int);
func(1, 2, 3);
```

Noncompliant Code Example [Function Pointers]

If a function pointer refers to an incompatible function, invoking that function via the pointer may corrupt the process stack. As a result, unexpected data may be accessed by the called function.

In this noncompliant code example, the function pointer `fn_ptr` refers to the function `add()`, which accepts three integer arguments. However, `fn_ptr` is specified to accept two integer arguments. Setting `fn_ptr` to refer to `add()` results in unexpected program behavior. This example also violates [EXP37-C. Call functions with the correct number and type of arguments](#):

```
int add(int x, int y, int z) {
    return x + y + z;
}

int main(int argc, char *argv[]) {
    int (*fn_ptr)(int, int);
    int res;
    fn_ptr = add;
    res = fn_ptr(2, 3); /* Incorrect */
    /* ... */
    return 0;
}
```

Compliant Solution [Function Pointers]

To correct this example, the declaration of `fn_ptr` is changed to accept three arguments:

```
int add(int x, int y, int z) {
    return x + y + z;
}

int main(int argc, char *argv[]) {
    int (*fn_ptr)(int, int, int);
    int res;
    fn_ptr = add;
    res = fn_ptr(2, 3, 4);
    /* ... */
    return 0;
}
```

Risk Assessment

Failing to include type information for function declarators can result in [unexpected](#) or unintended program behavior.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL07-C	Low	Unlikely	Low	P3	L3

Related Guidelines

ISO/IEC TR 24772:2013	Type System [IHN] Subprogram Signature Mismatch [OTR]
ISO/IEC TS 17961	Using a tainted value as an argument to an unprototyped function pointer [taintnoproto]
MISRA C:2012	Rule 8.2 (required)

Bibliography

[ISO/IEC 9899:2011]	Subclause 6.11.7, "Function Definitions"
[Spinellis 2006]	Section 2.6.1, "Incorrect Routine or Arguments"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/LoAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	<code>msg_for_function_pointer(['klass', 'node'])</code>
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low
reported_messages	If provided, only messages of these types are reported.	513
reported_severities	List of severities to display.	('error', 'warning', 'remark')
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

Possible Messages

Name	Message
implicit_function_return_type	Function shall have explicit return type
implicit_int	Type shall be explicitly stated
missing_parameter_type	Functions shall have a prototype declaration
parameterless_func_without_void_param	Function with no parameters shall be declared with <code>(void)</code>
reference_to_missing_function_prototype	Referenced function needs prototype declaration
reference_to_undeclared_function	Referenced function needs a declaration

CertC-DCL09

Declare functions that return `errno` with a return type of `errno_t`.

Input: IR

Source languages: C, C++

Details

When developing new code, declare functions that return `errno` with a return type of `errno_t`. Many existing functions that return `errno` are declared as returning a value of type `int`. It is semantically unclear by inspecting the function declaration or prototype if these functions return an error status or a value or, worse, some combination of the two. (See [ERR02-C. Avoid in-band error indicators](#).)

C11 Annex K introduced the new type `errno_t` that is defined to be type `int` in `errno.h` and elsewhere. Many of the functions defined in C11 Annex K return values of this type. The `errno_t` type should be used as the type of an object that may contain only values that might be found in `errno`. For example, a function that returns the value of `errno` should be declared as having the return type `errno_t`.

This recommendation depends on C11 Annex K being implemented. The following code can be added to remove this dependency:

```
#ifndef __STDC_LIB_EXT1__
    typedef int errno_t;
#endif
```

Noncompliant Code Example

This noncompliant code example shows a function called `opener()` that returns `errno` error codes. However, the function is declared as returning an `int`. Consequently, the meaning of the return value is not readily apparent.

```
#include <errno.h>
#include <stdio.h>

enum { NO_FILE_POS_VALUES = 3 };

int opener(
    FILE *file,
    size_t *width,
    size_t *height,
    size_t *data_offset
) {
    size_t file_w;
    size_t file_h;
    size_t file_o;
    fpos_t offset;

    if (file == NULL) { return EINVAL; }
    errno = 0;
    if (fgetpos(file, &offset) != 0) { return errno; }
    if (fscanf(file, "%zu %zu %zu", &file_w, &file_h, &file_o)
        != NO_FILE_POS_VALUES) {
        return -1;
    }

    errno = 0;
    if (fsetpos(file, &offset) != 0) { return errno; }

    if (width != NULL) { *width = file_w; }
    if (height != NULL) { *height = file_h; }
    if (data_offset != NULL) { *data_offset = file_o; }

    return 0;
}
```

This noncompliant code example nevertheless complies with [ERR30-C. Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure](#).

Compliant Solution (POSIX)

In this compliant solution, the `opener()` function returns a value of type `errno_t`, providing a clear indication that this function returns an error code:

```
#define __STDC_WANT_LIB_EXT1__ 1

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

enum { NO_FILE_POS_VALUES = 3 };

errno_t opener(
    FILE *file,
    size_t *width,
    size_t *height,
    size_t *data_offset
) {
    size_t file_w;
    size_t file_h;
    size_t file_o;
    fpos_t offset;

    if (NULL == file) { return EINVAL; }
    errno = 0;
    if (fgetpos(file, &offset) != 0) { return errno; }
    if (fscanf(file, "%zu %zu %zu", &file_w, &file_h, &file_o)
        != NO_FILE_POS_VALUES) {
        return EIO;
    }

    errno = 0;
    if (fsetpos(file, &offset) != 0) { return errno; }

    if (width != NULL) { *width = file_w; }
    if (height != NULL) { *height = file_h; }
    if (data_offset != NULL) { *data_offset = file_o; }

    return 0;
}
```

This compliant solution is categorized as a POSIX solution because it returns `EINVAL` and `EIO`, which are defined by POSIX (IEEE Std 1003.1, 2013 Edition) but not by the C Standard.

Failing to test for error conditions can lead to [vulnerabilities](#) of varying severity. Declaring functions that return an `errno` with a return type of `errno_t` will not eliminate this problem but may reduce errors caused by programmers' misunderstanding the purpose of a return value.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL09-C	Low	Unlikely	Low	P3	L3

Related Guidelines

SEI CERT C++ Coding Standard	DCL09-CPP. Declare functions that return errno with a return type of errno_t
ISO/IEC TR 24772:2013	Ignored Error Status and Unhandled Exceptions [OYB]

Bibliography

[\[IEEE Std 1003.1:2013\]](#)

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/QgAy>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
return_type_should_be_errno_t	Declare functions that return errno with a return type of errno_t

CertC-DCL11

Understand the type issues associated with variadic functions.

Input: IR

Source languages: C, C++

Details

The variable parameters of a variadic function - that is, those that correspond with the position of the ellipsis - are interpreted by the `va_arg()` macro. The `va_arg()` macro is used to extract the next argument from an initialized argument list within the body of a variadic function implementation. The size of each parameter is determined by the specified type. If the type is inconsistent with the corresponding argument, the behavior is [undefined](#) and may result in misinterpreted data or an alignment error (see [EXP36-C. Do not cast pointers into more strictly aligned pointer types](#)).

The variable arguments to a variadic function are not checked for type by the compiler. As a result, the programmer is responsible for ensuring that they are compatible with the corresponding parameter after the default argument promotions:

- Integer arguments of types ranked lower than `int` are promoted to `int` if `int` can hold all the values of that type; otherwise, they are promoted to `unsigned int` (the *integer promotions*).
- Arguments of type `float` are promoted to `double`.

Noncompliant Code Example (Type Interpretation Error)

The C `printf()` function is implemented as a variadic function. This noncompliant code example swaps its null-terminated byte string and integer parameters with respect to how they are specified in the format string. Consequently, the integer is interpreted as a pointer to a null-terminated byte string and dereferenced, which will likely cause the program to [abnormally terminate](#). Note that the `error_message` pointer is likewise interpreted as an integer.

```
const char *error_msg = "Error occurred";
/* ... */
printf("%s:%d", 15, error_msg);
```

Compliant Solution (Type Interpretation Error)

This compliant solution modifies the format string so that the conversion specifiers correspond to the arguments:

```
const char *error_msg = "Error occurred";
/* ... */
printf("%d:%s", 15, error_msg);
```

As shown, care must be taken to ensure that the arguments passed to a format string function match up with the supplied format string.

Noncompliant Code Example {Type Alignment Error}

In this noncompliant code example, a type `long long` integer is incorrectly parsed by the `printf()` function with a `%d` specifier. This code may result in data truncation or misrepresentation when the value is extracted from the argument list.

```
long long a = 1;
const char msg[] = "Default message";
/* ... */
printf("%d %s", a, msg);
```

Because a `long long` was not interpreted, if the `long long` uses more bytes for storage, the subsequent format specifier `%s` is unexpectedly offset, causing unknown data to be used instead of the pointer to the message.

Compliant Solution {Type Alignment Error}

This compliant solution adds the length modifier `ll` to the `%d` format specifier so that the variadic function parser for `printf()` extracts the correct number of bytes from the variable argument list for the `long long` argument:

```
long long a = 1;
const char msg[] = "Default message";
/* ... */
printf("%lld %s", a, msg);
```

Noncompliant Code Example {NULL}

Because the C Standard allows `NULL` to be either an integer constant or a pointer constant, any architecture in which `int` is not the same size as a pointer might present a particular [vulnerability](#) with variadic functions. If `NULL` is defined as an `int` on such a platform, then `sizeof(NULL) != sizeof(void *)`, so variadic functions that accept an argument of pointer type will not correctly promote `NULL` to the correct size. Consequently, the following code will have [undefined behavior](#):

```
char* string = NULL;
printf("%s %d\n", string, 1);
```

On a system with 32-bit `int` and 64-bit pointers, `printf()` may interpret the `NULL` as high-order bits of the pointer and the third argument `1` as the low-order bits of the pointer. In this case, `printf()` will print a pointer with the value `0x00000001` and then attempt to read an additional argument for the `%d` conversion specifier, which was not provided.

Compliant Solution {NULL}

This compliant solution avoids sending `NULL` to `printf()`:

```
char* string = NULL;
printf("%s %d\n", (string ? string : "null"), 1);
```

Risk Assessment

Inconsistent typing in variadic functions can result in [abnormal program termination](#) or unintended information disclosure.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL11-C	High	Probable	High	P6	L2

Related Guidelines

ISO/IEC TR 24772:2013	Type System [IHN] Subprogram Signature Mismatch [OTR]
MISRA C:2012	Rule 17.1 (required)

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [https://www.securecoding.cert.org/confluence/x/lwA_1], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
allow_extra_args	Whether to allow additional arguments that are not used by the format string.	False
allow_gnu_extensions	Whether to allow the GNU extensions to format specifications.	True
allow_unknown_specs	Whether to allow unknown format specifications. It may be necessary to set this option when using implementation-specific extensions. Arguments are not checked when the format string contains unknown format specifications.	False
functions	A dictionary mapping the names of the functions to check, to a triple {function_kind, fmt_param_index, arg_start_index} where function_kind is either 'printf' or 'scanf', fmt_param_index is the index of the format-string parameter, and arg_start_index is the index of the first variadic argument.	dict{...}
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
arg_type_mismatch	{} expects argument of type '{}', but argument {} has type '{}'
buffer_too_small	{} may write up to {} characters to buffer of size {}. (disabled)
invalid_conversion	Invalid or non-standard conversion specification
matching_arg_expected	{} expects a matching '{}' argument
precision_for_conversion	Precision must not be used with %{} conversion specifier
too_many_args	Too many arguments for format.
unknown_buffer_size	Potential buffer overflow: {} used with buffer of unknown size. (disabled)
unlimited_read	Potential buffer overflow: {} has no limit on amount of characters read. (disabled)
unsupported_assignment_suppression	%n does not support assignment suppression
unsupported_field_width	%n does not support field width
unsupported_flags	%n does not support flags
unsupported_flags_modifiers	Cannot use any flags or modifiers with '%%'
unsupported_hash	%{} does not support the '#' flag
unsupported_i_flag	%{} does not support the 'l' flag
unsupported_length_modifier	%{} does not support the '{}' length modifier
unsupported_tick	%{} does not support the "" flag
unsupported_zero	%{} does not support the '0' flag

CertC-DCL12

Implement abstract data types using opaque types.

Input: IR

Source languages: C, C++

Details

Abstract data types are not restricted to object-oriented languages such as C++ and Java. They should be created and used in C language programs as well. Abstract data types are most effective when used with private (opaque) data types and information hiding.

Noncompliant Code Example

This noncompliant code example is based on the managed string library developed by CERT [Burch 2006]. In this example, the managed string type and the

functions that operate on this type are defined in the `string_mx.h` header file as follows:

```
struct string_mx {
    size_t size;
    size_t maxsize;
    unsigned char strtype;
    char *cstr;
};

typedef struct string_mx string_mx;

/* Function declarations */
extern errno_t strcpy_m(string_mx *s1, const string_mx *s2);
extern errno_t strcat_m(string_mx *s1, const string_mx *s2);
/* ... */
```

The implementation of the `string_mx` type is fully visible to the user of the data type after including the `string_mx.h` file. Programmers are consequently more likely to directly manipulate the fields within the structure, violating the software engineering principles of information hiding and data encapsulation and increasing the probability of developing incorrect or nonportable code.

Compliant Solution

This compliant solution reimplements the `string_mx` type as a private type, hiding the implementation of the data type from the user of the managed string library. To accomplish this, the developer of the private data type creates two header files: an external `string_mx.h` header file that is included by the user of the data type and an internal file that is included only in files that implement the managed string abstract data type.

In the external `string_mx.h` file, the `string_mx` type is defined to be an instance of `struct string_mx`, which in turn is declared as an [incomplete type](#):

```
struct string_mx;
typedef struct string_mx string_mx;

/* Function declarations */
extern errno_t strcpy_m(string_mx *s1, const string_mx *s2);
extern errno_t strcat_m(string_mx *s1, const string_mx *s2);
/* ... */
```

In the internal header file, `struct string_mx` is fully defined but not visible to a user of the data abstraction:

```
struct string_mx {
    size_t size;
    size_t maxsize;
    unsigned char strtype;
    char *cstr;
};
```

Modules that implement the abstract data type include both the external and internal definitions, whereas users of the data abstraction include only the external `string_mx.h` file. This allows the implementation of the `string_mx` data type to remain private.

Risk Assessment

The use of opaque abstract data types, though not essential to secure programming, can significantly reduce the number of defects and [vulnerabilities](#) introduced in code, particularly during ongoing maintenance.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL12-C	Low	Unlikely	High	P1	L3

Related Guidelines

MISRA C:2012	Directive 4.8 (advisory)
------------------------------	--------------------------

Bibliography

[\[Burch 2006\]](#)

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/UYBS>], Copyright [C] 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	1
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
composite_can_be_opaque	Implementation of composite type should be hidden in unit {}

CertC-DCL13

Declare function parameters that are pointers to values not changed by the function as `const`.

Input: IR

Source languages: C, C++

Details

Declaring function parameters `const` indicates that the function promises not to change these values.

In C, function arguments are passed by value rather than by reference. Although a function may change the values passed in, these changed values are discarded once the function returns. For this reason, many programmers assume a function will not change its arguments and that declaring the function's parameters as `const` is unnecessary.

```
void foo(int x) {  
    x = 3; /* Visible only in the function */  
    /* ... */  
}
```

Pointers behave in a similar fashion. A function may change a pointer to reference a different object, or `NULL`, yet that change is discarded once the function exits. Consequently, declaring a pointer as `const` is unnecessary.

```
void foo(int *x) {  
    x = NULL; /* Visible only in the function */  
    /* ... */  
}
```

Noncompliant Code Example

Unlike passed-by-value arguments and pointers, pointed-to values are a concern. A function may modify a value referenced by a pointer argument, leading to a [side effect](#) that persists even after the function exits. Modification of the pointed-to value is not diagnosed by the compiler, which assumes this behavior was intended.

```
void foo(int *x) {  
    if (x != NULL) {  
        *x = 3; /* Visible outside function */  
    }  
    /* ... */  
}
```

If the function parameter is `const`-qualified, any attempt to modify the pointed-to value should cause the compiler to issue a diagnostic message.

```
void foo(const int *x) {  
    if (x != NULL) {  
        *x = 3; /* Compiler should generate diagnostic message */  
    }  
    /* ... */  
}
```

As a result, the `const` violation must be resolved before the code can be compiled without a diagnostic message being issued.

Compliant Solution

This compliant solution addresses the `const` violation by not modifying the constant argument:

```
void foo(const int * x) {  
    if (x != NULL) {  
        printf("Value is %d\n", *x);  
    }  
    /* ... */  
}
```

Noncompliant Code Example

This noncompliant code example defines a fictional version of the standard `strcat()` function called `strcat_nc()`. This function differs from `strcat()` in that the second argument is not `const`-qualified.

```
char *strcat_nc(char *s1, char *s2);  
  
char *c_str1 = "c_str1";  
const char *c_str2 = "c_str2";  
char c_str3[9] = "c_str3";  
const char c_str4[9] = "c_str4";  
  
strcat_nc(c_str3, c_str2); /* Compiler warns that c_str2 is const */  
strcat_nc(c_str1, c_str3); /* Attempts to overwrite string literal! */  
strcat_nc(c_str4, c_str3); /* Compiler warns that c_str4 is const */
```

The function behaves the same as `strcat()`, but the compiler generates warnings in incorrect locations and fails to generate them in correct locations.

In the first `strcat_nc()` call, the compiler generates a warning about attempting to cast away `const` on `c_str2` because `strcat_nc()` does not modify its second argument yet fails to declare it `const`.

In the second `strcat_nc()` call, the compiler compiles the code with no warnings, but the resulting code will attempt to modify the "`c_str1`" literal. This violates [STR05-C. Use pointers to const when referring to string literals](#) and [STR30-C. Do not attempt to modify string literals](#).

In the final `strcat_nc()` call, the compiler generates a warning about attempting to cast away `const` on `c_str4`, which is a valid warning.

Compliant Solution

This compliant solution uses the prototype for the `strcat()` from C90. Although the `restrict` type qualifier did not exist in C90, `const` did. In general, function parameters should be declared in a manner consistent with the semantics of the function. In the case of `strcat()`, the initial argument can be changed by the function, but the second argument cannot.

```
char *strcat(char *s1, const char *s2);

char *c_str1 = "c_str1";
const char *c_str2 = "c_str2";
char c_str3[9] = "c_str3";
const char c_str4[9] = "c_str4";

strcat(c_str3, c_str2);

/* Args reversed to prevent overwriting string literal */
strcat(c_str3, c_str1);
strcat(c_str4, c_str3); /* Compiler warns that c_str4 is const */
```

The `const`-qualification of the second argument, `s2`, eliminates the spurious warning in the initial invocation but maintains the valid warning on the final invocation in which a `const`-qualified object is passed as the first argument (which can change). Finally, the middle `strcat()` invocation is now valid because `c_str3` is a valid destination string and may be safely modified.

Risk Assessment

Failing to declare an unchanging value `const` prohibits the function from working with values already cast as `const`. This problem can be sidestepped by type casting away the `const`, but doing so violates [EXP05-C. Do not cast away a const qualification](#).

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL13-C	Low	Unlikely	Low	P3	L3

Related Guidelines

SEI CERT C++ Coding Standard	DCL13-CPP. Declare function parameters that are pointers to values not changed by the function as const
ISO/IEC TR 24772:2013	Passing Parameters and Return Values [CSJ]
MISRA C:2012	Rule 8.13 (advisory)

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/p4Lu>], Copyright [C] 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
check_all_variables	Whether non-parameter variables should be checked as well.	False
ignore_forwarding_references	Whether to allow C++11 forwarding references (universal references) even if all template instances would also work with a const lvalue reference.	True
ignore_move_constructors	Whether to allow non-const rvalue references on constructors.	True
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	const_argument(['klass', 'message', 'argument_type'])
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low
report_overrides	Whether to report violations on overrides. For virtual methods, this rule will report a violation only if the parameter can be marked const in all related methods in the inheritance hierarchy. Thus, the same violation is reported for each method in the inheritance hierarchy. You can set this option to False to report such violations only for the base methods.	True
reported_messages	If provided, only messages of these types are reported.	[167, 137]
reported_severities	List of severities to display.	('error', 'warning', 'remark')
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

Possible Messages

Name	Message
parameter_can_point_to_const	{ } can be declared as pointer/reference to const.

CertC-DCL15

Declare file-scope objects or functions that do not need external linkage as static.

Input: IR

Source languages: C, C++

Details

If a file-scope object or a function does not need to be visible outside of the file, it should be hidden by being declared as `static`. This practice creates more modular code and limits pollution of the global name space.

Subclause 6.2.2 of the C Standard [[ISO/IEC 9899:2011](#)] states:

If the declaration of a file scope identifier for an object or a function contains the storage-class specifier `static`, the identifier has internal linkage.

and

If the declaration of an identifier for an object has file scope and no storage-class specifier, its linkage is external.

Noncompliant Code Example

This noncompliant code example includes a `helper()` function that is implicitly declared to have external linkage:

```
enum { MAX = 100 };

int helper(int i) {
    /* Perform some computation based on i */
}

int main(void) {
    size_t i;
    int out[MAX];

    for (i = 0; i < MAX; i++) {
```

```

        out[i] = helper(i);
    }
    /* ... */
}

```

Compliant Solution

This compliant solution declares `helper()` to have internal linkage, thereby preventing external functions from using it:

```

enum {MAX = 100};

static int helper(int i) {
    /* Perform some computation based on i */
}

int main(void) {
    size_t i;
    int out[MAX];

    for (i = 0; i < MAX; i++) {
        out[i] = helper(i);
    }

    /* ... */
}

```

Risk Assessment

Allowing too many objects to have external linkage can use up descriptive identifiers, leading to more complicated identifiers, violations of abstraction models, and possible name conflicts with libraries. If the compilation unit implements a data abstraction, it may also expose invocations of private functions from outside the abstraction.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL15-C	Low	Unlikely	Low	P3	L3

Related Guidelines

SEI CERT C++ Coding Standard	DCL15-CPP. Declare file-scope objects or functions that do not need external linkage in an unnamed namespace
MISRA C:2012	Rule 8.7 (advisory) Rule 8.8 (required)

Bibliography

ISO/IEC 9899:2011	Subclause 6.2.2, "Linkages of Identifiers"
-----------------------------------	--

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/BoMRAQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
allow_moving_to_other_primary_file	When a variable is only used in a single file X, but currently implemented in a different file Y, this controls whether the check suggests to move it into X and make it static there.	False
exclude_dllexport	If True, no suggestions will be produced to make functions marked as <code>dllexport</code> or <code>dllimport</code> static in a primary file.	True
exclude_function_locals	If True, variables that could even be function-local are not reported.	True
exclude_undefined	Whether only-declared symbols should be reported as well	True
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low
template_args_can_be_static	Whether functions whose address is used as template argument can be made static (some compilers, like Microsoft's, don't allow it then).	False

Possible Messages

Name	Message
function_file_static	{ } can be declared static in primary file.
var_file_static	{ } can be declared static in primary file.

CertC-DCL16

Use "L", not "l", to indicate a long value.

Input: IR

Source languages: C, C++

Details

Lowercase letter *l* (ell) can easily be confused with the digit *1* (one). This can be particularly confusing when indicating that an integer literal constant is a long value. This recommendation is similar to [DCL02-C. Use visually distinct identifiers](#).

Likewise, you should use uppercase *LL* rather than lowercase *ll* when indicating that an integer literal constant is a `long long` value.

Noncompliant Code Example

This noncompliant example highlights the result of adding an integer and a long value even though it appears that two integers `1111` are being added:

```
printf("Sum is %ld\n", 1111 + 1111);
```

Compliant Solution

The compliant solution improvises by using an uppercase *L* instead of lowercase *l* to disambiguate the visual appearance:

```
printf("Sum is %ld\n", 1111 + 1111L);
```

Risk Assessment

Confusing a lowercase letter *l* (ell) with a digit *1* (one) when indicating that an integer denotation is a `long` value could lead to an incorrect value being written into code.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL16-C	Low	Unlikely	Low	P3	L3

Related Guidelines

SEI CERT C++ Coding Standard	DCL16-CPP. Use "L," not "l," to indicate a long value
MISRA C:2012	Rule 7.3 (required)

Bibliography

[Lockheed Martin 2005]	AV Rule 14, Literal suffixes shall use uppercase rather than lowercase letters
--	--

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/koAtAQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
lowercase_l_suffix	Lowercase "l" should not be used in a literal suffix

CertC-DCL18

Do not begin integer constants with 0 when specifying a decimal value.

Input: IR

Source languages: C, C++

Details

The C Standard defines octal constants as a 0 followed by octal digits [0 1 2 3 4 5 6 7]. Programming errors can occur when decimal values are mistakenly specified as octal constants.

Noncompliant Code Example

In this noncompliant code example, a decimal constant is mistakenly prefaced with zeros so that all the constants are a fixed length:

```
i_array[0] = 2719;
i_array[1] = 4435;
i_array[2] = 0042;
```

Although it may appear that `i_array[2]` is assigned the decimal value 42, it is actually assigned the decimal value 34.

Compliant Solution

To avoid using wrong values and to make the code more readable, do not preface constants with zeroes if the value is meant to be decimal:

```
i_array[0] = 2719;
i_array[1] = 4435;
i_array[2] =    42;
```

Risk Assessment

Misrepresenting decimal values as octal can lead to incorrect comparisons and assignments.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL18-C	Low	Unlikely	Low	P3	L3

Related Guidelines

MISRA C:2012	Rule 7.1 (required)
------------------------------	---------------------

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [https://www.securecoding.cert.org/confluence/x/_QC7AQ], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
octal_literal	Use of octal literal.

CertC-DCL19

Minimize the scope of variables and functions.

Input: IR

Source languages: C, C++

Details

Variables and functions should be declared in the minimum scope from which all references to the identifier are still possible.

When a larger scope than necessary is used, code becomes less readable, harder to maintain, and more likely to reference unintended variables (see [DCL01-C. Do not reuse variable names in subscopes](#)).

Noncompliant Code Example

In this noncompliant code example, the function `counter()` increments the global variable `count` and then returns immediately if this variable exceeds a maximum value:

```
unsigned int count = 0;

void counter() {
    if (count++ > MAX_COUNT) return;
    /* ... */
}
```

Assuming that the variable `count` is only accessed from this function, this example is noncompliant because it does not define `count` within the minimum possible scope.

Compliant Solution

In this compliant solution, the variable `count` is declared within the scope of the `counter()` function as a static variable. The static modifier, when applied to a local variable (one inside of a function), modifies the lifetime (duration) of the variable so that it persists for as long as the program does and does not disappear between invocations of the function.

```
void counter() {
    static unsigned int count = 0;
    if (count++ > MAX_COUNT) return;
    /* ... */
}
```

The keyword `static` also prevents reinitialization of the variable.

Noncompliant Code Example

The counter variable `i` is declared outside of the `for` loop, which goes against this recommendation because it is not declared in the block in which it is used. If this code were reused with another index variable `j`, but there was a previously declared variable `i`, the loop could iterate over the wrong variable.

```
size_t i = 0;

for (i=0; i < 10; i++) {
    /* Perform operations */
}
```

Compliant Solution

Complying with this recommendation requires that you declare variables where they are used, which improves readability and reusability. In this example, you would declare the loop's index variable `i` within the initialization of the `for` loop. This requirement was recently relaxed in the C Standard.

```
for (size_t i=0; i < 10; i++) {
    /* Perform operations */
}
```

Noncompliant Code Example [Function Declaration]

In this noncompliant code example, the function `f()` is called only from within the function `g()`, which is defined in the same compilation unit. By default, function declarations are *extern*, meaning that these functions are placed in the global symbol table and are available from other compilation units.

```
int f(int i) {
    /* Function definition */
}

int g(int i) {
    int j = f(i);
    /* ... */
}
```

Compliant Solution

In this compliant solution, the function `f()` is declared with internal linkage. This practice limits the scope of the function declaration to the current compilation unit and prevents the function from being included in the external symbol table. It also limits cluttering in the global name space and prevents the function from being accidentally or intentionally invoked from another compilation unit. See [DCL15-C. Declare file-scope objects or functions that do not need external linkage as static](#) for more information.

```
static int f(int i) {
    /* Function definition */
}

int g(int i) {
    int j = f(i);
    /* ... */
}
```

Risk Assessment

Failure to minimize scope could result in less reliable, readable, and reusable code.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL19-C	Low	Unlikely	Medium	P2	L3

Related Guidelines

SEI CERT C++ Coding Standard	DCL07-CPP. Minimize the scope of variables and methods
MISRA C:2012	Rule 8.9 (advisory)

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/DADAAQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
allow_moving_to_other_primary_file	When a variable is only used in a single file X, but currently implemented in a different file Y, this controls whether the check suggests to move it into X and make it static there.	False
check_loop_counter	Whether to report for-loop counters being declared before the loop that could be declared in the for-init part instead from C99 on	True
exclude_dllexport	If True, no suggestions will be produced to make functions marked as dllexport or dllimport static in a primary file.	True
exclude_function_locals	If True, variables that could even be function-local are not reported.	True
exclude_undefined	Whether only-declared symbols should be reported as well	True
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
move_global_const_into_function	Controls suggestions for global constants that could be declared locally in a function.	True
only_check_unit_locals	Whether only global static variables should be checked.	False
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium
template_args_can_be_static	Whether functions whose address is used as template argument can be made static (some compilers, like Microsoft's, don't allow it then).	False

Possible Messages

Name	Message
function_file_static	{ } can be declared static in primary file.
locality_block	{ } can be declared in a more local scope.
locality_function	Global { } can be declared inside function.
locality_loop_init	{ } can be declared in the for-loop's initialization.
var_file_static	{ } can be declared static in primary file.

CertC-DCL20

Explicitly specify void when a function accepts no arguments.

Input: IR

Source languages: C, C++

Details

According to the C Standard, subclause 6.7.6.3, paragraph 14 [[ISO/IEC 9899:2011](#)],

An identifier list declares only the identifiers of the parameters of the function. An empty list in a function declarator that is part of a definition of that function specifies that the function has no parameters. The empty list in a function declarator that is not part of a definition of that function specifies that no information about the number or types of the parameters is supplied.

Subclause 6.11.6 states that

The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature.

Consequently, functions that accept no arguments should explicitly declare a `void` parameter in their parameter list. This holds true in both the declaration and definition sections (which should match).

Defining a function with a `void` argument list differs from declaring it with no arguments because, in the latter case, the compiler will not check whether the function is called with parameters at all [[TIGCC, void usage](#)]. Consequently, function calling with arbitrary parameters will be accepted without a warning at compile time.

Failure to declare a `void` parameter will result in

- An ambiguous functional interface between the caller and callee.
- Sensitive information outflow.

A similar recommendation deals with parameter type in a more general sense: [DCL07-C. Include the appropriate type information in function declarators](#).

Noncompliant Code Example (Ambiguous Interface)

In this noncompliant code example, the caller calls `foo()` with an argument of 3. The caller expects `foo()` to accept a single `int` argument and to output the argument as part of a longer message. Because `foo()` is declared without the `void` parameter, the compiler will not perform any caller check. It is therefore possible that the caller may not detect the error. In this example, for instance, `foo()` might output the value 3 as expected.

Because no function parameter has the same meaning as an arbitrary parameter, the caller can provide an arbitrary number of arguments to the function.

```
/* In foo.h */
void foo();

/* In foo.c */
void foo() {
    int i = 3;
    printf("i value: %d\n", i);
}

/* In caller.c */
#include "foo.h"

foo(3);
```

Compliant Solution (Ambiguous Interface)

In this compliant solution, `void` is specified explicitly as a parameter in the declaration of `foo`'s prototype:

```
/* In foo.h */
void foo(void);

/* In foo.c */
void foo(void) {
    int i = 3;
    printf("i value: %d\n", i);
}

/* In caller.c */
#include "foo.h"

foo(3);
```

Implementation Details (Ambiguous Interface)

When the compliant solution is used and `foo(3)` is called, the GCC compiler issues the following diagnostic, which alerts the programmer about the misuse of the function interface:

```
error: too many arguments to function "foo"
```

Noncompliant Code Example (Information Outflow)

Another possible vulnerability is the leak of privileged information. In this noncompliant code example, a user with high privileges feeds some secret input to the caller that the caller then passes to `foo()`. Because of the way `foo()` is defined, we might assume there is no way for `foo()` to retrieve information from the caller. However, because the value of `i` is really passed into a stack (before the return address of the caller), a malicious programmer can change the internal implementation and copy the value manually into a less privileged file.

```
/* Compile using gcc4.3.3 */
void foo() {
    /*
     * Use assembly code to retrieve i
     * implicitly from caller
     * and transfer it to a less privileged file.
     */
}

...
```

```
/* Caller */
foo(i); /* i is fed from user input */
```

Compliant Solution [Information Outflow]

```
void foo(void) {
    int i = 3;
    printf("i value: %d\n", i);
}
```

Again, the simplest solution is to explicitly specify `void` as the only parameter.

Risk Assessment

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL20-C	Medium	Probable	Low	P12	L1

Related Guidelines

In C++, `foo()` and `foo(void)` have exactly the same meaning and effect, so this rule doesn't apply to C++. However, `foo(void)` should be declared explicitly instead of `foo()` to distinguish it from `foo(...)`, which accepts an arbitrary number and type of arguments.

MISRA C:2012	Rule 8.2 [required]
------------------------------	---------------------

Bibliography

[ISO/IEC 9899:2011]	Subclause 6.7.6.3, "Function Declarators (including Prototypes)" Subclause 6.11.6, "Function Declarators"
[TIGCC, void usage]	Manual, "C Language Keywords": void

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/9YAzAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
parameterless_func_without_void_param	Function with no parameters shall be declared with (void)

CertC-DCL21

Understand the storage of compound literals.

Input: IR

Source languages: C, C++

Details

Subclause 6.5.2.5 of the C Standard [[ISO/IEC 9899:2011](#)] defines a compound literal as

A postfix expression that consists of a parenthesized type name followed by a brace-enclosed list of initializers. . . . The value of the compound literal is that of an unnamed object initiated by the initializer list.

The storage for this object is either static (if the compound literal occurs at file scope) or automatic (if the compound literal occurs at block scope), and the storage duration is associated with its immediate enclosing block. For example, in the function

```
void func(void) {
    int *ip = (int[4]) {1,2,3,4};
    /* ... */
}
```

following initialization, the `int` pointer `ip` contains the address of an unnamed object of type `int[4]`, allocated on the stack. Once `func` returns, any attempts to access this object will produce [undefined behavior](#).

Note that only one object is created per compound literal - even if the compound literal appears in a loop and has dynamic initializers.

This recommendation is a specific instance of [DCL30-C. Declare objects with appropriate storage durations](#).

Noncompliant Code Example

In this noncompliant code example, the programmer mistakenly assumes that the elements of the `ints` array of the pointer to `int_struct` are assigned the addresses of distinct `int_struct` objects, one for each integer in the range `[0, MAX_INTS - 1]`:

```
#include <stdio.h>

typedef struct int_struct {
    int x;
} int_struct;

#define MAX_INTS 10

int main(void) {
    size_t i;
    int_struct *ints[MAX_INTS];

    for (i = 0; i < MAX_INTS; i++) {
        ints[i] = &(int_struct){i};
    }

    for (i = 0; i < MAX_INTS; i++) {
        printf("%d\n", ints[i]->x);
    }

    return 0;
}
```

However, only one `int_struct` object is created. At each iteration of the first loop, the `x` member of this object is set equal to the current value of the loop counter `i`. Therefore, just before the first loop terminates, the value of the `x` member is `MAX_INTS - 1`.

Because the storage duration of the compound literal is associated with the `for` loop that contains it, dereferencing `ints` in the second loop results in [undefined behavior](#) 9 (Annex J of the C Standard).

Even if the region of memory that contained the compound literal is not written to between loops, the print loop will display the value `MAX_INTS - 1` for `MAX_INTS` lines. This is contrary to the intuitive expected result, which is that the integers 0 through `MAX_INTS - 1` would be printed in order.

Compliant Solution

This compliant solution uses an array of structures rather than an array of pointers. That way, an actual copy of each `int_struct` (rather than a pointer to the object) is stored.

```
#include <stdio.h>

typedef struct int_struct {
    int x;
} int_struct;

#define MAX_INTS 10

int main(void) {
    size_t i;
    int_struct ints[MAX_INTS];

    for (i = 0; i < MAX_INTS; i++) {
        ints[i] = (int_struct){i};
    }

    for (i = 0; i < MAX_INTS; i++) {
        printf("%d\n", ints[i].x);
    }

    return 0;
}
```

Risk Assessment

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
DCL21-C	Low	Unlikely	Medium	P2	L3

Bibliography

[ISO/IEC 9899:2011]	Subclause 6.5.2.5, "Compound Literals"
Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [https://www.securecoding.cert.org/confluence/x/yQCMAG], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.	

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
escaping_compound_literal_address	The address of a compound literal should not escape its block

CertC-DCL23

Guarantee that mutually visible identifiers are unique.

Input: IR

Source languages: C, C++

Details

According to subclause 6.2.7 of the C Standard [[ISO/IEC 9899:2011](#)],

All declarations that refer to the same object or function shall have compatible type; otherwise, the behavior is undefined.

(See also [undefined behavior 15](#) of Annex J.)

Further, according to subclause 6.4.2.1,

Any identifiers that differ in a significant character are different identifiers. If two identifiers differ only in nonsignificant characters, the behavior is undefined.

(See also [undefined behavior 31](#) of Annex J.)

Identifiers in mutually visible scopes must be deemed unique by the compiler to prevent confusion about which variable or function is being referenced.

[Implementations](#) can allow additional nonunique characters to be appended to the end of identifiers, making the identifiers appear unique while actually being indistinguishable.

It is reasonable for scopes that are not visible to each other to have duplicate identifiers. For example, two functions can each have a local variable with the same name because their scopes cannot access each other. But a function's local variable names should be distinct from each other as well as from all static variables declared within the function's file (and from all included header files.)

To guarantee that identifiers are unique, the number of significant characters recognized by the most restrictive compiler used must be determined. This assumption must be documented in the code.

The standard defines the following minimum requirements:

- 63 significant initial characters in an internal identifier or a macro name. (Each universal character name or extended source character is considered a single character.)
- 31 significant initial characters in an external identifier. (Each universal character name specifying a short identifier of 0000FFFF or less is considered 6 characters; each universal character name specifying a short identifier of 00010000 or more is considered 10 characters; and each extended source character, if any exist, is considered the same number of characters as the corresponding universal character name.)

Restriction of the significance of an external name to fewer than 255 characters in the standard (considering each universal character name or extended source character as a single character) is an obsolescent feature that is a concession to existing implementations. As a result, it is not necessary to comply with this restriction as long as the identifiers are unique and the assumptions concerning the number of significant characters are documented.

Noncompliant Code Example (Source Character Set)

On implementations that support only the minimum requirements for significant characters required by the standard, this code example is noncompliant because the first 31 characters of the external identifiers are identical:

```
extern int *global_symbol_definition_lookup_table_a;
extern int *global_symbol_definition_lookup_table_b;
```

Compliant Solution (Source Character Set)

In a compliant solution, the significant characters in each identifier must differ:

```
extern int *a_global_symbol_definition_lookup_table;
extern int *b_global_symbol_definition_lookup_table;
```

Noncompliant Code Example (Universal Character Names)

In this noncompliant code example, both external identifiers consist of four universal character names. Because the first three universal character names of each identifier are identical, both identify the same integer array on implementations that support only the minimum requirements for significant characters required by the standard:

```
extern int *\u000010401\u000010401\u000010401\u000010401;
extern int *\u000010401\u000010401\u000010401\u000010402;
```

Compliant Solution (Universal Character Names)

For portability, the first three universal character name combinations used in an identifier must be unique:

```
extern int *\u000010401\u000010401\u000010401\u000010401;
extern int *\u000010402\u000010401\u000010401\u000010401;
```

Risk Assessment

Nonunique identifiers can lead to abnormal program termination, denial-of-service attacks, or unintended information disclosure.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL23-C	Medium	Unlikely	Low	P6	L2

Related Guidelines

ISO/IEC TR 24772:2013	Choice of Clear Names [NAI] Identifier Name Reuse [YOW]
MISRA C:2012	Rule 5.1 (required) Rule 5.2 (required) Rule 5.3 (required) Rule 5.4 (required) Rule 5.5 (required)

Bibliography

[ISO/IEC 9899:2011]	Subclause 6.2.7, "Compatible Type and Composite Type" Subclause 6.4.1, "Keywords"
-------------------------------------	--

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/QAU>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
maxlen		31
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low
report_external_identifiers	Whether external identifiers should be compared to each other.	True
report_internal_identifiers	Whether internal identifiers should be compared to each other (including macros).	True
report_short_identifiers	If True, identifiers shorter than maxlen are considered as well.	True

Possible Messages

Name	Message
external_identifiers_not_distinct	External identifiers not distinct.
external_identifiers_sharing	External identifiers sharing first {} characters.
internal_identifiers_not_distinct	Internal identifiers not distinct.
internal_identifiers_sharing	Internal identifiers sharing first {} characters.

CertC-DCL30

Declare objects with appropriate storage durations.

Input: IR

Source languages: C, C++

Details

Every object has a storage duration that determines its lifetime: *static*, *thread*, *automatic*, or *allocated*.

According to the C Standard, 6.2.4, paragraph 2 [[ISO/IEC 9899:2011](#)],

The lifetime of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address, and retains its last-stored value throughout its lifetime. If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime.

Do not attempt to access an object outside of its lifetime. Attempting to do so is [undefined behavior](#) and can lead to an exploitable [vulnerability](#). (See also [undefined behavior 9](#) in the C Standard, Annex J.)

Noncompliant Code Example [Differing Storage Durations]

In this noncompliant code example, the address of the variable `c_str` with automatic storage duration is assigned to the variable `p`, which has static storage duration. The assignment itself is valid, but it is invalid for `c_str` to go out of scope while `p` holds its address, as happens at the end of `dont_do_this()`.

```
#include <stdio.h>

const char *p;
void dont_do_this(void) {
    const char c_str[] = "This will change";
    p = c_str; /* Dangerous */
}

void innocuous(void) {
    printf("%s\n", p);
}

int main(void) {
    dont_do_this();
    innocuous();
    return 0;
}
```

Compliant Solution [Same Storage Durations]

In this compliant solution, `p` is declared with the same storage duration as `c_str`, preventing `p` from taking on an [indeterminate value](#) outside of `this_is_OK()`:

```
void this_is_OK(void) {
    const char c_str[] = "Everything OK";
    const char *p = c_str;
    /* ... */
}
/* p is inaccessible outside the scope of string c_str */
```

Alternatively, both `p` and `c_str` could be declared with static storage duration.

Compliant Solution [Differing Storage Durations]

If it is necessary for `p` to be defined with static storage duration but `c_str` with a more limited duration, then `p` can be set to `NULL` before `c_str` is destroyed. This practice prevents `p` from taking on an [indeterminate value](#), although any references to `p` must check for `NULL`.

```
const char *p;
void is_this_OK(void) {
    const char c_str[] = "Everything OK?";
    p = c_str;
    /* ... */
    p = NULL;
}
```

Noncompliant Code Example [Return Values]

In this noncompliant code sample, the function `init_array()` returns a pointer to a character array with automatic storage duration, which is accessible to the caller:

```
char *init_array(void) {
    char array[10];
    /* Initialize array */
    return array;
}
```

Some compilers generate a diagnostic message when a pointer to an object with automatic storage duration is returned from a function, as in this example. Programmers should compile code at high warning levels and resolve any diagnostic messages. (See [MSC00-C. Compile cleanly at high warning levels](#).)

Compliant Solution [Return Values]

The solution, in this case, depends on the intent of the programmer. If the intent is to modify the value of `array` and have that modification persist outside the

scope of `init_array()`, the desired behavior can be achieved by declaring `array` elsewhere and passing it as an argument to `init_array()`:

```
#include <stddef.h>
void init_array(char *array, size_t len) {
    /* Initialize array */
    return;
}

int main(void) {
    char array[10];
    init_array(array, sizeof(array) / sizeof(array[0]));
    /* ... */
    return 0;
}
```

Noncompliant Code Example (Output Parameter)

In this noncompliant code example, the function `squirrel_away()` stores a pointer to local variable `local` into a location pointed to by function parameter `ptr_param`. Upon the return of `squirrel_away()`, the pointer `ptr_param` points to a variable that has an expired lifetime.

```
void squirrel_away(char **ptr_param) {
    char local[10];
    /* Initialize array */
    *ptr_param = local;
}

void rodent(void) {
    char *ptr;
    squirrel_away(&ptr);
    /* ptr is live but invalid here */
}
```

Compliant Solution (Output Parameter)

In this compliant solution, the variable `local` has static storage duration; consequently, `ptr` can be used to reference the `local` array within the `rodent()` function:

```
char local[10];

void squirrel_away(char **ptr_param) {
    /* Initialize array */
    *ptr_param = local;
}

void rodent(void) {
    char *ptr;
    squirrel_away(&ptr);
    /* ptr is valid in this scope */
}
```

Risk Assessment

Referencing an object outside of its lifetime can result in an attacker being able to execute arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL30-C	High	Probable	High	P6	L2

Related Guidelines

CERT C Secure Coding Standard	MSC00-C. Compile cleanly at high warning levels
SEI CERT C++ Coding Standard	EXP54-CPP. Do not access an object outside of its lifetime
ISO/IEC TR 24772:2013	Dangling References to Stack Frames [DCM]
ISO/IEC TS 17961	Escaping of the address of an automatic object [addrescape]
MISRA C:2012	Rule 18.6 (required)

Bibliography

[Coverity 2007]	
[ISO/IEC 9899:2011]	6.2.4, "Storage Durations of Objects"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki (<https://www.securecoding.cert.org/confluence/x/bQ4>), Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
global_reference_to_local_var	Address of local variable escapes via global variable.
parameter_reference_to_local_var	Address of local variable escapes via parameter.
returning_reference_to_local_var	Return of reference/pointer to local variable.

CertC-DCL31

Declare identifiers before using them.

Input: IR

Source languages: C, C++

Details

The C11 Standard requires type specifiers and forbids implicit function declarations. The C90 Standard allows implicit typing of variables and functions. Consequently, some existing legacy code uses implicit typing. Some C compilers still support legacy code by allowing implicit typing, but it should not be used for new code. Such an [implementation](#) may choose to assume an implicit declaration and continue translation to support existing programs that used this feature.

Noncompliant Code Example (Implicit int)

C no longer allows the absence of type specifiers in a declaration. The C Standard, 6.7.2 [[ISO/IEC 9899:2011](#)], states

At least one type specifier shall be given in the declaration specifiers in each declaration, and in the specifier-qualifier list in each `struct` declaration and type name.

This noncompliant code example omits the type specifier:

```
extern foo;
```

Some C [implementations](#) do not issue a diagnostic for the violation of this constraint. These nonconforming C translators continue to treat such declarations as implying the type `int`.

Compliant Solution (Implicit int)

This compliant solution explicitly includes a type specifier:

```
extern int foo;
```

Noncompliant Code Example (Implicit Function Declaration)

Implicit declaration of functions is not allowed; every function must be explicitly declared before it can be called. In C90, if a function is called without an explicit prototype, the compiler provides an implicit declaration.

The C90 Standard [[ISO/IEC 9899:1990](#)] includes this requirement:

If the expression that precedes the parenthesized argument list in a function call consists solely of an identifier, and if no declaration is visible for this identifier, the identifier is implicitly declared exactly as if, in the innermost block containing the function call, the declaration `extern int identifier();` appeared.

If a function declaration is not visible at the point at which a call to the function is made, C90-compliant platforms assume an implicit declaration of `extern int identifier();`.

This declaration implies that the function may take any number and type of arguments and return an `int`. However, to conform to the current C Standard, programmers must explicitly prototype every function before invoking it. An implementation that conforms to the C Standard may or may not perform implicit function declarations, but C does require a conforming implementation to issue a diagnostic if it encounters an undeclared function being used.

In this noncompliant code example, if `malloc()` is not declared, either explicitly or by including `stdlib.h`, a compiler that conforms only to C90 may implicitly declare `malloc()` as `int malloc()`. If the platform's size of `int` is 32 bits, but the size of pointers is 64 bits, the resulting pointer would likely be truncated as a result of the implicit declaration of `malloc()`, returning a 32-bit integer.

```
#include <stddef.h>
/* #include <stdlib.h> is missing */

int main(void) {
    for (size_t i = 0; i < 100; ++i) {
        /* int malloc() assumed */
        char *ptr = (char *)malloc(0x10000000);
        *ptr = 'a';
    }
    return 0;
}
```

Implementation Details

When compiled with Microsoft Visual Studio 2013 for a 64-bit platform, this noncompliant code example will eventually cause an access violation when dereferencing `ptr` in the loop.

Compliant Solution (Implicit Function Declaration)

This compliant solution declares `malloc()` by including the appropriate header file:

```
#include <stdlib.h>

int main(void) {
    for (size_t i = 0; i < 100; ++i) {
        char *ptr = (char *)malloc(0x10000000);
        *ptr = 'a';
    }
    return 0;
}
```

For more information on function declarations, see [DCL07-C. Include the appropriate type information in function declarators](#).

Noncompliant Code Example (Implicit Return Type)

Do not declare a function with an implicit return type. For example, if a function returns a meaningful integer value, declare it as returning `int`. If it returns no meaningful value, declare it as returning `void`.

```
#include <limits.h>
#include <stdio.h>

foo(void) {
    return UINT_MAX;
}

int main(void) {
    long long int c = foo();
    printf("%lld\n", c);
    return 0;
}
```

Because the compiler assumes that `foo()` returns a value of type `int` for this noncompliant code example, `UINT_MAX` is incorrectly converted to -1.

Compliant Solution (Implicit Return Type)

This compliant solution explicitly defines the return type of `foo()` as `unsigned int`. As a result, the function correctly returns `UINT_MAX`.

```
#include <limits.h>
#include <stdio.h>

unsigned int foo(void) {
    return UINT_MAX;
}

int main(void) {
    long long int c = foo();
    printf("%lld\n", c);
    return 0;
}
```

Risk Assessment

Because implicit declarations lead to less stringent type checking, they can introduce [unexpected](#) and erroneous behavior. Occurrences of an omitted type specifier in existing code are rare, and the consequences are generally minor, perhaps resulting in [abnormal program termination](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL31-C	Low	Unlikely	Low	P3	L3

Related Guidelines

CERT C Secure Coding Standard	DCL07-C. Include the appropriate type information in function declarators
ISO/IEC TR 24772:2013	Subprogram Signature Mismatch [OTR]
MISRA C:2012	Rule 8.1 (required)

Bibliography

[ISO/IEC 9899:1990]	
[ISO/IEC 9899:2011]	Subclause 6.7.2, "Type Specifiers"
[Jones 2008]	

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/tgDl>], Copyright [C] 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
implicit_int	Type shall be explicitly stated
reference_to_missing_function_prototype	Referenced function needs prototype declaration
reference_to_undeclared_function	Referenced function needs a declaration

CertC-DCL36

Do not declare an identifier with conflicting linkage classifications.

Input: IR

Source languages: C, C++

Details

Linkage can make an identifier declared in different scopes or declared multiple times within the same scope refer to the same object or function. Identifiers are classified as *externally linked*, *internally linked*, or *not linked*. These three kinds of linkage have the following characteristics [[Kirch-Prinz 2002](#)]:

- **External linkage:** An identifier with external linkage represents the same object or function throughout the entire program, that is, in all compilation units and libraries belonging to the program. The identifier is available to the linker. When a second declaration of the same identifier with external linkage occurs, the linker associates the identifier with the same object or function.
- **Internal linkage:** An identifier with internal linkage represents the same object or function within a given translation unit. The linker has no information about identifiers with internal linkage. Consequently, these identifiers are internal to the translation unit.
- **No linkage:** If an identifier has no linkage, then any further declaration using the identifier declares something new, such as a new variable or a new type.

According to the C Standard, 6.2.2 [[ISO/IEC 9899:2011](#)], linkage is determined as follows:

If the declaration of a file scope identifier for an object or a function contains the storage class specifier `static`, the identifier has internal linkage.

For an identifier declared with the storage-class specifier `extern` in a scope in which a prior declaration of that identifier is visible, if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

If the declaration of an identifier for a function has no storage-class specifier, its linkage is determined exactly as if it were declared with the storage-class specifier `extern`. If the declaration of an identifier for an object has file scope and no storage-class specifier, its linkage is external.

The following identifiers have no linkage: an identifier declared to be anything other than an object or a function; an identifier declared to be a function parameter; a block scope identifier for an object declared without the storage-class specifier `extern`.

Use of an identifier (within one translation unit) classified as both internally and externally linked is [undefined behavior](#). (See also [undefined behavior 8](#).) A translation unit includes the source file together with its headers and all source files included via the preprocessing directive `#include`.

The following table identifies the linkage assigned to an object that is declared twice in a single translation unit. The column designates the first declaration, and the row designates the redeclaration.

		Second		
		static	No linkage	extern
First	static	Internal	Undefined	Internal
	No linkage	Undefined	No linkage	External
	extern	Undefined	Undefined	External

Noncompliant Code Example

In this noncompliant code example, `i2` and `i5` are defined as having both internal and external linkage. Future use of either identifier results in [undefined behavior](#).

```
int i1 = 10;          /* Definition, external linkage */
static int i2 = 20;   /* Definition, internal linkage */
extern int i3 = 30;   /* Definition, external linkage */
int i4;              /* Tentative definition, external linkage */
static int i5;        /* Tentative definition, internal linkage */

int i1;   /* Valid tentative definition */
int i2;   /* Undefined, linkage disagreement with previous */
int i3;   /* Valid tentative definition */
int i4;   /* Valid tentative definition */
int i5;   /* Undefined, linkage disagreement with previous */

int main(void) {
    /* ... */
    return 0;
}
```

Implementation Details

Microsoft Visual Studio 2013 issues no warnings about this code, even at the highest diagnostic levels.

The GCC compiler generates a fatal diagnostic for the conflicting definitions of `i2` and `i5`.

Compliant Solution

This compliant solution does not include conflicting definitions:

```
int i1 = 10;          /* Definition, external linkage */
static int i2 = 20;   /* Definition, internal linkage */
extern int i3 = 30;   /* Definition, external linkage */
int i4;              /* Tentative definition, external linkage */
static int i5;        /* Tentative definition, internal linkage */

int main(void) {
    /* ... */
    return 0;
}
```

Risk Assessment

Use of an identifier classified as both internally and externally linked is [undefined behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL36-C	Medium	Probable	Medium	P8	L2

Related Guidelines

MISRA C:2012	Rule 8.2 (required) Rule 8.4 (required) Rule 8.8 (required) Rule 17.3 (mandatory)
------------------------------	--

Bibliography

[Banahan 2003]	Section 8.2, "Declarations, Definitions and Accessibility"
[ISO/IEC 9899:2011]	6.2.2, "Linkages of Identifiers"
[Kirch-Prinz 2002]	

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/hoAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
reported_messages	If provided, only messages of these types are reported.	172
reported_severities	List of severities to display.	('error', 'warning', 'remark')
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

CertC-DCL37

Do not declare or define a reserved identifier.

Input: IR

Source languages: C, C++

Details

According to the C Standard, 7.1.3 [[ISO/IEC 9899:2011](#)],

All identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use.

All identifiers that begin with an underscore are always reserved for use as identifiers with file scope in both the ordinary and tag name spaces.

Each macro name in any of the following subclauses (including the future library directions) is reserved for use as specified if any of its associated headers is included, unless explicitly stated otherwise.

All identifiers with external linkage (including future library directions) and `errno` are always reserved for use as identifiers with external linkage.

Each identifier with file scope listed in any of the following subclauses (including the future library directions) is reserved for use as a macro name and as an identifier with file scope in the same name space if any of its associated headers is included.

Additionally, subclause 7.31 defines many other reserved identifiers for future library directions.

No other identifiers are reserved. (The POSIX standard extends the set of identifiers reserved by the C Standard to include an open-ended set of its own. See *Portable Operating System Interface [POSIX^R], Base Specifications, Issue 7, Section 2.2, "The Compilation Environment"* [[IEEE Std 1003.1-2013](#).] The behavior of a program that declares or defines an identifier in a context in which it is reserved or that defines a reserved identifier as a macro name is undefined. (See [undefined behavior 106](#).)

Noncompliant Code Example (Header Guard)

A common, but noncompliant, practice is to choose a reserved name for a macro used in a preprocessor conditional guarding against multiple inclusions of a header file. (See also [PRE06-C. Enclose header files in an inclusion guard](#).) The name may clash with reserved names defined by the implementation of the C standard library in its headers or with reserved names implicitly predefined by the compiler even when no C standard library header is included.

```
#ifndef __MY_HEADER_H__
#define __MY_HEADER_H__

/* Contents of <my_header.h> */

#endif /* __MY_HEADER_H__ */
```

Compliant Solution (Header Guard)

This compliant solution avoids using leading underscores in the name of the header guard:

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

/* Contents of <my_header.h> */

#endif /* MY_HEADER_H */
```

Noncompliant Code Example (File Scope Objects)

In this noncompliant code example, the names of the file scope objects `_max_limit` and `_limit` both begin with an underscore. Because `_max_limit` is static,

this declaration might seem to be impervious to clashes with names defined by the implementation. However, because the header `<stddef.h>` is included to define `size_t`, a potential for a name clash exists. (Note, however, that a [conforming](#) compiler may implicitly declare reserved names regardless of whether any C standard library header is explicitly included.)

In addition, because `_limit` has external linkage, it may clash with a symbol of the same name defined in the language runtime library even if such a symbol is not declared in any header. Consequently, it is not safe to start the name of any file scope identifier with an underscore even if its linkage limits its visibility to a single translation unit.

```
#include <stddef.h>

static const size_t _max_limit = 1024;
size_t _limit = 100;

unsigned int getValue(unsigned int count) {
    return count < _limit ? count : _limit;
}
```

Compliant Solution {File Scope Objects}

In this compliant solution, names of file scope objects do not begin with an underscore:

```
#include <stddef.h>

static const size_t max_limit = 1024;
size_t limit = 100;

unsigned int getValue(unsigned int count) {
    return count < limit ? count : limit;
}
```

Noncompliant Code Example {Reserved Macros}

In this noncompliant code example, because the C standard library header `<inttypes.h>` is specified to include `<stdint.h>`, the name `SIZE_MAX` conflicts with a standard macro of the same name, which is used to denote the upper limit of `size_t`. In addition, although the name `INTFAST16_LIMIT_MAX` is not defined by the C standard library, it is a reserved identifier because it begins with the `INT` prefix and ends with the `_MAX` suffix. (See the C Standard, 7.31.10.)

```
#include <inttypes.h>
#include <stdio.h>

static const int_fast16_t INTFAST16_LIMIT_MAX = 12000;

void print_fast16(int_fast16_t val) {
    enum { SIZE_MAX = 80 };
    char buf[SIZE_MAX];
    if (INTFAST16_LIMIT_MAX < val) {
        sprintf(buf, "The value is too large");
    } else {
        snprintf(buf, SIZE_MAX, "The value is %" PRIdFAST16, val);
    }
}
```

Compliant Solution {Reserved Macros}

This compliant solution avoids redefining reserved names or using reserved prefixes and suffixes:

```
#include <inttypes.h>
#include <stdio.h>

static const int_fast16_t MY_INTFAST16_UPPER_LIMIT = 12000;

void print_fast16(int_fast16_t val) {
    enum { BUFSIZE = 80 };
    char buf[BUFSIZE];
    if (MY_INTFAST16_UPPER_LIMIT < val) {
        sprintf(buf, "The value is too large");
    } else {
        snprintf(buf, BUFSIZE, "The value is %" PRIdFAST16, val);
    }
}
```

Noncompliant Code Example {Identifiers with External Linkage}

In addition to symbols defined as functions in each C standard library header, identifiers with external linkage include `errno` and `math_errhandling`, among others, regardless of whether any of them are masked by a macro of the same name.

This noncompliant example provides definitions for the C standard library functions `malloc()` and `free()`. Although this practice is permitted by many traditional implementations of UNIX (for example, the [Dmalloc](#) library), it is [undefined behavior](#) according to the C Standard. Even on systems that allow replacing `malloc()`, doing so without also replacing `aligned_malloc()`, `calloc()`, and `realloc()` is likely to cause problems.

```
#include <stddef.h>

void *malloc(size_t nbytes) {
    void *ptr;
    /* Allocate storage from own pool and set ptr */
    return ptr;
}

void free(void *ptr) {
    /* Return storage to own pool */
}
```

Compliant Solution (Identifiers with External Linkage)

The compliant, portable solution avoids redefining any C standard library identifiers with external linkage. In addition, it provides definitions for all memory allocation functions:

```
#include <stddef.h>

void *my_malloc(size_t nbytes) {
    void *ptr;
    /* Allocate storage from own pool and set ptr */
    return ptr;
}

void *my_aligned_alloc(size_t alignment, size_t size) {
    void *ptr;
    /* Allocate storage from own pool, align properly, set ptr */
    return ptr;
}

void *my_calloc(size_t nelems, size_t elsize) {
    void *ptr;
    /* Allocate storage from own pool, zero memory, and set ptr */
    return ptr;
}

void *my_realloc(void *ptr, size_t nbytes) {
    /* Relocate storage from own pool and set ptr */
    return ptr;
}

void my_free(void *ptr) {
    /* Return storage to own pool */
}
```

Noncompliant Code Example (errno)

According to the C Standard, 7.5, paragraph 2 [ISO/IEC 9899:2011], the behavior of a program is [undefined](#) when

A macro definition of `errno` is suppressed in order to access an actual object, or the program defines an identifier with the name `errno`.

See [undefined behavior 114](#).

The `errno` identifier expands to a modifiable [lvalue](#) that has type `int` but is not necessarily the identifier of an object. It might expand to a modifiable lvalue resulting from a function call, such as `*errno()`. It is unspecified whether `errno` is a macro or an identifier declared with external linkage. If a macro definition is suppressed to access an actual object, or if a program defines an identifier with the name `errno`, the behavior is [undefined](#).

Legacy code is apt to include an incorrect declaration, such as the following:

```
extern int errno;
```

Compliant Solution (errno)

The correct way to declare `errno` is to include the header `<errno.h>`:

```
#include <errno.h>
```

[Implementations conforming](#) to C are required to declare `errno` in `<errno.h>`, although some historic implementations failed to do so.

Exceptions

DCL37-C-EX1: Provided that a library function can be declared without reference to any type defined in a header, it is permissible to declare that function without including its header provided that declaration is compatible with the standard declaration.

```
/* Not including stdlib.h */
void free(void *);

void func(void *ptr) {
    free(ptr);
}
```

Such code is compliant because the declaration matches what `stdlib.h` would provide and does not redefine the reserved identifier. However, it would not be acceptable to provide a definition for the `free()` function in this example.

DCL37-C-EX2: For compatibility with other compiler vendors or language standard modes, it is acceptable to create a macro identifier that is the same as a reserved identifier so long as the behavior is idempotent, as in this example:

```
/* Sometimes generated by configuration tools such as autoconf */
#define const const

/* Allowed compilers with semantically equivalent extension behavior */
#define inline __inline
```

DCL37-C-EX3: As a compiler vendor or standard library developer, it is acceptable to use identifiers reserved for your implementation. Reserved identifiers may be defined by the compiler, in standard library headers or headers included by a standard library header, as in this example declaration from the glibc standard C library implementation:

```
/*
The following declarations of reserved identifiers exist in the glibc implementation of
<stdio.h>. The original source code may be found at:
https://sourceware.org/git/?p=glibc.git;a=blob_plain;f=include/stdio.h;hb=HEAD
```

```
/*
# define __need_size_t
# include <stddef.h>
/* Generate a unique file name (and possibly open it). */
extern int __path_search (char *__tmp1, size_t __tmp1_len,
    const char *__dir, const char *__pfx,
    int __try_tempdir);
```

Risk Assessment

Using reserved identifiers can lead to incorrect program operation.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL37-C	Low	Unlikely	Low	P3	L3

Related Guidelines

CERT C Secure Coding Standard	PRE00-C. Prefer inline or static functions to function-like macros PRE06-C. Enclose header files in an inclusion guard PRE31-C. Avoid side effects in arguments to unsafe macros
SEI CERT C++ Coding Standard	DCL51-CPP. Do not declare or define a reserved identifier
ISO/IEC TS 17961	Using identifiers that are reserved for the implementation [resident]
MISRA C:2012	Rule 21.1 (required) Rule 21.2 (required)

Bibliography

[IEEE Std 1003.1-2013]	Section 2.2, "The Compilation Environment"
[ISO/IEC 9899:2011]	7.1.3, "Reserved Identifiers" 7.31.10, "Integer Types <stdint.h>"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/-4AzAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
additional_reserved_identifiers	Names listed here are seen as violations as well.	set(['assert', 'errno', 'defined'])
allow_function_declarations	Whether nondefining function declarations with library names are allowed	True
allow_reserved_identifier_as_include_guard	If True, macros used for include guards are not checked for being a reserved identifier.	False
check_for_keyword	Whether macro names being keywords should be reported as well.	False
check_locals	Whether parameters and local variables should also be checked	True
check_reserved_enum_identifier	Whether enumerator names should be checked to use a reserved identifier	True
check_reserved_function_identifier	Whether function names should be checked to use a reserved identifier	True
check_reserved_macro_identifier	Whether to report reserved names (e.g. names starting with underscore).	False
check_reserved_type_identifier	Whether type names should be checked to use a reserved identifier	False
check_reserved_variable_identifier	Whether variable names should be checked to use a reserved identifier	True
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low
report_fields	Whether fields using a library name should be reported.	True

Possible Messages

Name	Message
decl_using_library_macro	The names of standard library macros, objects and functions, as well as reserved identifiers, shall not be reused.
enumerator_having_libname	The names of standard library macros, objects and functions, as well as reserved identifiers, shall not be reused.
field_having_libname	The names of standard library macros, objects and functions shall not be reused.
macro_having_libname	The names of standard library macros, objects and functions shall not be reused.
macro_having_reserved_name	Definition of reserved identifier or standard library element
routine_having_libname	The names of standard library macros, objects and functions, as well as reserved identifiers, shall not be reused.
type_having_libname	The names of standard library macros, objects and functions shall not be reused.
undef_of_reserved_name	#undef of reserved identifier or standard library element
variable_having_libname	The names of standard library macros, objects and functions, as well as reserved identifiers, shall not be reused.

CertC-DCL38

Use the correct syntax when declaring a flexible array member.

Input: IR

Source languages: C, C++

Details

Flexible array members are a special type of array in which the last element of a structure with more than one named member has an incomplete array type; that is, the size of the array is not specified explicitly within the structure. This "struct hack" was widely used in practice and supported by a variety of compilers. Consequently, a variety of different syntaxes have been used for declaring flexible array members. For conforming C implementations, use the syntax guaranteed to be valid by the C Standard.

Flexible array members are defined in the C Standard, 6.7.2.1, paragraph 18 [ISO/IEC 9899:2011], as follows:

As a special case, the last element of a structure with more than one named member may have an incomplete array type; this is called a *flexible*

array member. In most situations, the flexible array member is ignored. In particular, the size of the structure is as if the flexible array member were omitted except that it may have more trailing padding than the omission would imply. However, when a . (or ->) operator has a left operand that is (a pointer to) a structure with a flexible array member and the right operand names that member, it behaves as if that member were replaced with the longest array (with the same element type) that would not make the structure larger than the object being accessed; the offset of the array shall remain that of the flexible array member, even if this would differ from that of the replacement array. If this array would have no elements, it behaves as if it had one element but the behavior is undefined if any attempt is made to access that element or to generate a pointer one past it.

Structures with a flexible array member can be used to produce code with defined behavior. However, some restrictions apply:

1. The incomplete array type *must* be the last element within the structure.
2. There cannot be an array of structures that contain a flexible array member.
3. Structures that contain a flexible array member cannot be used as a member of another structure.
4. The structure must contain at least one named member in addition to the flexible array member.

Noncompliant Code Example

Before the introduction of flexible array members in the C Standard, structures with a one-element array as the final member were used to achieve similar functionality. This noncompliant code example illustrates how `struct flexArrayStruct` is declared in this case.

This noncompliant code example attempts to allocate a flexible array-like member with a one-element array as the final member. When the structure is instantiated, the size computed for `malloc()` is modified to account for the actual size of the dynamic array.

```
#include <stdlib.h>

struct flexArrayStruct {
    int num;
    int data[1];
};

void func(size_t array_size) {
    /* Space is allocated for the struct */
    struct flexArrayStruct *structP
        = (struct flexArrayStruct *)
            malloc(sizeof(struct flexArrayStruct)
                + sizeof(int) * (array_size - 1));
    if (structP == NULL) {
        /* Handle malloc failure */
    }

    structP->num = array_size;

    /*
     * Access data[] as if it had been allocated
     * as data[array_size].
     */
    for (size_t i = 0; i < array_size; ++i) {
        structP->data[i] = 1;
    }
}
```

This example has [undefined behavior](#) when accessing any element other than the first element of the `data` array. (See the C Standard, 6.5.6.) Consequently, the compiler can generate code that does not return the expected value when accessing the second element of data.

This approach may be the only alternative for compilers that do not yet implement the standard C syntax.

Compliant Solution

This compliant solution uses a flexible array member to achieve a dynamically sized structure:

```
#include <stdlib.h>

struct flexArrayStruct{
    int num;
    int data[];
};

void func(size_t array_size) {
    /* Space is allocated for the struct */
    struct flexArrayStruct *structP
        = (struct flexArrayStruct *)
            malloc(sizeof(struct flexArrayStruct)
                + sizeof(int) * array_size);
    if (structP == NULL) {
        /* Handle malloc failure */
    }

    structP->num = array_size;

    /*
     * Access data[] as if it had been allocated
     * as data[array_size].
     */
    for (size_t i = 0; i < array_size; ++i) {
        structP->data[i] = 1;
    }
}
```

This compliant solution allows the structure to be treated as if its member `data[]` was declared to be `data[array_size]` in a manner that conforms to the C Standard.

Failing to use the correct syntax when declaring a flexible array member can result in [undefined behavior](#), although the incorrect syntax will work on most implementations.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL38-C	Low	Unlikely	Low	P3	L3

Related Guidelines

This rule supplements [MEM33-C. Allocate and copy structures containing a flexible array member dynamically](#)

Bibliography

[ISO/IEC 9899:2011]	6.5.6, "Additive Operators" 6.7.2.1, "Structure and Union Specifiers"
[McCluskey 2001]	"Flexible Array Members and Designators in C9X"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [https://www.securecoding.cert.org/confluence/x/0wU_Ag], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
misused_nonflexible_array_member	Use flexible array member instead of abusing fixed-sized ones.

CertC-DCL39

Avoid information leakage when passing a structure across a trust boundary.

Input: IR

Source languages: C, C++

Details

The C Standard, 6.7.2.1, discusses the layout of structure fields. It specifies that non-bit-field members are aligned in an [implementation-defined](#) manner and that there may be padding within or at the end of a structure. Furthermore, initializing the members of the structure does not guarantee initialization of the padding bytes. The C Standard, 6.2.6.1, paragraph 6 [ISO/IEC 9899:2011], states

When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values.

Additionally, the storage units in which a bit-field resides may also have padding bits. For an object with automatic storage duration, these padding bits do not take on specific values and can contribute to leaking sensitive information.

When passing a pointer to a structure across a trust boundary to a different trusted domain, the programmer must ensure that the padding bytes and bit-field storage unit padding bits of such a structure do not contain sensitive information.

Noncompliant Code Example

This noncompliant code example runs in kernel space and copies data from `arg` to user space. However, padding bytes may be used within the structure, for example, to ensure the proper alignment of the structure members. These padding bytes may contain sensitive information, which may then be leaked when the data is copied to user space.

```
#include <stddef.h>

struct test {
    int a;
    char b;
    int c;
};
```

```
/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
    struct test arg = {.a = 1, .b = 2, .c = 3};
    copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

Noncompliant Code Example (`memset()`)

The padding bytes can be explicitly initialized by calling `memset()`:

```
#include <string.h>

struct test {
    int a;
    char b;
    int c;
};

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
    struct test arg;

    /* Set all bytes (including padding bytes) to zero */
    memset(&arg, 0, sizeof(arg));

    arg.a = 1;
    arg.b = 2;
    arg.c = 3;

    copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

However, compilers are free to implement `arg.b = 2` by setting the low byte of a 32-bit register to 2, leaving the high bytes unchanged and storing all 32 bits of the register into memory. This implementation could leak the high-order bytes resident in the register to a user.

Compliant Solution

This compliant solution serializes the structure data before copying it to an untrusted context:

```
#include <stddef.h>
#include <string.h>

struct test {
    int a;
    char b;
    int c;
};

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
    struct test arg = {.a = 1, .b = 2, .c = 3};
    /* May be larger than strictly needed */
    unsigned char buf[sizeof(arg)];
    size_t offset = 0;

    memcpy(buf + offset, &arg.a, sizeof(arg.a));
    offset += sizeof(arg.a);
    memcpy(buf + offset, &arg.b, sizeof(arg.b));
    offset += sizeof(arg.b);
    memcpy(buf + offset, &arg.c, sizeof(arg.c));
    offset += sizeof(arg.c);

    copy_to_user(usr_buf, buf, offset /* size of info copied */);
}
```

This code ensures that no uninitialized padding bytes are copied to unprivileged users. The structure copied to user space is now a packed structure and the `copy_to_user()` function would need to unpack it to recreate the original padded structure.

Compliant Solution (Padding Bytes)

Padding bytes can be explicitly declared as fields within the structure. This solution is not portable, however, because it depends on the [implementation](#) and target memory architecture. The following solution is specific to the x86-32 architecture:

```
#include <assert.h>
#include <stddef.h>

struct test {
    int a;
    char b;
    char padding_1, padding_2, padding_3;
    int c;
};

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
    /* Ensure c is the next byte after the last padding byte */
    static_assert(offsetof(struct test, c) ==
```

```

        offsetof(struct test, padding_3) + 1,
        "Structure contains intermediate padding");
/* Ensure there is no trailing padding */
static_assert(sizeof(struct test) ==
        offsetof(struct test, c) + sizeof(int),
        "Structure contains trailing padding");
struct test arg = { .a = 1, .b = 2, .c = 3 };
arg.padding_1 = 0;
arg.padding_2 = 0;
arg.padding_3 = 0;
copy_to_user(usr_buf, &arg, sizeof(arg));
}

```

The C Standard `static_assert()` macro accepts a constant expression and an [error message](#). The expression is evaluated at compile time and, if false, the compilation is terminated and the error message is output. (See [DCL03-C. Use a static assertion to test the value of a constant expression](#) for more details.) The explicit insertion of the padding bytes into the `struct` should ensure that no additional padding bytes are added by the compiler and consequently both static assertions should be true. However, it is necessary to validate these assumptions to ensure that the solution is correct for a particular implementation.

Compliant Solution [Structure Packing-GCC]

GCC allows specifying declaration attributes using the keyword `__attribute__((__packed__))`. When this attribute is present, the compiler will not add padding bytes for memory alignment unless otherwise required by the `_Alignas` alignment specifier, and it will attempt to place fields at adjacent memory offsets when possible.

```

#include <stddef.h>

struct test {
    int a;
    char b;
    int c;
} __attribute__((__packed__));
/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
    struct test arg = { .a = 1, .b = 2, .c = 3 };
    copy_to_user(usr_buf, &arg, sizeof(arg));
}

```

Compliant Solution [Structure Packing-Microsoft Visual Studio]

Microsoft Visual Studio supports `#pragma pack()` to suppress padding bytes [\[MSDN\]](#). The compiler adds padding bytes for memory alignment, depending on the current packing mode, but still honors the alignment specified by `__declspec(align())`. In this compliant solution, the packing mode is set to 1 in an attempt to ensure all fields are given adjacent offsets:

```

#include <stddef.h>

#pragma pack(push, 1) /* 1 byte */
struct test {
    int a;
    char b;
    int c;
};
#pragma pack(pop)

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
    struct test arg = { 1, 2, 3 };
    copy_to_user(usr_buf, &arg, sizeof(arg));
}

```

The `pack` pragma takes effect at the first `struct` declaration after the pragma is seen.

Noncompliant Code Example

This noncompliant code example also runs in kernel space and copies data from `struct test` to user space. However, padding bits will be used within the structure due to the bit-field member lengths not adding up to the number of bits in an `unsigned` object. Further, there is an unnamed bit-field that causes no further bit-fields to be packed into the same storage unit. These padding bits may contain sensitive information, which may then be leaked when the data is copied to user space. For instance, the uninitialized bits may contain a sensitive kernel space pointer value that can be trivially reconstructed by an attacker in user space.

```

#include <stddef.h>

struct test {
    unsigned a : 1;
    unsigned : 0;
    unsigned b : 4;
};

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
    struct test arg = { .a = 1, .b = 10 };
    copy_to_user(usr_buf, &arg, sizeof(arg));
}

```

However, compilers are free to implement the initialization of `arg.a` and `arg.b` by setting the low byte of a 32-bit register to the value specified, leaving the

high bytes unchanged and storing all 32 bits of the register into memory. This implementation could leak the high-order bytes resident in the register to a user.

Compliant Solution

Padding bits can be explicitly declared, allowing the programmer to specify the value of those bits. When explicitly declaring all of the padding bits, any unnamed bit-fields of length 0 must be removed from the structure because the explicit padding bits ensure that no further bit-fields will be packed into the same storage unit.

```
#include <assert.h>
#include <limits.h>
#include <stddef.h>

struct test {
    unsigned a : 1;
    unsigned padding1 : sizeof(unsigned) * CHAR_BIT - 1;
    unsigned b : 4;
    unsigned padding2 : sizeof(unsigned) * CHAR_BIT - 4;
};

/* Ensure that we have added the correct number of padding bits. */
static_assert(sizeof(struct test) == sizeof(unsigned) * 2,
              "Incorrect number of padding bits for type: unsigned");

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
    struct test arg = { .a = 1, .padding1 = 0, .b = 10, .padding2 = 0 };
    copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

This solution is not portable, however, because it depends on the [implementation](#) and target memory architecture. The explicit insertion of padding bits into the `struct` should ensure that no additional padding bits are added by the compiler. However, it is still necessary to validate these assumptions to ensure that the solution is correct for a particular implementation. For instance, the DEC Alpha is an example of a 64-bit architecture with 32-bit integers that allocates 64 bits to a storage unit.

In addition, this solution assumes that there are no integer padding bits in an `unsigned int`. The portable version of the width calculation from [INT35-C. Use correct integer precisions](#) cannot be used because the bit-field width must be an integer constant expression.

From this situation, it can be seen that special care must be taken because no solution to the bit-field padding issue will be 100% portable.

Risk Assessment

Padding units might contain sensitive data because the C Standard allows any padding to take [unspecified values](#). A pointer to such a structure could be passed to other functions, causing information leakage.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL39-C	Low	Unlikely	High	P1	L3

Related Guidelines

CERT C Secure Coding Standard	DCL03-C. Use a static assertion to test the value of a constant expression
---	--

Bibliography

[ISO/IEC 9899:2011]	6.2.6.1, "General" 6.7.2.1, "Structure and Union Specifiers"
[Graff 2003]	
[Sun 1993]	

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/IABIAw>], Copyright [C] 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	1
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high
trust_boundary_functions	When names of functions are provided in this set, only calls to these functions are reported when a pointer to some struct/class with padding is passed as argument	set[[])

Possible Messages

Name	Message
composite_with_padding	Composite type has padding after field '{}'.
pointer_to_padded_composite_as_argument	Pointer to composite type '{}' with padding after field '{}' is passed as argument: potential information leak.

CertC-DCL40

Do not create incompatible declarations of the same function or object.

Input: IR

Source languages: C, C++

Details

Two or more incompatible declarations of the same function or object must not appear in the same program because they result in [undefined behavior](#). The C Standard, 6.2.7, mentions that two types may be distinct yet compatible and addresses precisely when two distinct types are compatible.

The C Standard identifies four situations in which [undefined behavior \(UB\)](#) may arise as a result of incompatible declarations of the same function or object:

UB	Description	Code
15	<i>Two declarations of the same object or function specify types that are not compatible (6.2.7).</i>	All noncompliant code in this guideline
31	<i>Two identifiers differ only in nonsignificant characters (6.4.2.1).</i>	Excessively Long Identifiers
37	<i>An object has its stored value accessed other than by an lvalue of an allowable type (6.5).</i>	Incompatible Object Declarations Incompatible Array Declarations
41	<i>A function is defined with a type that is not compatible with the type [of the expression] pointed to by the expression that denotes the called function (6.5.2.2).</i>	Incompatible Function Declarations Excessively Long Identifiers

Although the effect of two incompatible declarations simply appearing in the same program may be benign on most [implementations](#), the effects of invoking a function through an expression whose type is incompatible with the function definition are typically catastrophic. Similarly, the effects of accessing an object using an [lvalue](#) of a type that is incompatible with the object definition may range from unintended information exposure to memory overwrite to a hardware trap.

Noncompliant Code Example (Incompatible Object Declarations)

In this noncompliant code example, the variable `i` is declared to have type `int` in file `a.c` but defined to be of type `short` in file `b.c`. The declarations are incompatible, resulting in [undefined behavior 15](#). Furthermore, accessing the object using an [lvalue](#) of an incompatible type, as shown in function `f()`, is [undefined behavior 37](#) with possible observable results ranging from unintended information exposure to memory overwrite to a hardware trap.

```
/* In a.c */
extern int i; /* UB 15 */

int f(void) {
    return ++i; /* UB 37 */
}

/* In b.c */
short i; /* UB 15 */
```

Compliant Solution (Incompatible Object Declarations)

This compliant solution has compatible declarations of the variable `i`:

```
/* In a.c */
extern int i;

int f(void) {
    return ++i;
}

/* In b.c */
int i;
```

Noncompliant Code Example (Incompatible Array Declarations)

In this noncompliant code example, the variable `a` is declared to have a pointer type in file `a.c` but defined to have an array type in file `b.c`. The two declarations are incompatible, resulting in [undefined behavior 15](#). As before, accessing the object in function `f()` is [undefined behavior 37](#) with the typical effect of triggering a hardware trap.

```
/* In a.c */
extern int *a; /* UB 15 */

int f(unsigned int i, int x) {
    int tmp = a[i]; /* UB 37: read access */
    a[i] = x; /* UB 37: write access */
    return tmp;
}

/* In b.c */
int a[] = { 1, 2, 3, 4 }; /* UB 15 */
```

Compliant Solution (Incompatible Array Declarations)

This compliant solution declares `a` as an array in `a.c` and `b.c`:

```
/* In a.c */
extern int a[];

int f(unsigned int i, int x) {
    int tmp = a[i];
    a[i] = x;
    return tmp;
}

/* In b.c */
int a[] = { 1, 2, 3, 4 };
```

Noncompliant Code Example (Incompatible Function Declarations)

In this noncompliant code example, the function `f()` is declared in file `a.c` with one prototype but defined in file `b.c` with another. The two prototypes are incompatible, resulting in [undefined behavior 15](#). Furthermore, invoking the function is [undefined behavior 41](#) and typically has catastrophic consequences.

```
/* In a.c */
extern int f(int a); /* UB 15 */

int g(int a) {
    return f(a); /* UB 41 */
}

/* In b.c */
long f(long a) { /* UB 15 */
    return a * 2;
}
```

Compliant Solution (Incompatible Function Declarations)

This compliant solution has compatible prototypes for the function `f()`:

```
/* In a.c */
extern int f(int a);

int g(int a) {
    return f(a);
}

/* In b.c */
int f(int a) {
    return a * 2;
}
```

Noncompliant Code Example (Incompatible Variadic Function Declarations)

In this noncompliant code example, the function `buginf()` is defined to take a variable number of arguments and expects them all to be signed integers with a sentinel value of `-1`:

```
/* In a.c */
void buginf(const char *fmt, ...) {
    /* ... */
}

/* In b.c */
void buginf();
```

Although this code appears to be well defined because of the prototype-less declaration of `buginf()`, it exhibits [undefined behavior](#) in accordance with the C

For two function types to be compatible, both shall specify compatible return types. Moreover, the parameter type lists, if both are present, shall agree in the number of parameters and in use of the ellipsis terminator; corresponding parameters shall have compatible types. If one type has a parameter type list and the other type is specified by a function declarator that is not part of a function definition and that contains an empty identifier list, the parameter list shall not have an ellipsis terminator and the type of each parameter shall be compatible with the type that results from the application of the default argument promotions.

Compliant Solution (Incompatible Variadic Function Declarations)

In this compliant solution, the prototype for the function `buginf()` is included in the scope in the source file where it will be used:

```
/* In a.c */
void buginf(const char *fmt, ...) {
    /* ... */
}

/* In b.c */
void buginf(const char *fmt, ...);
```

Noncompliant Code Example (Excessively Long Identifiers)

In this noncompliant code example, the length of the identifier declaring the function pointer `bash_groupname_completion()` in the file `bashline.h` exceeds by 3 the minimum implementation limit of 31 significant initial characters in an external identifier. This introduces the possibility of colliding with the `bash_groupname_completion_funct` integer variable defined in file `b.c`, which is exactly 31 characters long. On an implementation that exactly meets this limit, this is [undefined behavior 31](#). It results in two incompatible declarations of the same function. (See [undefined behavior 15](#).) In addition, invoking the function leads to [undefined behavior 41](#) with typically catastrophic effects.

```
/* In bashline.h */
/* UB 15, UB 31 */
extern char * bash_groupname_completion_function(const char *, int);

/* In a.c */
#include "bashline.h"

void f(const char *s, int i) {
    bash_groupname_completion_function(s, i); /* UB 41 */
}

/* In b.c */
int bash_groupname_completion_funct; /* UB 15, UB 31 */
```

NOTE: The identifier `bash_groupname_completion_function` referenced here was taken from GNU [Bash](#), version 3.2.

Compliant Solution (Excessively Long Identifiers)

In this compliant solution, the length of the identifier declaring the function pointer `bash_groupname_completion()` in `bashline.h` is less than 32 characters. Consequently, it cannot clash with `bash_groupname_completion_funct` on any compliant platform.

```
/* In bashline.h */
extern char * bash_groupname_completion(const char *, int);

/* In a.c */
#include "bashline.h"

void f(const char *s, int i) {
    bash_groupname_completion(s, i);
}

/* In b.c */
int bash_groupname_completion_funct;
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL40-C	Low	Unlikely	Medium	P2	L3

Related Guidelines

ISO/IEC TS 17961	Declaring the same function or object in incompatible ways [funcdecl]
MISRA C:2012	Rule 8.4 (required)

Bibliography

Hatton 1995	Section 2.8.3
ISO/IEC 9899:2011	6.7.6.3, "Function Declarators (including Prototypes)" J.2, "Undefined Behavior"

Configuration

Name	Explanation	Value
check_undefined	Whether only-declared variables should also be checked.	False
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
maxlen		31
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
report_external_identifiers	Whether external identifiers should be compared to each other.	True
report_internal_identifiers	Whether internal identifiers should be compared to each other (including macros).	False
report_short_identifiers	If True, identifiers shorter than maxlen are considered as well.	False
reported_messages	If provided, only messages of these types are reported.	147
reported_severities	List of severities to display.	('error', 'warning', 'remark')
require_exact_match	Whether to check for identical or compatible types.	True
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

Possible Messages

Name	Message
decl_return_type_mismatch	Return type at declarations differs
decl_type_mismatch	Type at declarations differs
external_identifiers_not_distinct	External identifiers not distinct.
external_identifiers_sharing	External identifiers sharing first {} characters.
incompatible_parameters	Parameters of definition and declarations of a function shall be compatible
internal_identifiers_not_distinct	Internal identifiers not distinct.
internal_identifiers_sharing	Internal identifiers sharing first {} characters.
return_type_mismatch	Return type at definition differs from type at declaration
type_mismatch	Type at definition differs from type at declaration

CertC-DCL41

Do not declare variables inside a switch statement before the first case label.

Input: IR

Source languages: C, C++

Details

According to the C Standard, 6.8.4.2, paragraph 4 [[ISO/IEC 9899:2011](#)],

A switch statement causes control to jump to, into, or past the statement that is the switch body, depending on the value of a controlling expression, and on the presence of a default label and the values of any case labels on or in the switch body.

If a programmer declares variables, initializes them before the first case statement, and then tries to use them inside any of the case statements, those variables will have scope inside the `switch` block but will not be initialized and will consequently contain indeterminate values.

Noncompliant Code Example

This noncompliant code example declares variables and contains executable statements before the first case label within the `switch` statement:

```
#include <stdio.h>

extern void f(int i);

void func(int expr) {
    switch (expr) {
        int i = 4;
        f(i);
    case 0:
        i = 17;
        /* Falls through into default code */
    default:
        printf("%d\n", i);
    }
}
```

Implementation Details

When the preceding example is executed on GCC 4.8.1, the variable `i` is instantiated with automatic storage duration within the block, but it is not initialized. Consequently, if the controlling expression `expr` has a nonzero value, the call to `printf()` will access an indeterminate value of `i`. Similarly, the call to `f()` is not executed.

Value of <code>expr</code>	Output
0	17
Nonzero	Indeterminate

Compliant Solution

In this compliant solution, the statements before the first case label occur before the `switch` statement:

```
#include <stdio.h>

extern void f(int i);

int func(int expr) {
    /*
     * Move the code outside the switch block; now the statements
     * will get executed.
     */
    int i = 4;
    f(i);

    switch (expr) {
        case 0:
            i = 17;
            /* Falls through into default code */
        default:
            printf("%d\n", i);
    }
    return 0;
}
```

Risk Assessment

Using test conditions or initializing variables before the first case statement in a `switch` block can result in [unexpected behavior](#) and [undefined behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL41-C	Medium	Unlikely	Medium	P4	L3

Related Guidelines

MISRA C:2012	Rule 16.1 (required)
------------------------------	----------------------

Bibliography

[ISO/IEC 9899:2011]	6.8.4.2, "The <code>switch</code> Statement"
-------------------------------------	--

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/A4EzAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
switch_not_well_formed	Switch has a statement before the first case label.

CertC-EXP00

Use parentheses for precedence of operation.

Input: IR

Source languages: C, C++

Details

C programmers commonly make errors regarding the precedence rules of C operators because of the unintuitive low-precedence levels of `&`, `|`, `^`, `<<`, and `>>`. Mistakes regarding precedence rules can be avoided by the suitable use of parentheses. Using parentheses defensively reduces errors and, if not taken to excess, makes the code more readable.

Subclause 6.5 of the C Standard defines the precedence of operation by the order of the subclauses.

Noncompliant Code Example

The intent of the expression in this noncompliant code example is to test the least significant bit of `x`:

```
x & 1 == 0
```

Because of operator precedence rules, the expression is parsed as

```
x & (1 == 0)
```

which evaluates to

```
(x & 0)
```

and then to 0.

Compliant Solution

In this compliant solution, parentheses are used to ensure the expression evaluates as expected:

```
(x & 1) == 0
```

Exceptions

EXP00-C-EX1: Mathematical expressions that follow algebraic order do not require parentheses. For instance, in the expression

```
x + y * z
```

the multiplication is performed before the addition by mathematical convention. Consequently, parentheses to enforce the algebraic order would be redundant:

```
x + (y * z)
```

Risk Assessment

Mistakes regarding precedence rules may cause an expression to be evaluated in an unintended way, which can lead to [unexpected](#) and abnormal program behavior.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
EXP00-C	Low	Probable	Medium	P4	L3

Related Guidelines

SEI CERT C++ Coding Standard	EXP00-CPP_Use parentheses for precedence of operation
ISO/IEC TR 24772:2013	Operator Precedence/Order of Evaluation [JCW]
MISRA C:2012	Rule 12.1 (advisory)

Bibliography

[Dowd 2006]	Chapter 6, "C Language Issues" ("Precedence," pp. 287-288)
[Kernighan 1988]	
[NASA-GB-1740.13]	Section 6.4.3, "C Language"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [https://www.securecoding.cert.org/confluence/x_wl], Copyright [C] 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
allow_algebraic_order	Whether parens can be omitted for multiplicative operators as operands in additive operators	True
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
missing_parens_depends_on_precedence	Parentheses should be used to avoid dependence on precedence rules
parens_duplicating_algebraic_order	Mathematical expressions that follow algebraic order do not require parentheses

CertC-EXP02

Be aware of the short-circuit behaviour of the logical AND and OR operators.

Input: IR

Source languages: C, C++

Details

The logical AND and logical OR operators (`&&` and `||`, respectively) exhibit "short-circuit" operation. That is, the second operand is not evaluated if the result can be deduced solely by evaluating the first operand.

Programmers should exercise caution if the second operand contains [side effects](#) because it may not be apparent whether the side effects actually occur.

In the following code, the value of `i` is incremented only when `i >= 0`:

```
enum { max = 15 };
int i = /* Initialize to user-supplied value */;

if ( (i >= 0) && ( (i++) <= max) ) {
    /* Code */
}
```

Although the behavior is well defined, it is not immediately obvious whether or not `i` gets incremented.

Noncompliant Code Example

In this noncompliant code example, the second operand of the logical OR operator invokes a function that results in side effects:

```
char *p = /* Initialize; may or may not be NULL */

if (p || (p = (char *) malloc(BUF_SIZE)) ) {
    /* Perform some computation based on p */
    free(p);
    p = NULL;
} else {
    /* Handle malloc() error */
    return;
}
```

Because `malloc()` is called only if `p` is `NULL` when entering the `if` clause, `free()` might be called with a pointer to local data not allocated by `malloc()`. (See [MEM34-C. Only free memory allocated dynamically.](#)) This behavior is partially due to the uncertainty of whether or not `malloc()` is actually called.

Compliant Solution

In this compliant solution, a second pointer, `q`, is used to indicate whether `malloc()` is called; if not, `q` remains set to `NULL`. Passing `NULL` to `free()` is guaranteed to safely do nothing.

```
char *p = /* Initialize; may or may not be NULL */
char *q = NULL;
if (p == NULL) {
    q = (char *) malloc(BUF_SIZE);
    p = q;
}
if (p == NULL) {
    /* Handle malloc() error */
    return;
}

/* Perform some computation based on p */
free(q);
q = NULL;
```

Risk Assessment

Failing to understand the short-circuit behavior of the logical OR or AND operator may cause unintended program behavior.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
EXP02-C	Low	Unlikely	Medium	P2	L3

Related Guidelines

SEI CERT C++ Coding Standard	EXP02-CPP. Be aware of the short-circuit behavior of the logical AND and OR operators
MITRE CWE	CWE-768, Incorrect short circuit evaluation

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/loAD>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
allow_volatile_access	Whether access to a volatile variable should be reported as side effect.	False
assume_effect_when_undefined	Whether all calls to undefined [external] functions should be seen as having a side-effect.	True
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
call_with_side_effect	Call in right-hand operand of && or may have side-effect
modifies_local_var	Right-hand operand of && or modifies '{}'
side_effect	Right-hand operand of && or has side-effect

CertC-EXP05

Do not cast away a const qualification.

Input: IR

Source languages: C, C++

Details

Do not cast away a `const` qualification on an object of pointer type. Casting away the `const` qualification allows a program to modify the object referred to by the pointer, which may result in [undefined behavior](#). See [undefined behavior 64](#) in Appendix J of the C Standard.

As an illustration, the C Standard [[ISO/IEC 9899:2011](#)] provides a footnote [subclause 6.7.3, paragraph 4]:

The implementation may place a `const` object that is not volatile in a read-only region of storage. Moreover, the implementation need not allocate storage for such an object if its address is never used.

Noncompliant Code Example

The `remove_spaces()` function in this noncompliant code example accepts a pointer to a string `str` and a string length `slen` and removes the space character from the string by shifting the remaining characters toward the front of the string. The function `remove_spaces()` is passed a `const char` pointer as an argument. The `const` qualification is cast away, and then the contents of the string are modified.

```
void remove_spaces(const char *str, size_t slen) {
    char *p = (char *)str;
    size_t i;
    for (i = 0; i < slen && str[i]; i++) {
        if (str[i] != ' ') *p++ = str[i];
    }
    *p = '\0';
}
```

Compliant Solution

In this compliant solution, the function `remove_spaces()` is passed a `non-const char` pointer. The calling function must ensure that the null-terminated byte string passed to the function is not `const` by making a copy of the string or by other means.

```
void remove_spaces(char *str, size_t slen) {
    char *p = str;
    size_t i;
    for (i = 0; i < slen && str[i]; i++) {
        if (str[i] != ' ') *p++ = str[i];
    }
    *p = '\0';
}
```

Noncompliant Code Example

In this noncompliant code example, the contents of the `const int` array `vals` are cleared by the call to `memset()`:

```
const int vals[3] = {3, 4, 5};
memset(vals, 0, sizeof(vals));
```

Because the `memset()` function takes a `(non-const)` pointer to `void`, the compiler must implicitly cast away `const`.

Implementation Details

The GCC compiler issues a warning when an implicit cast is performed.

Compliant Solution

If the intention is to allow the array values to be modified, do not declare the array as `const`:

```
int vals[3] = {3, 4, 5};
memset(vals, 0, sizeof(vals));
```

Otherwise, do not attempt to modify the contents of the array.

Exceptions

EXP05-C-EX1: An exception to this recommendation is allowed when it is necessary to cast away `const` when invoking a legacy API that does not accept a `const` argument, provided the function does not attempt to modify the referenced variable. For example, the following code casts away the `const` qualification of `INVNAME` in the call to the `audit_log()` function.

```
/* Legacy function defined elsewhere - cannot be modified */
void audit_log(char *errstr) {
    fprintf(stderr, "Error: %s.\n", errstr);
}

/* ... */
const char INVNAME[] = "Invalid file name.";
audit_log((char *)INVNAME); /* EXP05-EX1 */
/* ... */
```

EXP05-C-EX2: A number of C standard library functions are specified to return `non-const` pointers that refer to their `const`-qualified arguments. When the actual arguments to such functions reference `const` objects, attempting to use the returned `non-const` pointers to modify the `const` objects would be a violation of [EXP40-C. Do not modify constant objects](#) and would lead to [undefined behavior](#). These functions are the following:

memchr	strchr	strpbrk	strrchr
strstr	strtod	strtod	strtold
strtol	strtoll	strtoul	strtoull
wmemchr	wcschr	wcspbrk	wcsrchr
wcsstr			

For instance, in following example, the function `strchr` returns an unqualified `char*` that points to the terminating null character of the constant character array `s` (which could be stored in ROM). Even though the pointer is not `const`, attempting to modify the character it points to would lead to undefined behavior.

```
extern const char s[];
char* where;
where = strchr(s, '\0');
/* Modifying *s is undefined */
```

Similarly, in the following example, the function `strtol` sets the unqualified `char*` pointer referenced by `end` to point just past the last successfully parsed character of the constant character array `s` (which could be stored in ROM). Even though the pointer is not `const`, attempting to modify the character it points to would lead to undefined behavior.

```
extern const char s[];
long x;
char* end;
x = strtol(s, &end, 0);
/* Modifying **end is undefined */
```

EXP05-C-EX3: Because `const` means "read-only," and not "constant," it is sometimes useful to declare `struct` members as (pointer to) `const` objects to obtain diagnostics when the user tries to change them in some way other than via the functions that are specifically designed to maintain that data type. Within those functions, however, it may be necessary to strip off the `const` qualification to update those members.

Risk Assessment

If the object is constant, the compiler may allocate storage in ROM or write-protected memory. Attempting to modify such an object may lead to a program crash or [denial-of-service attack](#).

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
EXP05-C	Medium	Probable	Medium	P8	L2

Related Guidelines

SEI CERT C++ Coding Standard	EXP55-CPP. Do not access a cv-qualified object through a cv-unqualified type
ISO/IEC TR 24772:2013	Pointer Casting and Pointer Type Changes [HFC] Type System [IHN]
MISRA C:2012	Rule 11.8 (required)
MITRE CWE	CWE-704 , Incorrect type conversion or cast

Bibliography

[ISO/IEC 9899:2011]	Subclause 6.7.3, "Type Qualifiers"
-------------------------------------	------------------------------------

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/VAE>], Copyright [C] 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
check_const	Whether casts affecting const-qualifiers should be reported	True
check_volatile	Whether casts affecting volatile-qualifiers should be reported	False
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
only_top_level	Whether the check only applies to the top-level pointee or recursively to deeper pointer levels as well	False
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
cast_removes_const	Cast removes const qualification
cast_removes_volatile	Cast removes volatile qualification
implicit_cast_removes_const	Implicit cast removes const qualification
implicit_cast_removes_volatile	Implicit cast removes volatile qualification

CertC-EXP07

Do not diminish the benefits of constants by assuming their values in expressions.

Input: IR

Source languages: C, C++

Details

If a constant value is given for an identifier, do not diminish the maintainability of the code in which it is used by assuming its value in expressions. Simply giving the constant a name is not enough to ensure modifiability; you must be careful to always use the name, and remember that the value can change. This recommendation is related to [DCL06-C. Use meaningful symbolic constants to represent literal values](#).

Noncompliant Code Example

The header `<stdio.h>` defines the `BUFSIZ` macro, which expands to an integer constant expression that is the size of the buffer used by the `setbuf()` function. This noncompliant code example defeats the purpose of defining `BUFSIZ` as a constant by assuming its value in the following expression:

```
#include <stdio.h>
/* ... */
nblocks = 1 + ((nbytes - 1) >> 9); /* BUFSIZ = 512 = 2^9 */
```

The programmer's assumption underlying this code is that "everyone knows that `BUFSIZ` equals 512," and right-shifting 9 bits is the same (for positive numbers) as dividing by 512. However, if `BUFSIZ` changes to 1024 on some systems, modifications are difficult and error prone.

Compliant Solution

This compliant solution uses the identifier assigned to the constant value in the expression:

```
#include <stdio.h>
/* ... */
nblocks = 1 + (nbytes - 1) / BUFSIZ;
```

Most modern C compilers will optimize this code appropriately.

Risk Assessment

Assuming the value of an expression diminishes the maintainability of code and can produce [unexpected behavior](#) under any circumstances in which the constant changes.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
EXP07-C	Low	Unlikely	Medium	P2	L3

Related Guidelines

Bibliography

[Plum 1985]	Rule 1-5
-------------	----------

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/XgBi>], Copyright [C] 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
allow_nonconst_variable_initialization	If set to 'True', allow the initialization of a non-const variable with a string literal in addition to the initialization of a const variable. A reassignment of such a variable is still a violation.	False
allowed	Literal values that are ok.	{0.0, 1.0, 2.0}
allowed_contexts	Optional set of PIR classes or functions <code>{(node) -> bool}</code> for allowed contexts, e.g. Case_Label.	<code>set[[]]</code>
exceptions	Optional predicate to filter out cases that should be allowed as an exception. The predicate takes a literal node and returns True if the given node is such an exception.	None
exclude_pp_literals	If True, literals in conditions of #if are ignored.	True
exclude_single_uses	If True, report only literals used more than once.	False
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
magic_number	Use of magic literal.
magic_number_without_token	Use of magic literal.
possible_magic_number	Potential use of magic literal.

CertC-EXP10

Do not depend on the order of evaluation of subexpressions or the order in which side effects take place.

Input: IR

Source languages: C, C++

Details

The order of evaluation of subexpressions and the order in which [side effects](#) take place are frequently defined as [unspecified behavior](#) by the C Standard. Counterintuitively, [unspecified behavior](#) in behavior for which the standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance. Consequently, unspecified behavior can be a portability issue because different [implementations](#) can make different choices. If dynamic scheduling is used, however, there may not be a fixed-code execution sequence over the life of a process. Operations that can be executed in different sequences may in fact be executed in a different order.

According to the C Standard, subclause 6.5 [[ISO/IEC 9899:2011](#)],

Except as specified later, side effects and value computations of subexpressions are unsequenced.

Following are specific examples of situations in which the order of evaluation of subexpressions or the order in which [side effects](#) take place is unspecified:

- The order in which the arguments to a function are evaluated (C Standard, subclause 6.5.2.2, "Function Calls")
- The order of evaluation of the operands in an assignment statement (C Standard, subclause 6.5.16, "Assignment Operators")
- The order in which any side effects occur among the initialization list expressions is unspecified. In particular, the evaluation order need not be the same as the order of subobject initialization (C Standard, subclause 6.7.9, "Initialization")

This recommendation is related to [EXP30-C. Do not depend on the order of evaluation for side effects](#), but it focuses on behavior that is nonportable or potentially confusing.

Noncompliant Code Example

The order of evaluation of the function designator, the actual arguments, and subexpressions within the actual arguments are unspecified, but there is a sequence point before the actual call. For example, in the function call

```
(*pf[f1()]) (f2(), f3() + f4())
```

the functions `f1()`, `f2()`, `f3()`, and `f4()` may be called in any order. All side effects have to be completed before the function pointed to by `pf[f1()]` is called.

Consequently, the result of this noncompliant code example depends on [unspecified behavior](#):

```
#include <stdio.h>

int g;

int f(int i) {
    g = i;
    return i;
}

int main(void) {
    int x = f(1) + f(2);
    printf("g = %d\n", g);
    /* ... */
    return 0;
}
```

This code may result in `g` being assigned the value 1, or equally likely, being assigned the value 2.

Compliant Solution

This compliant solution is independent of the order of evaluation of the operands and can be interpreted in only one way:

```
#include <stdio.h>

int g;

int f(int i) {
    g = i;
    return i;
}

int main(void) {
    int x = f(1);
    x += f(2);
    printf("g = %d\n", g);
    /* ... */
    return 0;
}
```

This code always results in `g` being assigned the value 2.

Exceptions

EXP10-C-EX1: The `&&` and `||` operators guarantee left-to-right evaluation; there is a sequence point after the evaluation of the first operand.

EXP10-C-EX2: The first operand of a condition expression is evaluated; there is a sequence point after its evaluation. The second operand is evaluated only if the first compares unequal to 0; the third operand is evaluated only if the first compares equal to 0.

EXP10-C-EX3: There is a sequence point before function calls, meaning that the function designator, the actual arguments, and subexpressions within the actual arguments are evaluated before the function is invoked.

EXP10-C-EX4: The left operand of a comma operator is evaluated before the right operand is evaluated. There is a sequence point in between.

Note that whereas commas serve to delimit multiple arguments in a function call, these commas are not considered comma operators. Multiple arguments of a function call may be evaluated in any order, with no sequence points between each other.

Risk Assessment

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
EXP10-C	Medium	Probable	Medium	P8	L2

Related Guidelines

SEI CERT C++ Coding Standard	EXP50-CPP. Do not depend on the order of evaluation for side effects
ISO/IEC TR 24772:2013	Operator Precedence/Order of Evaluation [JCW] Side-effects and Order of Evaluation [SAM]
MISRA C:2012	Rule 13.5 (required)

Bibliography

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/WQD3>], Copyright [C] 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
ignore_exceptions	Ignore possible exceptions in the computation of side effects. (this option only has an effect if report_calls is enabled)	False
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium
report_calls	If True, unsequenced function calls are reported.	True

Possible Messages

Name	Message
unsequenced_call_volatile	Function calls unsequenced with volatile access
unsequenced_calls	Unsequenced function calls
unsequenced_read_write	Unsequenced read and write accesses
unsequenced_volatiles	Unsequenced volatile accesses
unsequenced_writes	Unsequenced writes

CertC-EXP12

Do not ignore values returned by functions.

Input: IR

Source languages: C, C++

Details

Many functions return useful values whether or not the function has side effects. In most cases, this value is used to signify whether the function successfully completed its task or if some error occurred (see [ERR02-C. Avoid in-band error indicators](#)). Other times, the value is the result of some computation and is an integral part of the function's API.

Subclause 6.8.3 of the C Standard [[ISO/IEC 9899:2011](#)] states:

The expression in an expression statement is evaluated as a void expression for its side effects.

All expression statements, such as function calls with an ignored value, are implicitly cast to `void`. Because a return value often contains important information about possible errors, it should always be checked; otherwise, the cast should be made explicit to signify programmer intent. If a function returns no meaningful value, it should be declared with return type `void`.

This recommendation encompasses [ERR33-C. Detect and handle standard library errors](#).

Noncompliant Code Example

This noncompliant code example calls `puts()` and fails to check whether a write error occurs:

```
puts("foo");
```

However, `puts()` can fail and return `EOF`.

Compliant Solution

This compliant solution checks to make sure no output error occurred (see [ERR33-C. Detect and handle standard library errors](#)).

```
if (puts("foo") == EOF) {
    /* Handle error */
}
```

Exceptions

EXP12-C-EX1: If the return value is inconsequential or if any errors can be safely ignored, such as for functions called because of their side effects, the function should be explicitly cast to `void` to signify programmer intent. For an example of this exception, see "Compliant Solution (Remove Existing Destination File)" under the section "Portable Behavior" in [F1010-C. Take care when using the rename\(\) function](#).

EXP12-C-EX2: If a function cannot fail or if the return value cannot signify an error condition, the return value may be ignored. Such functions should be added to a whitelist when automatic checkers are used.

```
strcpy(dst, src);
```

Risk Assessment

Failure to handle error codes or other values returned by functions can lead to incorrect program flow and violations of data integrity.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
EXP12-C	Medium	Unlikely	Medium	P4	L3

Related Guidelines

SEI CERT C++ Coding Standard	EXP12-CPP. Do not ignore values returned by functions or methods
CERT Oracle Secure Coding Standard for Java	EXP00-J. Do not ignore values returned by methods
ISO/IEC TR 24772:2013	Passing Parameters and Return Values [CSJ]
MITRE CWE	CWE-754 , Improper check for unusual or exceptional conditions

Bibliography

[ISO/IEC 9899:2011]	Subclause 6.8.3, "Expression and Null Statements"
-------------------------------------	---

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/9YIRAQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
error_types	List of user defined error type names (if empty, all ignored int values are reported).	[]
inspect_template_instances	Whether calls in template instances should be reported.	False
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium
whitelist	Dictionary of header globbing to (list of) function names whose return codes can be ignored.	dict{...}

Possible Messages

Name	Message
discarded_return	Return value of function discarded.

CertC-EXP14

Beware of integer promotion when performing bitwise operations on integer types smaller than int.

Input: IR

Source languages: C, C++

Details

Deprecated

This guideline has been deprecated by

- [INT02-C. Understand integer conversion rules](#)

Integer types smaller than `int` are promoted when an operation is performed on them. If all values of the original type can be represented as an `int`, the value of the smaller type is converted to an `int`; otherwise, it is converted to an `unsigned int` (see [INT02-C. Understand integer conversion rules](#)). If the conversion is to a wider type, the original value is zero-extended for unsigned values or sign-extended for signed types. Consequently, bitwise operations on integer types smaller than `int` may have unexpected results.

Noncompliant Code Example

This noncompliant code example demonstrates how performing bitwise operations on integer types smaller than `int` may have unexpected results.

```
uint8_t port = 0x5a;
uint8_t result_8 = (~port) >> 4;
```

In this example, a bitwise complement of `port` is first computed and then shifted 4 bits to the right. If both of these operations are performed on an 8-bit unsigned integer, then `result_8` will have the value `0x0a`. However, `port` is first promoted to a `signed int`, with the following results (on a typical architecture where type `int` is 32 bits wide):

Expression	Type	Value	Notes
<code>port</code>	<code>uint8_t</code>	<code>0x5a</code>	
<code>~port</code>	<code>int</code>	<code>0xffffffffa5</code>	
<code>~port >> 4</code>	<code>int</code>	<code>0x0fffffa</code>	Whether or not value is negative is implementation-defined.
<code>result_8</code>	<code>uint8_t</code>	<code>0xfa</code>	

Compliant Solution

In this compliant solution, the bitwise complement of `port` is converted back to 8 bits. Consequently, `result_8` is assigned the expected value of `0x0a`.

```
uint8_t port = 0x5a;
uint8_t result_8 = (uint8_t) (~port) >> 4;
```

Risk Assessment

Bitwise operations on shorts and chars can produce incorrect data.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
EXP14-C	low	likely	high	P3	L3

Related Guidelines

SEI CERT C++ Coding Standard	EXP15-CPP. Beware of integer promotion when performing bitwise operations on chars or shorts
MISRA-C	Rule 10.5

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/UwDFAQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
<code>level</code>	Grouping of priorities into different levels	3
<code>likelihood</code>	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
<code>priority</code>	Priority based on the combination of severity, likelihood and remediation cost	3
<code>recommendation</code>	Whether this check is classified as a recommendation or rule	True
<code>remediation_cost</code>	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
<code>bitop_small_without_cast</code>	Bitwise operator requires cast to underlying type on result

CertC-EXP15

Do not place a semicolon on the same line as an if, for, or while statement.

Input: IR

Source languages: C, C++

Details

Do not use a semicolon on the same line as an `if`, `for`, or `while` statement because it typically indicates programmer error and can result in unexpected behavior.

Noncompliant Code Example

In this noncompliant code example, a semicolon is used on the same line as an `if` statement:

```
if (a == b); {  
/* ... */  
}
```

Compliant Solution

It is likely, in this example, that the semicolon was accidentally inserted:

```
if (a == b) {  
/* ... */  
}
```

Related Guidelines

SEI CERT Oracle Coding Standard for Java	MSC51-J. Do not place a semicolon immediately following an if, for, or while condition
ISO/IEC TR 24772:2013	Likely Incorrect Expression [KOA]
MITRE CWE	CWE-480 , Use of incorrect operator

Bibliography

[Hatton 1995]	Section 2.7.2, "Errors of Omission and Addition"
-------------------------------	--

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/i4FtAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	27
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
null_statement_in_if	Null statement used as branch of an if
null_statement_in_loop	Null statement used as loop body

CertC-EXP19

Use braces for the body of an `if`, `for`, or `while` statement.

Input: IR

Source languages: C, C++

Details

Opening and closing braces for `if`, `for`, and `while` statements should always be used even if the statement's body contains only a single statement.

If an `if`, `while`, or `for` statement is used in a macro, the macro definition should not conclude with a semicolon. (See [PRE11-C. Do not conclude macro definitions with a semicolon](#).)

Braces improve the uniformity and readability of code. More important, when inserting an additional statement into a body containing only a single statement,

it is easy to forget to add braces because the indentation gives strong (but misleading) guidance to the structure.

Braces also help ensure that macros with multiple statements are properly expanded. Such a macro should be wrapped in a `do-while` loop. (See [PRE10-C. Wrap multistatement macros in a do-while loop](#).) However, when the `do-while` loop is not present, braces can still ensure that the macro expands as intended.

Noncompliant Code Example

This noncompliant code example uses an `if` statement without braces to authenticate a user:

```
int login;  
  
if (invalid_login())  
    login = 0;  
else  
    login = 1;
```

A developer might add a debugging statement to determine when the login is valid but forget to add opening and closing braces:

```
int login;  
  
if (invalid_login())  
    login = 0;  
else  
    printf("Login is valid\n"); /* Debugging line added here */  
    login = 1; /* This line always gets executed */  
    /* regardless of a valid login! */
```

Because of the indentation of the code, it is difficult to tell that the code will not function as intended by the programmer, potentially leading to a security breach.

Compliant Solution

In the compliant solution, opening and closing braces are used even when the body is a single statement:

```
int login;  
  
if (invalid_login()) {  
    login = 0;  
} else {  
    login = 1;  
}
```

Noncompliant Code Example

This noncompliant code example has an `if` statement nested in another `if` statement without braces around the `if` and `else` bodies:

```
int privileges;  
  
if (invalid_login())  
    if (allow_guests())  
        privileges = GUEST;  
    else  
        privileges = ADMINISTRATOR;
```

The indentation could lead the programmer to believe that a user is given administrator privileges only when the user's login is valid. However, the `else` statement actually attaches to the inner `if` statement:

```
int privileges;  
  
if (invalid_login())  
    if (allow_guests())  
        privileges = GUEST;  
    else  
        privileges = ADMINISTRATOR;
```

This is a security loophole: users with invalid logins can still obtain administrator privileges.

Compliant Solution

In the compliant solution, adding braces removes the ambiguity and ensures that privileges are correctly assigned:

```
int privileges;  
  
if (invalid_login()) {  
    if (allow_guests()) {  
        privileges = GUEST;  
    }  
} else {  
    privileges = ADMINISTRATOR;  
}
```

Risk Assessment

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
EXP19-C	Medium	Probable	Medium	P8	L2

MISRA C:2012	Rule 15.6 (required)
--------------	----------------------

Bibliography

[GNU 2010]	Coding Standards, Section 5.3, "Clean Use of C Constructs"
------------	--

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/1QGMAg>], Copyright [C] 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
dangerous_else	Block or if statement required as else branch.
dangerous_then	Block required as then branch.
loop_body_not_compound	Loop body shall be a statement sequence.

CertC-EXP20

Perform explicit tests to determine success, true and false, and equality.

Input: IR

Source languages: C, C++

Details

Perform explicit tests to determine success, true/false, and equality to improve the readability and maintainability of code and for compatibility with common conventions.

In particular, do not default the test for nonzero. For instance, suppose a `foo()` function returns 0 to indicate failure or a nonzero value to indicate success. Testing for inequality with 0,

```
if (foo() != 0) ...
```

is preferable to

```
if (foo()) ...
```

despite the convention that 0 indicates failure. Explicitly testing for inequality with 0 benefits maintainability if `foo()` is later modified to return -1 rather than 0 on failure.

This recommendation is derived from and considers the implications of the following common conventions:

1. Functions return 0 if false and nonzero if true [[StackOvflw 2009](#)].
2. Function failures can typically be indicated by -1 or any nonzero number.
3. Comparison functions (such as the standard library function `strcmp()`, which has a trinary return value) return 0 if the arguments are equal and nonzero otherwise (see [strcmp function](#)).

Noncompliant Code Example

In this noncompliant code example, `is_banned()` returns 0 if false and nonzero if true:

```
LinkedList bannedUsers;
int is_banned(User usr) {
    int x = 0;
    Node cur_node = (bannedUsers->head);
```

```

while (cur_node != NULL) {
    if(!strcmp((char *)cur_node->data, usr->name)) {
        x++;
    }
    cur_node = cur_node->next;
}

return x;
}

void processRequest(User usr) {
    if(is_banned(usr) == 1) {
        return;
    }
    serveResults();
}

```

If a banned user is listed twice, the user is granted access. Although `is_banned()` follows the common convention of returning nonzero for true, `processRequest` checks for equality only with 1.

Compliant Solution

Because most functions guarantee a return value of nonzero only for true, the preceding code is better written by checking for inequality with 0 (false), as follows:

```

LinkedList bannedUsers;

int is_banned(User usr) {
    int x = 0;

    Node cur_node = (bannedUsers->head);

    while(cur_node != NULL) {
        if (strcmp((char *)cur_node->data, usr->name)==0) {
            x++;
        }
        cur_node = cur_node->next;
    }

    return x;
}

void processRequest(User usr) {
    if (is_banned(usr) != 0) {
        return;
    }
    serveResults();
}

```

Noncompliant Code Example

In noncompliant code, function status can typically be indicated by returning -1 on failure or any nonnegative number on success. This is a common convention in the standard C library, but it is discouraged in [ERR02-C. Avoid in-band error indicators](#).

Although failures are frequently indicated by a return value of 0, some common conventions may conflict in the future with code in which the test for nonzero is not explicit. In this case, defaulting the test for nonzero welcomes bugs if and when a developer modifies `validateUser()` to return an error code or -1 rather than 0 to indicate a failure (all of which are also common conventions).

```

int validateUser(User usr) {
    if(listContains(validUsers, usr)) {
        return 1;
    }

    return 0;
}

void processRequest(User usr, Request request) {
    if(!validateUser(usr)) {
        return "invalid user";
    }
    else {
        serveResults();
    }
}

```

Although the code will work as intended, it is possible that a future modification will result in the following:

```

errno_t validateUser(User usr) {
    if(list_contains(allUsers, usr) == 0) {
        return 303; /* User not found error code */
    }
    if(list_contains(validUsers, usr) == 0) {
        return 304; /* Invalid user error code */
    }

    return 0;
}

void processRequest(User usr, Request request) {
    if(!validateUser(usr)) {
        return "invalid user";
    }
    else {
        serveResults();
    }
}

```

In this code, the programmer intended to add error code functionality to indicate the cause of a [validation](#) failure. The new code, however, validates any invalid or nonexistent user. Because there is no explicit test in `processRequest()`, the logical error is not obvious and seems correct by certain conventions.

Compliant Solution

This compliant code is preferable for improved maintenance. By defining what constitutes a failure and explicitly testing for it, the behavior is clearly implied, and future modifications are more likely to preserve it. If a future modification is made, such as in the previous example, it is immediately obvious that the `if` statement in `processRequest()` does not correctly utilize the specification of `validateUser()`.

```
int validateUser(User usr) {
    if(list_contains(validUsers, usr)) {
        return 1;
    }
    return 0;
}

void processRequest(User usr, Request request) {
    if(validateUser(usr) == 0) {
        return "invalid user";
    }
    else {
        serveResults();
    }
}
```

Noncompliant Code Example

Comparison functions (such as the standard library `strcmp()` function) return 0 if the arguments are equal and nonzero otherwise.

Because many comparison functions return 0 for equality and nonzero for inequality, they can cause confusion when used to test for equality. If someone were to switch the following `strcmp()` call with a function testing for equality, but the programmer did not follow the same convention as `strcmp()`, the programmer might instinctively just replace the function name. Also, when quickly reviewed, the code could easily appear to test for inequality.

```
void login(char *usr, char *pw) {
    User user = find_user(usr);
    if (!strcmp((user->password), pw)) {
        grantAccess();
    }
    else {
        denyAccess("Incorrect Password");
    }
}
```

The preceding code works correctly. However, to simplify the login code or to facilitate checking a user's password more than once, a programmer can separate the password-checking code from the login function in the following way:

```
int check_password(User *user, char *pw_given) {
    if (!strcmp((user->password), pw_given)) {
        return 1;
    }
    return 0;
}

void login(char *usr, char *pw) {
    User user = find_user(usr);
    if (!check_password(user, pw)) {
        grantAccess();
    }
    else {
        denyAccess("Incorrect Password");
    }
}
```

In an attempt to leave the previous logic intact, the developer just replaces `strcmp()` with a call to the new function. However, doing so produces incorrect behavior. In this case, any user who inputs an incorrect password is granted access. Again, two conventions conflict and produce code that is easily corrupted when modified. To make code maintainable and to avoid these conflicts, such a result should never be defaulted.

Compliant Solution

This compliant solution, using a comparison function for this purpose, is the preferred approach. By performing an explicit test, any programmer who wishes to modify the equality test can clearly see the implied behavior and convention that is being followed.

```
void login(char *usr, char *pw) {
    User user = find_user(usr);
    if (strcmp((user->password), pw) == 0) {
        grantAccess();
    }
    else {
        denyAccess("Incorrect Password");
    }
}
```

Risk Assessment

Code that does not conform to the common practices presented is difficult to maintain. Bugs can easily arise when modifying helper functions that evaluate true/false or success/failure. Bugs can also easily arise when modifying code that tests for equality using a comparison function that obeys the same conventions as standard library functions such as `strcmp`.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
EXP20-C	Medium	Probable	Low	P12	L1

Bibliography

[StackOvflw 2009]	"Should I Return TRUE/FALSE Values from a C Function?"
Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [https://www.securecoding.cert.org/confluence/x/GgCVAg], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.	

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
only_in_conditions	If True, only logical operators inside conditions are checked.	False
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
nonbool_if_condition	Condition should have essentially Boolean type
nonbool_logical_operator_operand	Operand of logical operator shall be effectively boolean

CertC-EXP30

Do not depend on the order of evaluation for side effects.

Input: IR

Source languages: C, C++

Details

Evaluation of an expression may produce [side effects](#). At specific points during execution, known as [sequence points](#), all side effects of previous evaluations are complete, and no side effects of subsequent evaluations have yet taken place. Do not depend on the order of evaluation for side effects unless there is an intervening sequence point.

The C Standard, 6.5, paragraph 2 [[ISO/IEC 9899:2011](#)], states

If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. If there are multiple allowable orderings of the subexpressions of an expression, the behavior is undefined if such an unsequenced side effect occurs in any of the orderings.

This requirement must be met for each allowable ordering of the subexpressions of a full expression; otherwise, the behavior is [undefined](#). (See [undefined behavior 35](#).)

The following sequence points are defined in the C Standard, Annex C [[ISO/IEC 9899:2011](#)]:

- Between the evaluations of the function designator and actual arguments in a function call and the actual call
- Between the evaluations of the first and second operands of the following operators:
 - Logical AND: &&
 - Logical OR: ||
 - Comma: ,
- Between the evaluations of the first operand of the conditional ?: operator and whichever of the second and third operands is evaluated
- The end of a full declarator
- Between the evaluation of a full expression and the next full expression to be evaluated; the following are full expressions:
 - An initializer that is not part of a compound literal
 - The expression in an expression statement
 - The controlling expression of a selection statement (if or switch)
 - The controlling expression of a while or do statement
 - Each of the (optional) expressions of a for statement
 - The (optional) expression in a return statement
- Immediately before a library function returns
- After the actions associated with each formatted input/output function conversion specifier

- Immediately before and immediately after each call to a comparison function, and also between any call to a comparison function and any movement of the objects passed as arguments to that call

This rule means that statements such as

```
i = i + 1;
a[i] = i;
```

have defined behavior, and statements such as the following do not:

```
/* i is modified twice between sequence points */
i = ++i + 1;

/* i is read other than to determine the value to be stored */
a[i++] = i;
```

Not all instances of a comma in C code denote a usage of the comma operator. For example, the comma between arguments in a function call is not a sequence point. However, according to the C Standard, 6.5.2.2, paragraph 10 [ISO/IEC 9899:2011]

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.

This rule means that the order of evaluation for function call arguments is unspecified and can happen in any order.

Noncompliant Code Example

Programs cannot safely rely on the order of evaluation of operands between sequence points. In this noncompliant code example, `i` is evaluated twice without an intervening sequence point, so the behavior of the expression is [undefined](#):

```
#include <stdio.h>

void func(int i, int *b) {
    int a = i + b[i];
    printf("%d, %d", a, i);
}
```

Compliant Solution

These examples are independent of the order of evaluation of the operands and can be interpreted in only one way:

```
#include <stdio.h>

void func(int i, int *b) {
    int a;
    ++i;
    a = i + b[i];
    printf("%d, %d", a, i);
}
```

Alternatively:

```
#include <stdio.h>

void func(int i, int *b) {
    int a = i + b[i + 1];
    ++i;
    printf("%d, %d", a, i);
}
```

Noncompliant Code Example

The call to `func()` in this noncompliant code example has [undefined behavior](#) because there is no sequence point between the argument expressions:

```
extern void func(int i, int j);

void f(int i) {
    func(i++, i);
}
```

The first (left) argument expression reads the value of `i` (to determine the value to be stored) and then modifies `i`. The second (right) argument expression reads the value of `i` between the same pair of sequence points as the first argument, but not to determine the value to be stored in `i`. This additional attempt to read the value of `i` has undefined behavior.

Compliant Solution

This compliant solution is appropriate when the programmer intends for both arguments to `func()` to be equivalent:

```
extern void func(int i, int j);

void f(int i) {
    i++;
    func(i, i);
}
```

This compliant solution is appropriate when the programmer intends for the second argument to be 1 greater than the first:

```
extern void func(int i, int j);

void f(int i) {
```

```

int j = i++;
func(j, i);
}

```

Noncompliant Code Example

The order of evaluation for function arguments is unspecified. This noncompliant code example exhibits [unspecified behavior](#) but not [undefined behavior](#):

```

extern void c(int i, int j);
int glob;

int a(void) {
    return glob + 10;
}

int b(void) {
    glob = 42;
    return glob;
}

void func(void) {
    c(a(), b());
}

```

It is unspecified what order `a()` and `b()` are called in; the only guarantee is that both `a()` and `b()` will be called before `c()` is called. If `a()` or `b()` rely on shared state when calculating their return value, as they do in this example, the resulting arguments passed to `c()` may differ between compilers or architectures.

Compliant Solution

In this compliant solution, the order of evaluation for `a()` and `b()` is fixed, and so no unspecified behavior occurs:

```

extern void c(int i, int j);
int glob;

int a(void) {
    return glob + 10;
}
int b(void) {
    glob = 42;
    return glob;
}

void func(void) {
    int a_val, b_val;

    a_val = a();
    b_val = b();

    c(a_val, b_val);
}

```

Risk Assessment

Attempting to modify an object multiple times between sequence points may cause that object to take on an unexpected value, which can lead to [unexpected program behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP30-C	Medium	Probable	Medium	P8	L2

Related Guidelines

SEI CERT C++ Coding Standard	EXP50-CPP. Do not depend on the order of evaluation for side effects
CERT Oracle Secure Coding Standard for Java	EXP05-J. Do not follow a write by a subsequent write or read of the same object within an expression
ISO/IEC TR 24772:2013	Operator Precedence/Order of Evaluation [JCW] Side-effects and Order of Evaluation [SAM]
MISRA C:2012	Rule 12.1 (advisory)

Bibliography

[ISO/IEC 9899:2011]	6.5, "Expressions" 6.5.2.2, "Function Calls" Annex C, "Sequence Points"
[Saks 2007]	
[Summit 2005]	Questions 3.1, 3.2, 3.3, 3.3b, 3.7, 3.8, 3.9, 3.10a, 3.10b, and 3.11

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/ZwE>], Copyright [C] 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
ignore_exceptions	Ignore possible exceptions in the computation of side effects. (this option only has an effect if report_calls is enabled)	False
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
report_calls	If True, reports unsequenced function calls.	True

Possible Messages

Name	Message
unsequenced_call_volatile	Function calls unsequenced with volatile access
unsequenced_calls	Unsequenced function calls
unsequenced_read_write	Unsequenced read and write accesses
unsequenced_volatiles	Unsequenced volatile accesses
unsequenced_writes	Unsequenced writes

CertC-EXP32

Do not access a volatile object through a nonvolatile reference.

Input: IR

Source languages: C, C++

Details

An object that has volatile-qualified type may be modified in ways unknown to the [implementation](#) or have other unknown [side effects](#). Referencing a volatile object by using a non-volatile lvalue is [undefined behavior](#). The C Standard, 6.7.3 [[ISO/IEC 9899:2011](#)], states

If an attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type, the behavior is undefined.

See [undefined behavior 65](#).

Noncompliant Code Example

In this noncompliant code example, a volatile object is accessed through a non-volatile-qualified reference, resulting in [undefined behavior](#):

```
#include <stdio.h>

void func(void) {
    static volatile int **ipp;
    static int *ip;
    static volatile int i = 0;

    printf("i = %d.\n", i);

    ip = &ip; /* May produce a warning diagnostic */
    ip = (int**) &ip; /* Constraint violation; may produce a warning diagnostic */
    *ipp = &i; /* Valid */
    if (*ip != 0) { /* Valid */
        /* ... */
    }
}
```

The assignment `ip = &ip` is not safe because it allows the valid code that follows to reference the value of the volatile object `i` through the non-volatile-qualified reference `ip`. In this example, the compiler may optimize out the entire `if` block because `*ip != 0` must be false if the object to which `ip` points is not volatile.

Implementation Details

This example compiles without warning on Microsoft Visual Studio 2013 when compiled in C mode (`/TC`) but causes errors when compiled in C++ mode (`/TP`).

GCC 4.8.1 generates a warning but compiles successfully.

Compliant Solution

In this compliant solution, `ip` is declared `volatile`:

```
#include <stdio.h>

void func(void) {
    static volatile int **ipp;
    static volatile int *ip;
    static volatile int i = 0;

    printf("i = %d.\n", i);

    ipp = &ip;
    *ipp = &i;
    if (*ip != 0) {
        /* ... */
    }
}
```

Risk Assessment

Accessing an object with a volatile-qualified type through a reference with a non-volatile-qualified type is [undefined behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP32-C	Low	Likely	Medium	P6	L2

Related Guidelines

ISO/IEC TR 24772:2013	Pointer Casting and Pointer Type Changes [HFC] Type System [IHN]
MISRA C:2012	Rule 11.8 (required)
SEI CERT C++ Coding Standard	EXP55-CPP. Do not access a cv-qualified object through a cv-unqualified type

Bibliography

[ISO/IEC 9899:2011]	6.7.3, "Type Qualifiers"
-------------------------------------	--------------------------

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/hAY>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
check_const	Whether casts affecting const-qualifiers should be reported	False
check_volatile	Whether casts affecting volatile-qualifiers should be reported	True
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
only_top_level	Whether the check only applies to the top-level pointee or recursively to deeper pointer levels as well	False
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
cast_adds_const	Cast adds const qualification
cast_adds_volatile	Cast adds volatile qualification
cast_removes_const	Cast removes const qualification
cast_removes_volatile	Cast removes volatile qualification
implicit_cast_adds_const	Implicit cast adds const qualification
implicit_cast_adds_volatile	Implicit cast adds volatile qualification
implicit_cast_removes_const	Implicit cast removes const qualification
implicit_cast_removes_volatile	Implicit cast removes volatile qualification

CertC-EXP33

Do not read uninitialized memory.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Local, automatic variables assume unexpected values if they are read before they are initialized. The C Standard, 6.7.9, paragraph 10, specifies [[ISO/IEC 9899:2011](#)]

If an object that has automatic storage duration is not initialized explicitly, its value is [indeterminate](#).

See [undefined behavior 11](#).

When local, automatic variables are stored on the program stack, for example, their values default to whichever values are currently stored in stack memory.

Additionally, some dynamic memory allocation functions do not initialize the contents of the memory they allocate.

Function	Initialization
aligned_alloc()	Does not perform initialization
calloc()	Zero-initializes allocated memory
malloc()	Does not perform initialization
realloc()	Copies contents from original pointer; may not initialize all memory

Uninitialized automatic variables or dynamically allocated memory has [indeterminate values](#), which for objects of some types, can be a [trap representation](#). Reading such trap representations is [undefined behavior](#); it can cause a program to behave in an [unexpected](#) manner and provide an avenue for attack. (See [undefined behavior 10](#) and [undefined behavior 12](#).) In many cases, compilers issue a warning diagnostic message when reading uninitialized variables. (See [MSC00-C. Compile cleanly at high warning levels](#) for more information.)

Noncompliant Code Example (Return-by-Reference)

In this noncompliant code example, the `set_flag()` function is intended to set the parameter, `sign_flag`, to the sign of `number`. However, the programmer neglected to account for the case where `number` is equal to 0. Because the local variable `sign` is uninitialized when calling `set_flag()` and is never written to by `set_flag()`, the comparison operation exhibits [undefined behavior](#) when reading `sign`.

```
void set_flag(int number, int *sign_flag) {
    if (NULL == sign_flag) {
        return;
    }

    if (number > 0) {
        *sign_flag = 1;
    } else if (number < 0) {
        *sign_flag = -1;
    }
}

int is_negative(int number) {
    int sign;
    set_flag(number, &sign);
    return sign < 0;
}
```

Some compilers assume that when the address of an uninitialized variable is passed to a function, the variable is initialized within that function. Because compilers frequently fail to diagnose any resulting failure to initialize the variable, the programmer must apply additional scrutiny to ensure the correctness of the code.

This defect results from a failure to consider all possible data states. (See [MSC01-C. Strive for logical completeness](#) for more information.)

Compliant Solution (Return-by-Reference)

This compliant solution trivially repairs the problem by accounting for the possibility that `number` can be equal to 0.

Although compilers and [static analysis](#) tools often detect uses of uninitialized variables when they have access to the source code, diagnosing the problem is difficult or impossible when either the initialization or the use takes place in object code for which the source code is inaccessible. Unless doing so is prohibitive for performance reasons, an additional defense-in-depth practice worth considering is to initialize local variables immediately after declaration.

```
void set_flag(int number, int *sign_flag) {
    if (NULL == sign_flag) {
        return;
    }

    /* Account for number being 0 */
    if (number >= 0) {
        *sign_flag = 1;
    } else {
        *sign_flag = -1;
    }
}

int is_negative(int number) {
    int sign = 0; /* Initialize for defense-in-depth */
    set_flag(number, &sign);
    return sign < 0;
}
```

Noncompliant Code Example (Uninitialized Local)

In this noncompliant code example, the programmer mistakenly fails to set the local variable `error_log` to the `msg` argument in the `report_error()` function [[Mercy 2006](#)]. Because `error_log` has not been initialized, an [indeterminate value](#) is read. The `sprintf()` call copies data from the arbitrary location pointed to by the indeterminate `error_log` variable until a null byte is reached, which can result in a buffer overflow.

```
#include <stdio.h>

/* Get username and password from user, return -1 on error */
extern int do_auth(void);
enum { BUFFERSIZE = 24 };
void report_error(const char *msg) {
    const char *error_log;
    char buffer[BUFFERSIZE];

    sprintf(buffer, "Error: %s", error_log);
    printf("%s\n", buffer);
}

int main(void) {
    if (do_auth() == -1) {
        report_error("Unable to login");
    }
    return 0;
}
```

Noncompliant Code Example (Uninitialized Local)

In this noncompliant code example, the `report_error()` function has been modified so that `error_log` is properly initialized:

```
#include <stdio.h>
enum { BUFFERSIZE = 24 };
void report_error(const char *msg) {
    const char *error_log = msg;
    char buffer[BUFFERSIZE];

    sprintf(buffer, "Error: %s", error_log);
    printf("%s\n", buffer);
}
```

This example remains problematic because a buffer overflow will occur if the null-terminated byte string referenced by `msg` is greater than 17 characters, including the null terminator. (See [STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator](#) for more information.)

Compliant Solution (Uninitialized Local)

In this compliant solution, the buffer overflow is eliminated by calling the `snprintf()` function:

```
#include <stdio.h>
enum { BUFFERSIZE = 24 };
void report_error(const char *msg) {
    char buffer[BUFFERSIZE];

    if (0 < snprintf(buffer, BUFFERSIZE, "Error: %s", msg))
        printf("%s\n", buffer);
    else
        puts("Unknown error");
}
```

Compliant Solution (Uninitialized Local)

A less error-prone compliant solution is to simply print the error message directly instead of using an intermediate buffer:

```
#include <stdio.h>

void report_error(const char *msg) {
    printf("Error: %s\n", msg);
}
```

Noncompliant Code Example (`mbstate_t`)

In this noncompliant code example, the function `mbrlen()` is passed the address of an automatic `mbstate_t` object that has not been properly initialized. This is [undefined behavior 200](#) because `mbrlen()` dereferences and reads its third argument.

```
#include <string.h>
#include <wchar.h>

void func(const char *mbs) {
    size_t len;
    mbstate_t state;

    len = mbrlen(mbs, strlen(mbs), &state);
}
```

Compliant Solution (`mbstate_t`)

Before being passed to a multibyte conversion function, an `mbstate_t` object must be either initialized to the initial conversion state or set to a value that corresponds to the most recent shift state by a prior call to a multibyte conversion function. This compliant solution sets the `mbstate_t` object to the initial conversion state by setting it to all zeros:

```
#include <string.h>
#include <wchar.h>

void func(const char *mbs) {
    size_t len;
    mbstate_t state;

    memset(&state, 0, sizeof(state));
    len = mbrlen(mbs, strlen(mbs), &state);
}
```

Noncompliant Code Example (POSIX, Entropy)

In this noncompliant code example described in "[More Randomness or Less](#)" [Wang 2012], the process ID, time of day, and uninitialized memory `junk` is used to seed a random number generator. This behavior is characteristic of some distributions derived from Debian Linux that use uninitialized memory as a source of entropy because the value stored in `junk` is indeterminate. However, because accessing an [indeterminate value](#) is [undefined behavior](#), compilers may optimize out the uninitialized variable access completely, leaving only the time and process ID and resulting in a loss of desired entropy.

```
#include <time.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/time.h>

void func(void) {
    struct timeval tv;
    unsigned long junk;

    gettimeofday(&tv, NULL);
    srand((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
}
```

In security protocols that rely on unpredictability, such as RSA encryption, a loss in entropy results in a less secure system.

Compliant Solution (POSIX, Entropy)

This compliant solution seeds the random number generator by using the CPU clock and the real-time clock instead of reading uninitialized memory:

```
#include <time.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/time.h>

void func(void) {
    double cpu_time;
    struct timeval tv;

    cpu_time = ((double) clock()) / CLOCKS_PER_SEC;
    gettimeofday(&tv, NULL);
    srand((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ cpu_time);
}
```

Noncompliant Code Example (`realloc()`)

The `realloc()` function changes the size of a dynamically allocated memory object. The initial `size` bytes of the returned memory object are unchanged, but any newly added space is uninitialized, and its value is [indeterminate](#). As in the case of `malloc()`, accessing memory beyond the size of the original object is [undefined behavior 181](#).

It is the programmer's responsibility to ensure that any memory allocated with `malloc()` and `realloc()` is properly initialized before it is used.

In this noncompliant code example, an array is allocated with `malloc()` and properly initialized. At a later point, the array is grown to a larger size but not initialized beyond what the original array contained. Subsequently accessing the uninitialized bytes in the new array is undefined behavior.

```
#include <stdlib.h>
```

```

#include <stdio.h>
enum { OLD_SIZE = 10, NEW_SIZE = 20 };

int *resize_array(int *array, size_t count) {
    if (0 == count) {
        return 0;
    }

    int *ret = (int *)realloc(array, count * sizeof(int));
    if (!ret) {
        free(array);
        return 0;
    }

    return ret;
}

void func(void) {
    int *array = (int *)malloc(OLD_SIZE * sizeof(int));
    if (0 == array) {
        /* Handle error */
    }

    for (size_t i = 0; i < OLD_SIZE; ++i) {
        array[i] = i;
    }

    array = resize_array(array, NEW_SIZE);
    if (0 == array) {
        /* Handle error */
    }

    for (size_t i = 0; i < NEW_SIZE; ++i) {
        printf("%d ", array[i]);
    }
}

```

Compliant Solution (`realloc()`)

In this compliant solution, the `resize_array()` helper function takes a second parameter for the old size of the array so that it can initialize any newly allocated elements:

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

enum { OLD_SIZE = 10, NEW_SIZE = 20 };

int *resize_array(int *array, size_t old_count, size_t new_count) {
    if (0 == new_count) {
        return 0;
    }

    int *ret = (int *)realloc(array, new_count * sizeof(int));
    if (!ret) {
        free(array);
        return 0;
    }

    if (new_count > old_count) {
        memset(ret + old_count, 0, (new_count - old_count) * sizeof(int));
    }

    return ret;
}

void func(void) {
    int *array = (int *)malloc(OLD_SIZE * sizeof(int));
    if (0 == array) {
        /* Handle error */
    }

    for (size_t i = 0; i < OLD_SIZE; ++i) {
        array[i] = i;
    }

    array = resize_array(array, OLD_SIZE, NEW_SIZE);
    if (0 == array) {
        /* Handle error */
    }

    for (size_t i = 0; i < NEW_SIZE; ++i) {
        printf("%d ", array[i]);
    }
}

```

Exceptions

EXP33-C-EX1: Reading uninitialized memory by an [lvalue](#) of type `unsigned char` does not trigger [undefined behavior](#). The `unsigned char` type is defined to not have a trap representation, which allows for moving bytes without knowing if they are initialized. (See the C Standard, 6.2.6.1, paragraph 3.) However, on some architectures, such as the Intel Itanium, registers have a bit to indicate whether or not they have been initialized. The C Standard, 6.3.2.1, paragraph 2, allows such [implementations](#) to cause a trap for an object that never had its address taken and is stored in a register if such an object is referred to in any way.

Reading uninitialized variables is [undefined behavior](#) and can result in [unexpected program behavior](#). In some cases, these [security flaws](#) may allow the execution of arbitrary code.

Reading uninitialized variables for creating entropy is problematic because these memory accesses can be removed by compiler optimization. [VU#925211](#) is an example of a [vulnerability](#) caused by this coding error.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP33-C	High	Probable	Medium	P12	L1

Related Guidelines

CERT C Secure Coding Standard	MSC00-C. Compile cleanly at high warning levels MSC01-C. Strive for logical completeness
SEI CERT C++ Coding Standard	EXP53-CPP. Do not read uninitialized memory
ISO/IEC TR 24772:2013	Initialization of Variables [LAV]
ISO/IEC TS 17961	Referencing uninitialized memory [uninitref]
MITRE CWE	CWE-119 , Improper Restriction of Operations within the Bounds of a Memory Buffer CWE-123 , Write-what-where Condition CWE-125 , Out-of-bounds Read CWE-665 , Improper Initialization

Bibliography

[Flake 2006]	
[ISO/IEC 9899:2011]	Subclause 6.7.9, "Initialization" Subclause 6.2.6.1, "General" Subclause 6.3.2.1, "Lvalues, Arrays, and Function Designators"
[Mercy 2006]	
[VU#925211]	
[Wang 2012]	"More Randomness or Less"
[xorl 2009]	"CVE-2009-1888: SAMBA ACLs Uninitialized Memory Read"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/4gE>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
init_functions	Names of functions to be inspected as well when called directly from constructor.	('Init', 'init')
inspect_directly_called_methods	Inspect all methods directly called from constructor.	False
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
only_member_initializer_list	Only inspect member initializer list and not the constructor body/methods.	False
perform_global_analysis	Whether global analysis (using iranalysis) should be done, or just a local analysis.	True
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
report_missing_field_constructors	Enables detection of constructors which rely on implicit field constructor calls.	False

Possible Messages

Name	Message
implicit_field_init	Field is only implicitly initialized in constructor.
missing_base_class_init	Base class is not explicitly initialized in constructor.
missing_field_init	Field is not initialized in constructor.
pass_as_pointer_to_const_param	Passing uninitialized variable by pointer as function parameter with pointer-to-const type
possible_uninit	Use of possibly uninitialized variable
uninit	Use of uninitialized variable

CertC-EXP34

Do not dereference null pointers

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Dereferencing a null pointer is [undefined behavior](#).

On many platforms, dereferencing a null pointer results in [abnormal program termination](#), but this is not required by the standard. See "[Clever Attack Exploits Fully-Patched Linux Kernel](#)" [Goodin 2009] for an example of a code execution [exploit](#) that resulted from a null pointer dereference.

Noncompliant Code Example

This noncompliant code example is derived from a real-world example taken from a vulnerable version of the `libpng` library as deployed on a popular ARM-based cell phone [Jack 2007]. The `libpng` library allows applications to read, create, and manipulate PNG (Portable Network Graphics) raster image files. The `libpng` library implements its own wrapper to `malloc()` that returns a null pointer on error or on being passed a 0-byte-length argument.

This code also violates [ERR33-C. Detect and handle standard library errors](#).

```
#include <png.h> /* From libpng */
#include <string.h>

void func(png_structp png_ptr, int length, const void *user_data) {
    png_charp chunkdata;
    chunkdata = (png_charp)png_malloc(png_ptr, length + 1);
    /* ... */
    memcpy(chunkdata, user_data, length);
    /* ... */
}
```

If `length` has the value -1, the addition yields 0, and `png_malloc()` subsequently returns a null pointer, which is assigned to `chunkdata`. The `chunkdata` pointer is later used as a destination argument in a call to `memcpy()`, resulting in user-defined data overwriting memory starting at address 0. In the case of the ARM and XScale architectures, the `0x0` address is mapped in memory and serves as the exception vector table; consequently, dereferencing `0x0` did not cause an [abnormal program termination](#).

Compliant Solution

This compliant solution ensures that the pointer returned by `png_malloc()` is not null. It also uses the unsigned type `size_t` to pass the `length` parameter, ensuring that negative values are not passed to `func()`.

```
#include <png.h> /* From libpng */
#include <string.h>

void func(png_structp png_ptr, size_t length, const void *user_data) {
    png_charp chunkdata;
    if (length == SIZE_MAX) {
        /* Handle error */
    }
    chunkdata = (png_charp)png_malloc(png_ptr, length + 1);
    if (NULL == chunkdata) {
        /* Handle error */
    }
    /* ... */
    memcpy(chunkdata, user_data, length);
    /* ... */
}
```

Noncompliant Code Example

In this noncompliant code example, `input_str` is copied into dynamically allocated memory referenced by `c_str`. If `malloc()` fails, it returns a null pointer that is assigned to `c_str`. When `c_str` is dereferenced in `memcpy()`, the program exhibits [undefined behavior](#). Additionally, if `input_str` is a null pointer, the call to `strlen()` dereferences a null pointer, also resulting in undefined behavior. This code also violates [ERR33-C. Detect and handle standard library errors](#).

```
#include <string.h>
#include <stdlib.h>

void f(const char *input_str) {
    size_t size = strlen(input_str) + 1;
    char *c_str = (char *)malloc(size);
    memcpy(c_str, input_str, size);
    /* ... */
    free(c_str);
    c_str = NULL;
    /* ... */
}
```

Compliant Solution

This compliant solution ensures that both `input_str` and the pointer returned by `malloc()` are not null:

```
#include <string.h>
#include <stdlib.h>

void f(const char *input_str) {
    size_t size;
    char *c_str;

    if (NULL == input_str) {
        /* Handle error */
    }

    size = strlen(input_str) + 1;
    c_str = (char *)malloc(size);
    if (NULL == c_str) {
        /* Handle error */
    }
    memcpy(c_str, input_str, size);
    /* ... */
    free(c_str);
    c_str = NULL;
    /* ... */
}
```

Noncompliant Code Example

This noncompliant code example is from a version of `drivers/net/tun.c` and affects Linux kernel 2.6.30 [[Goodin 2009](#)]:

```
static unsigned int tun_chr_poll(struct file *file, poll_table *wait) {
    struct tun_file *tfile = file->private_data;
    struct tun_struct *tun = __tun_get(tfile);
    struct sock *sk = tun->sk;
    unsigned int mask = 0;

    if (!tun)
        return POLLERR;

    DBG(KERN_INFO "%s: tun_chr_poll\n", tun->dev->name);

    poll_wait(file, &tun->socket.wait, wait);

    if (!skb_queue_empty(&tun->readq))
        mask |= POLLIN | POLLRDNORM;

    if (sock_writeable(sk) ||
        (!test_and_set_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags) &&
         sock_writeable(sk)))
        mask |= POLLOUT | POLLWRNORM;

    if (tun->dev->reg_state != NETREG_REGISTERED)
        mask = POLLERR;

    tun_put(tun);
    return mask;
}
```

The `sk` pointer is initialized to `tun->sk` before checking if `tun` is a null pointer. Because null pointer dereferencing is [undefined behavior](#), the compiler (GCC in this case) can optimize away the `if (!tun)` check because it is performed after `tun->sk` is accessed, implying that `tun` is non-null. As a result, this noncompliant code example is vulnerable to a null pointer dereference exploit, because null pointer dereferencing can be permitted on several platforms, for example, by using `mmap(2)` with the `MAP_FIXED` flag on Linux and Mac OS X, or by using the `shmat()` POSIX function with the `SHM_RND` flag [[Liu 2009](#)].

Compliant Solution

This compliant solution eliminates the null pointer deference by initializing `sk` to `tun->sk` following the null pointer check:

```
static unsigned int tun_chr_poll(struct file *file, poll_table *wait) {
    struct tun_file *tfile = file->private_data;
    struct tun_struct *tun = __tun_get(tfile);
    struct sock *sk;
    unsigned int mask = 0;

    if (!tun)
        return POLLERR;

    sk = tun->sk;

    /* The remaining code is omitted because it is unchanged... */
}
```

Risk Assessment

Dereferencing a null pointer is [undefined behavior](#), typically [abnormal program termination](#). In some situations, however, dereferencing a null pointer can lead to the execution of arbitrary code [[Jack 2007](#), [van Sprundel 2006](#)]. The indicated severity is for this more severe case; on platforms where it is not possible to exploit a null pointer dereference to execute arbitrary code, the actual severity is low.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP34-C	High	Likely	Medium	P18	L1

Related Guidelines

CERT Oracle Secure Coding Standard for Java	EXP01-J. Do not use a null in a case where an object is required
ISO/IEC TR 24772:2013	Pointer Casting and Pointer Type Changes [HFC] Null Pointer Dereference [XYH]
ISO/IEC TS 17961	Dereferencing an out-of-domain pointer [nullref]
MITRE CWE	CWE-476 , NULL Pointer Dereference

Bibliography

[Goodin 2009]	
[Jack 2007]	
[Liu 2009]	
[van Sprundel 2006]	
[Viega 2005]	Section 5.2.18, "Null-Pointer Dereference"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/PAw>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
null_deref	Pointer is NULL at dereference
possible_null_deref	Pointer may be NULL at dereference

CertC-EXP35

Do not modify objects with temporary lifetime.

Input: IR

Source languages: C, C++

Details

The C11 Standard [[ISO/IEC 9899:2011](#)] introduced a new term: *temporary lifetime*. Modifying an object with temporary lifetime is [undefined behavior](#). According to subclause 6.2.4, paragraph 8

A non-lvalue expression with structure or union type, where the structure or union contains a member with array type (including, recursively, members of all contained structures and unions) refers to an object with automatic storage duration and *temporary* lifetime. Its lifetime begins when the expression is evaluated and its initial value is the value of the expression. Its lifetime ends when the evaluation of the containing full expression or full declarator ends. Any attempt to modify an object with temporary lifetime results in undefined behavior.

This definition differs from the C99 Standard (which defines modifying the result of a function call or accessing it after the next sequence point as undefined behavior) because a temporary object's lifetime ends when the evaluation containing the full expression or full declarator ends, so the result of a function call can be accessed. This extension to the lifetime of a temporary also removes a quiet change to C90 and improves compatibility with C++.

C functions may not return arrays; however, functions can return a pointer to an array or a `struct` or `union` that contains arrays. Consequently, if a function call returns by value a `struct` or `union` containing an array, do not modify those arrays within the expression containing the function call. Do not access an array returned by a function after the next sequence point or after the evaluation of the containing full expression or full declarator ends.

Noncompliant Code Example (C99)

This noncompliant code example [conforms](#) to the C11 Standard; however, it fails to conform to C99. If compiled with a C99-conforming implementation, this code has [undefined behavior](#) because the sequence point preceding the call to `printf()` comes between the call and the access by `printf()` of the string in the returned object.

```
#include <stdio.h>

struct X { char a[8]; };

struct X salutation(void) {
    struct X result = { "Hello" };
    return result;
}

struct X addressee(void) {
    struct X result = { "world" };
    return result;
}

int main(void) {
    printf("%s, %s!\n", salutation().a, addressee().a);
    return 0;
}
```

Compliant Solution

This compliant solution stores the structures returned by the call to `addressee()` before calling the `printf()` function. Consequently, this program conforms to both C99 and C11.

```
#include <stdio.h>

struct X { char a[8]; };

struct X salutation(void) {
    struct X result = { "Hello" };
    return result;
}

struct X addressee(void) {
    struct X result = { "world" };
    return result;
}

int main(void) {
    struct X my_salutation = salutation();
    struct X my_addressee = addressee();

    printf("%s, %s!\n", my_salutation.a, my_addressee.a);
    return 0;
}
```

Noncompliant Code Example

This noncompliant code example attempts to retrieve an array and increment the array's first value. The array is part of a `struct` that is returned by a function call. Consequently, the array has temporary lifetime, and modifying the array is [undefined behavior](#).

```
#include <stdio.h>

struct X { int a[6]; };

struct X addressee(void) {
    struct X result = { { 1, 2, 3, 4, 5, 6 } };
    return result;
}

int main(void) {
    printf("%x", ++(addressee().a[0]));
    return 0;
}
```

Compliant Solution

This compliant solution stores the structure returned by the call to `addressee()` as `my_x` before calling the `printf()` function. When the array is modified, its lifetime is no longer temporary but matches the lifetime of the block in `main()`.

```
#include <stdio.h>

struct X { int a[6]; };

struct X addressee(void) {
```

```

    struct X result = { { 1, 2, 3, 4, 5, 6 } };
    return result;
}

int main(void) {
    struct X my_x = addressee();
    printf("%x", ++(my_x.a[0]));
    return 0;
}

```

Risk Assessment

Attempting to modify an array or access it after its lifetime expires may result in erroneous program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP35-C	Low	Probable	Medium	P4	L3

Related Guidelines

ISO/IEC TR 24772:2013	Dangling References to Stack Frames [DCM] Side-effects and Order of Evaluation [SAM]
---------------------------------------	---

Bibliography

[ISO/IEC 9899:2011]	6.2.4, "Storage Durations of Objects"
-------------------------------------	---------------------------------------

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/pYEt>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
temp_array_decay	Temporary array converted to pointer
temp_mutation	Mutation of temporary array

CertC-EXP36

Do not cast pointers into more strictly aligned pointer types.

Input: IR

Source languages: C, C++

Details

Do not convert a pointer value to a pointer type that is more strictly aligned than the referenced type. Different alignments are possible for different types of objects. If the type-checking system is overridden by an explicit cast or the pointer is converted to a void pointer (`void *`) and then to a different type, the alignment of an object may be changed.

The C Standard, 6.3.2.3, paragraph 7 [[ISO/IEC 9899:2011](#)], states

A pointer to an object or incomplete type may be converted to a pointer to a different object or incomplete type. If the resulting pointer is not correctly aligned for the referenced type, the behavior is undefined.

See [undefined behavior 25](#).

If the misaligned pointer is dereferenced, the program may [terminate abnormally](#). On some architectures, the cast alone may cause a loss of information even if the value is not dereferenced if the types involved have differing alignment requirements.

Noncompliant Code Example

In this noncompliant example, the `char` pointer `&c` is converted to the more strictly aligned `int` pointer `ip`. On some implementations, `cp` will not match `&c`. As a result, if a pointer to one object type is converted to a pointer to a different object type, the second object type must not require stricter alignment than the first.

```
#include <assert.h>

void func(void) {
    char c = 'x';
    int *ip = (int *)&c; /* This can lose information */
    char *cp = (char *)ip;

    /* Will fail on some conforming implementations */
    assert(cp == &c);
}
```

Compliant Solution (Intermediate Object)

In this compliant solution, the `char` value is stored into an object of type `int` so that the pointer's value will be properly aligned:

```
#include <assert.h>

void func(void) {
    char c = 'x';
    int i = c;
    int *ip = &i;

    assert(ip == &i);
}
```

Noncompliant Code Example

The C Standard allows any object pointer to be cast to and from `void *`. As a result, it is possible to silently convert from one pointer type to another without the compiler diagnosing the problem by storing or casting a pointer to `void *` and then storing or casting it to the final type. In this noncompliant code example, `loop_function()` is passed the `char` pointer `loop_ptr` but returns an object of type `int` pointer:

```
int *loop_function(void *v_pointer) {
    /* ... */
    return v_pointer;
}

void func(char *loop_ptr) {
    int *int_ptr = loop_function(loop_ptr);

    /* ... */
}
```

This example compiles without warning using GCC 4.8 on Ubuntu Linux 14.04. However, `v_pointer` can be more strictly aligned than an object of type `int *`.

Compliant Solution

Because the input parameter directly influences the return value, and `loop_function()` returns an object of type `int *`, the formal parameter `v_pointer` is redeclared to accept only an object of type `int *`:

```
int *loop_function(int *v_pointer) {
    /* ... */
    return v_pointer;
}

void func(int *loop_ptr) {
    int *int_ptr = loop_function(loop_ptr);

    /* ... */
}
```

Noncompliant Code Example

Some architectures require that pointers are correctly aligned when accessing objects larger than a byte. However, it is common in system code that unaligned data (for example, the network stacks) must be copied to a properly aligned memory location, such as in this noncompliant code example:

```
#include <string.h>

struct foo_header {
    int len;
    /* ... */
};

void func(char *data, size_t offset) {
    struct foo_header *tmp;
    struct foo_header header;

    tmp = (struct foo_header *) (data + offset);
    memcpy(&header, tmp, sizeof(header));

    /* ... */
}
```

Assigning an unaligned value to a pointer that references a type that needs to be aligned is [undefined behavior](#). An [implementation](#) may notice, for example, that `tmp` and `header` must be aligned and use an inline `memcpy()` that uses instructions that assume aligned data.

Compliant Solution

This compliant solution avoids the use of the `foo_header` pointer:

```
#include <string.h>

struct foo_header {
    int len;
    /* ... */
};

void func(char *data, size_t offset) {
    struct foo_header header;
    memcpy(&header, data + offset, sizeof(header));

    /* ... */
}
```

Exceptions

EXP36-C-EX1: Some hardware architectures have relaxed requirements with regard to pointer alignment. Using a pointer that is not properly aligned is correctly handled by the architecture, although there might be a performance penalty. On such an architecture, improper pointer alignment is permitted but remains an efficiency problem.

EXP36-C-EX2: If a pointer is known to be correctly aligned to the target type, then a cast to that type is permitted. There are several cases where a pointer is known to be correctly aligned to the target type. The pointer could point to an object declared with a suitable alignment specifier. It could point to an object returned by `aligned_alloc()`, `calloc()`, `malloc()`, or `realloc()`, as per the C standard, section 7.22.3, paragraph 1 [[ISO/IEC 9899:2011](#)].

This compliant solution uses the alignment specifier, which is new to C11, to declare the `char` object `c` with the same alignment as that of an object of type `int`. As a result, the two pointers reference equally aligned pointer types:

```
#include <stdalign.h>
#include <assert.h>

void func(void) {
    /* Align c to the alignment of an int */
    alignas(int) char c = 'x';
    int *ip = (int *)&c;
    char *cp = (char *)ip;
    /* Both cp and &c point to equally aligned objects */
    assert(cp == &c);
}
```

Risk Assessment

Accessing a pointer or an object that is not properly aligned can cause a program to crash or give erroneous information, or it can cause slow pointer accesses (if the architecture allows misaligned accesses).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP36-C	Low	Probable	Medium	P4	L3

Related Guidelines

SEI CERT C++ Coding Standard	VOID EXP56-CPP. Do not cast pointers into more strictly aligned pointer types
ISO/IEC TR 24772:2013	Pointer Casting and Pointer Type Changes [HFC]
ISO/IEC TS 17961	Converting pointer values to more strictly aligned pointer types [alignconv]
MISRA C:2012	Rule 11.1 (required) Rule 11.2 (required) Rule 11.5 (advisory) Rule 11.7 (required)

Bibliography

[Bryant 2003]	
[ISO/IEC 9899:2011]	6.3.2.3, "Pointers"
[Walfridsson 2003]	Aliasing, Pointer Casts and GCC 3.3

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/tgAV>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
allow_cast_from_char_ptr	Allow all casts from char*.	False
allow_cast_from_void_ptr	Allow all casts from void*.	False
functions_returning_aligned_pointer	No violation will be reported when casting the return value of these functions.	('malloc', 'calloc', 'realloc', 'aligned_alloc')
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
cast_increases_alignment	Pointer cast increases alignment from {} to {} bytes

CertC-EXP37

Call functions with the correct number and type of arguments.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Do not call a function with the wrong number or type of arguments.

The C Standard identifies five distinct situations in which [undefined behavior](#) (UB) may arise as a result of invoking a function using a declaration that is incompatible with its definition or by supplying incorrect types or numbers of arguments:

UB	Description
26	<i>A pointer is used to call a function whose type is not compatible with the referenced type</i> (6.3.2.3).
38	<i>For a call to a function without a function prototype in scope, the number of arguments does not equal the number of parameters</i> (6.5.2.2).
39	<i>For a call to a function without a function prototype in scope where the function is defined with a function prototype, either the prototype ends with an ellipsis or the types of the arguments after promotion are not compatible with the types of the parameters</i> (6.5.2.2).
40	<i>For a call to a function without a function prototype in scope where the function is not defined with a function prototype, the types of the arguments after promotion are not compatible with those of the parameters after promotion</i> (with certain exceptions) (6.5.2.2).
41	<i>A function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function</i> (6.5.2.2).

Functions that are appropriately declared (as in [DCL40-C. Do not create incompatible declarations of the same function or object](#)) will typically generate a compiler diagnostic message if they are supplied with the wrong number or types of arguments. However, there are cases in which supplying the incorrect arguments to a function will, at best, generate compiler [warnings](#). Although such warnings should be resolved, they do not prevent program compilation. (See [MSC00-C. Compile cleanly at high warning levels](#).)

Noncompliant Code Example

The header `<tgmath.h>` provides type-generic macros for math functions. Although most functions from the `<math.h>` header have a complex counterpart in `<complex.h>`, several functions do not. Calling any of the following type-generic functions with complex values is [undefined behavior](#).

Functions That Should Not Be Called with Complex Values

atan2()	erf()	fdim()	fmin()	ilogb()	llround()	logb()	nextafter()	rint()	tgamma()
cbrt()	erfc()	floor()	fmod()	ldexp()	log10()	lrint()	nexttoward()	round()	trunc()
ceil()	exp2()	fma()	frexp()	lgamma()	log1p()	lround()	remainder()	scalbn()	
copysign()	expm1()	fmax()	hypot()	llrint()	log2()	nearbyint()	remquo()	scalbln()	

This noncompliant code example attempts to take the base-2 logarithm of a complex number, resulting in undefined behavior:

```
#include <tgmath.h>

void func(void) {
    double complex c = 2.0 + 4.0 * I;
    double complex result = log2(c);
}
```

Compliant Solution [Complex Number]

If the `clog2()` function is not available for an implementation as an extension, the programmer can take the base-2 logarithm of a complex number, using `log()` instead of `log2()`, because `log()` can be used on complex arguments, as shown in this compliant solution:

```
#include <tgmath.h>

void func(void) {
    double complex c = 2.0 + 4.0 * I;
    double complex result = log(c)/log(2);
}
```

Compliant Solution [Real Number]

The programmer can use this compliant solution if the intent is to take the base-2 logarithm of the real part of the complex number:

```
#include <tgmath.h>

void func(void) {
    double complex c = 2.0 + 4.0 * I;
    double complex result = log2(creal(c));
}
```

Noncompliant Code Example

In this noncompliant example, the C standard library function `strchr()` is called through the function pointer `fp` declared with incorrectly typed arguments. According to the C Standard, 6.3.2.3, paragraph 8 [[ISO/IEC 9899:2011](#)]

A pointer to a function of one type may be converted to a pointer to a function of another type and back again; the result shall compare equal to the original pointer. If a converted pointer is used to call a function whose type is not compatible with the referenced type, the behavior is undefined.

See [undefined behavior 26](#).

```
#include <stdio.h>
#include <string.h>

char *(*fp)();

int main(void) {
    const char *c;
    fp = strchr;
    c = fp('e', "Hello");
    printf("%s\n", c);
    return 0;
}
```

Compliant Solution

In this compliant solution, the function pointer `fp`, which points to the C standard library function `strchr()`, is declared with the correct parameters and is invoked with the correct number and type of arguments:

```
#include <stdio.h>
#include <string.h>

char *(*fp)(const char *, int);

int main(void) {
    const char *c;
    fp = strchr;
    c = fp("Hello",'e');
    printf("%s\n", c);
    return 0;
}
```

Noncompliant Code Example

In this noncompliant example, the function `f()` is defined to take an argument of type `long` but `f()` is called from another file with an argument of type `int`:

```
/* In another source file */
long f(long x) {
    return x < 0 ? -x : x;
}

/* In this source file, no f prototype in scope */
long f();

long g(int x) {
    return f(x);
}
```

Compliant Solution

In this compliant solution, the prototype for the function `f()` is included in the source file in the scope of where it is called, and the function `f()` is correctly called with an argument of type `long`:

```
/* In another source file */

long f(long x) {
    return x < 0 ? -x : x;
}

/* f prototype in scope in this source file */

long f(long x);

long g(int x) {
    return f((long)x);
}
```

Noncompliant Code Example (POSIX)

The POSIX function `open()` [IEEE Std 1003.1:2013] is a variadic function with the following prototype:

```
int open(const char *path, int oflag, ...);
```

The `open()` function accepts a third argument to determine a newly created file's access mode. If `open()` is used to create a new file and the third argument is omitted, the file may be created with unintended access permissions. (See [F1006-C. Create files with appropriate access permissions](#).)

In this noncompliant code example from a [vulnerability](#) in the `useradd()` function of the `shadow-utils` package [CVE-2006-1174](#), the third argument to `open()` is accidentally omitted:

```
fd = open(ms, O_CREAT | O_EXCL | O_WRONLY | O_TRUNC);
```

Technically, it is incorrect to pass a third argument to `open()` when not creating a new file (that is, with the `O_CREAT` flag not set).

Compliant Solution (POSIX)

In this compliant solution, a third argument is specified in the call to `open()`:

```
#include <fcntl.h>

void func(const char *ms, mode_t perms) {
    /* ... */
    int fd;
    fd = open(ms, O_CREAT | O_EXCL | O_WRONLY | O_TRUNC, perms);
    if (fd == -1) {
        /* Handle error */
    }
}
```

Risk Assessment

Calling a function with incorrect arguments can result in [unexpected](#) or unintended program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP37-C	Medium	Probable	High	P4	L3

Related Guidelines

CERT C Secure Coding Standard	DCL07-C. Include the appropriate type information in function declarators MSC00-C. Compile cleanly at high warning levels FI006-C. Create files with appropriate access permissions
ISO/IEC TR 24772:2013	Subprogram Signature Mismatch [OTR]
ISO/IEC TS 17961	Calling functions with incorrect arguments [argcomp]
MISRA C:2012	Rule 8.2 (required) Rule 17.3 (mandatory)
MITRE CWE	CWE-628 , Function Call with Incorrectly Specified Arguments CWE-686 , Function Call with Incorrect Argument Type

Bibliography

[CVE]	CVE-2006-1174
[ISO/IEC 9899:2011]	6.3.2.3, "Pointers" 6.5.2.2, "Function Calls"
[IEEE Std 1003.1:2013]	open()
[Spinellis 2006]	Section 2.6.1, "Incorrect Routine or Arguments"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/VQBC>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	4
real_only_macros		set(['atan2', 'nexttoward', 'nearbyint', 'rint', 'ldexp', 'hypot', 'logb', 'log2', 'floor', 'remainder', 'lround', 'ilogb', 'frexp', 'cbrt', 'log10', 'nextafter', 'remquo', 'scalbln', 'fmin', 'fmax', 'tgamma', 'scalbn', 'copysign', 'ceil', 'llrint', 'lrint', 'trunc', 'expm1', 'fdim', 'exp2', 'lgamma', 'erf', 'erfc', 'fmod', 'llround', 'fma', 'log1p', 'round'])
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high
use_pointer_analysis	Whether to use pointer analysis to verify indirect calls. Note: pointer analysis can only be used if CONFIG.Run_IRAnalysis_Checks is enabled.	True

Possible Messages

Name	Message
ellipsis_called_without_prototype	Cannot call function with ellipsis without a prototype declaration
open_mode_missing	{()} calls with {} must specify the 'mode' argument.
open_mode_redundant	The 'mode' argument is redundant in {()} calls without O_CREAT.
open_too_many_args	Too many arguments passed to {()}.
real_only_called_with_complex_arg	Calling a real-only function with a complex argument results in undefined behavior.
wrong_argument_number	Number of arguments at function call does not match number of parameters
wrong_argument_type	Parameter {} expects type '{}', but '{}' was given.

CertC-EXP40

Do not modify constant objects.

Input: IR

Source languages: C, C++

Details

The C Standard, 6.7.3, paragraph 6 [[ISO/IEC 9899:2011](#)], states

If an attempt is made to modify an object defined with a `const`-qualified type through use of an `lvalue` with non-`const`-qualified type, the behavior is undefined.

See also [undefined behavior 64](#).

There are existing compiler [implementations](#) that allow `const`-qualified objects to be modified without generating a warning message.

Avoid casting away `const` qualification because doing so makes it possible to modify `const`-qualified objects without issuing diagnostics. (See [EXP05-C. Do not cast away a const qualification](#) and [STR30-C. Do not attempt to modify string literals](#) for more details.)

Noncompliant Code Example

This noncompliant code example allows a constant object to be modified:

```
const int **ipp;
int *ip;
const int i = 42;

void func(void) {
    ipp = &ip; /* Constraint violation */
    *ipp = &i; /* Valid */
    *ip = 0;   /* Modifies constant i (was 42) */
}
```

The first assignment is unsafe because it allows the code that follows it to attempt to change the value of the `const` object `i`.

Implementation Details

If `ipp`, `ip`, and `i` are declared as automatic variables, this example compiles without warning with Microsoft Visual Studio 2013 when compiled in C mode (/TC) and the resulting program changes the value of `i`. GCC 4.8.1 generates a warning but compiles, and the resulting program changes the value of `i`.

If `ipp`, `ip`, and `i` are declared with static storage duration, this program compiles without warning and terminates abnormally with Microsoft Visual Studio 2013, and compiles with warning and terminates abnormally with GCC 4.8.1.

Compliant Solution

The compliant solution depends on the intent of the programmer. If the intent is that the value of `i` is modifiable, then it should not be declared as a constant, as in this compliant solution:

```
int **ipp;
int *ip;
int i = 42;

void func(void) {
    ipp = &ip; /* Valid */
    *ipp = &i; /* Valid */
    *ip = 0;   /* Valid */
}
```

If the intent is that the value of `i` is not meant to change, then do not write noncompliant code that attempts to modify it.

Risk Assessment

Modifying constant objects through nonconstant references is [undefined behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP40-C	Low	Unlikely	Medium	P2	L3

Related Guidelines

CERT C Secure Coding Standard	EXP05-C. Do not cast away a const qualification STR30-C. Do not attempt to modify string literals
---	--

Bibliography

[ISO/IEC 9899:2011]	Subclause 6.7.3, "Type Qualifiers"
-------------------------------------	------------------------------------

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/gAU>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
check_const	Whether casts affecting const-qualifiers should be reported	True
check_volatile	Whether casts affecting volatile-qualifiers should be reported	False
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
only_top_level		False
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
cast_adds_const	Cast adds const qualification
cast_adds_VOLATILE	Cast adds volatile qualification
implicit_cast_adds_const	Implicit cast adds const qualification
implicit_cast_adds_VOLATILE	Implicit cast adds volatile qualification

CertC-EXP42

Do not compare padding data.

Input: IR

Source languages: C, C++

Details

The C Standard, 6.7.2.1 [[ISO/IEC 9899:2011](#)], states

There may be unnamed padding within a structure object, but not at its beginning. . . . There may be unnamed padding at the end of a structure or union.

Subclause 6.7.9, paragraph 9, states that

unnamed members of objects of structure and union type do not participate in initialization. Unnamed members of structure objects have indeterminate value even after initialization.

The only exception is that padding bits are set to zero when a static or thread-local object is implicitly initialized (paragraph 10):

If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate. If an object that has static or thread storage duration is not initialized explicitly, then:

- if it is an aggregate, every member is initialized (recursively) according to these rules, and any padding is initialized to zero bits;
- if it is a union, the first named member is initialized (recursively) according to these rules, and any padding is initialized to zero bits;

Because these padding values are unspecified, attempting a byte-by-byte comparison between structures can lead to incorrect results [Summit 1995].

Noncompliant Code Example

In this noncompliant code example, `memcmp()` is used to compare the contents of two structures, including any padding bytes:

```
#include <string.h>

struct s {
    char c;
    int i;
    char buffer[13];
};

void compare(const struct s *left, const struct s *right) {
    if (0 == memcmp(left, right, sizeof(struct s))) {
        /* ... */
    }
}
```

Compliant Solution

In this compliant solution, all of the fields are compared manually to avoid comparing any padding bytes:

```
#include <string.h>

struct s {
    char c;
    int i;
    char buffer[13];
};

void compare(const struct s *left, const struct s *right) {
    if ((left && right) &&
        (left->c == right->c) &&
        (left->i == right->i) &&
        (0 == memcmp(left->buffer, right->buffer, 13))) {
        /* ... */
    }
}
```

Exceptions

EXP42-C-EX1: A structure can be defined such that the members are aligned properly or the structure is packed using implementation-specific packing instructions. This is true only when the members' data types have no padding bits of their own and when their object representations are the same as their value representations. This frequently is not true for the `_Bool` type or floating-point types and need not be true for pointers. In such cases, the compiler does not insert padding, and use of functions such as `memcmp()` is acceptable.

This compliant example uses the `#pragma pack` compiler extension from Microsoft Visual Studio to ensure the structure members are packed as tightly as possible:

```
#include <string.h>

#pragma pack(push, 1)
struct s {
    char c;
    int i;
    char buffer[13];
};
#pragma pack(pop)

void compare(const struct s *left, const struct s *right) {
    if (0 == memcmp(left, right, sizeof(struct s))) {
        /* ... */
    }
}
```

Risk Assessment

Comparing padding bytes, when present, can lead to [unexpected program behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP42-C	Medium	Probable	Medium	P8	L2

Related Guidelines

ISO/IEC TS 17961	Comparison of padding data [padcomp]
SEI CERT C++ Coding Standard	EXP62-CPP. Do not access the bits of an object representation that are not part of the object's value representation

Bibliography

[ISO/IEC 9899:2011]	6.7.2.1, "Structure and Union Specifiers" 6.7.9, "Initialization"
[Summit 1995]	Question 2.8 Question 2.12

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/CoDYBg>], Copyright [C] 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
allow_char	Whether to allow memcmp on 'char' type.	True
allow_composites_without_padding	Whether to allow using memcmp on structs and unions that have no padding bytes.	True
allow_float	Whether to allow memcmp on floating point types.	True
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
disallowed_memcmp_pointer_arg	Disallowed type of pointer argument.
memcmp_char_pointer_arg	memcmp shall not be used with char pointer argument, use strncmp instead.
memcmp_float	memcmp shall not be used to compare floats as the same value may be stored using different representations.
memcmp_padding	memcmp shall not be used to compare structs with padding.
memcmp_struct_pointer_arg	memcmp shall not be used with struct pointer argument as it would compare padding as well.
memcmp_union_pointer_arg	memcmp shall not be used with union pointer argument as it would compare padding and different kinds of representation.

CertC-EXP44

Do not rely on side effects in operands to sizeof, __Alignof, or __Generic.

Input: IR

Source languages: C, C++

Details

Some operators do not evaluate their operands beyond the type information the operands provide. When using one of these operators, do not pass an operand that would otherwise yield a side effect since the side effect will not be generated.

The `sizeof` operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. In most cases, the operand is not evaluated. A possible exception is when the type of the operand is a variable length array type (VLA); then the expression is evaluated. When part of the operand of the `sizeof` operator is a VLA type and when changing the value of the VLA's size expression would not affect the result of the operator, it is [unspecified](#) whether or not the size expression is evaluated. (See [unspecified behavior 22](#).)

The operand passed to `__Alignof` is never evaluated, despite not being an expression. For instance, if the operand is a VLA type and the VLA's size expression contains a side effect, that side effect is never evaluated.

The operand used in the controlling expression of a `__Generic` selection expression is never evaluated.

Providing an expression that appears to produce side effects may be misleading to programmers who are not aware that these expressions are not evaluated, and in the case of a VLA used in `sizeof`, have unspecified results. As a result, programmers may make invalid assumptions about program state, leading to errors and possible software [vulnerabilities](#).

This rule is similar to [PRE31-C. Avoid side effects in arguments to unsafe macros](#).

Noncompliant Code Example (`sizeof`)

In this noncompliant code example, the expression `a++` is not evaluated:

```
#include <stdio.h>

void func(void) {
    int a = 14;
    int b = sizeof(a++);
    printf("%d, %d\n", a, b);
}
```

Consequently, the value of `a` after `b` has been initialized is 14.

Compliant Solution (`sizeof`)

In this compliant solution, the variable `a` is incremented outside of the `sizeof` operation:

```
#include <stdio.h>

void func(void) {
    int a = 14;
    int b = sizeof(a);
    ++a;
    printf("%d, %d\n", a, b);
}
```

Noncompliant Code Example (`sizeof`, VLA)

In this noncompliant code example, the expression `++n` in the initialization expression of `a` must be evaluated because its value affects the size of the VLA operand of the `sizeof` operator. However, in the initialization expression of `b`, the expression `++n % 1` evaluates to 0. This means that the value of `n` does not affect the result of the `sizeof` operator. Consequently, it is unspecified whether or not `n` will be incremented when initializing `b`.

```
#include <stddef.h>
#include <stdio.h>

void f(size_t n) {
    /* n must be incremented */
    size_t a = sizeof(int[++n]);

    /* n need not be incremented */
    size_t b = sizeof(int[++n % 1 + 1]);

    printf("%zu, %zu, %zu\n", a, b, n);
    /* ... */
}
```

Compliant Solution (`sizeof`, VLA)

This compliant solution avoids changing the value of the variable `n` used in each `sizeof` expression and instead increments `n` safely afterwards:

```
#include <stddef.h>
#include <stdio.h>

void f(size_t n) {
    size_t a = sizeof(int[n + 1]);
    ++n;

    size_t b = sizeof(int[n % 1 + 1]);
    ++n;
    printf("%zu, %zu, %zu\n", a, b, n);
    /* ... */
}
```

Noncompliant Code Example (`_Generic`)

This noncompliant code example attempts to modify a variable's value as part of the `_Generic` selection control expression. The programmer may expect that `a` is incremented, but because `_Generic` does not evaluate its control expression, the value of `a` is not modified.

```
#include <stdio.h>

#define S(val) _Generic(val, int : 2, \
                      short : 3, \
                      default : 1)

void func(void) {
    int a = 0;
    int b = S(a++);
    printf("%d, %d\n", a, b);
}
```

Compliant Solution (`_Generic`)

In this compliant solution, `a` is incremented outside of the `_Generic` selection expression:

```
#include <stdio.h>

#define S(val) _Generic(val, int : 2, \
                      short : 3, \
                      default : 1)

void func(void) {
    int a = 0;
    int b = S(a);
    ++a;
    printf("%d, %d\n", a, b);
}
```

Noncompliant Code Example (`_Alignof`)

This noncompliant code example attempts to modify a variable while getting its default alignment value. The user may have expected `val` to be incremented as part of the `_Alignof` expression, but because `_Alignof` does not evaluate its operand, `val` is unchanged.

```
#include <stdio.h>

void func(void) {
    int val = 0;
    /* ... */
    size_t align = _Alignof(int[++val]);
    printf("%zu, %d\n", align, val);
    /* ... */
}
```

Compliant Solution (`_Alignof`)

This compliant solution moves the expression out of the `_Alignof` operator:

```
#include <stdio.h>
void func(void) {
    int val = 0;
    /* ... */
    ++val;
    size_t align = _Alignof(int[val]);
    printf("%zu, %d\n", align, val);
    /* ... */
}
```

Exceptions

EXP44-C-EX1: Reading a `volatile`-qualified value is a side-effecting operation. However, accessing a value through a `volatile`-qualified type does not guarantee side effects will happen on the read of the value unless the underlying object is also `volatile`-qualified. Idiomatic reads of a `volatile`-qualified object are permissible as an operand to a `sizeof()`, `_Alignof()`, or `_Generic` expression, as in the following example:

```
void f(void) {
    int * volatile v;
    (void)sizeof(*v);
}
```

Risk Assessment

If expressions that appear to produce side effects are supplied to an operator that does not evaluate its operands, the results may be different than expected. Depending on how this result is used, it can lead to unintended program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP44-C	Low	Unlikely	Low	P3	L3

Related Guidelines

SEI CERT C++ Coding Standard	EXP52-CPP. Do not rely on side effects in unevaluated operands
--	--

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/LQo>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
side_effect_in_alignof	Operand of "_Alignof" shall not contain side effects
side_effect_in_generic	Selector of "_Generic" shall not contain side effects
side_effect_in_sizeof	Operand of "sizeof" shall not contain side effects

CertC-EXP45

Do not perform assignments in selection statements.

Input: IR

Source languages: C, C++

Details

Do not use the assignment operator in the contexts listed in the following table because doing so typically indicates programmer error and can result in [unexpected behavior](#).

Operator	Context
if	Controlling expression
while	Controlling expression
do ... while	Controlling expression
for	Second operand
? :	First operand
? :	Second or third operands, where the ternary expression is used in any of these contexts
&&	Either operand
	either operand
,	Second operand, when the comma expression is used in any of these contexts

Noncompliant Code Example

In this noncompliant code example, an assignment expression is the outermost expression in an if statement:

```
if (a = b) {
    /* ... */
}
```

Although the intent of the code may be to assign b to a and test the value of the result for equality to 0, it is frequently a case of the programmer mistakenly using the assignment operator = instead of the equals operator ==. Consequently, many compilers will warn about this condition, making this coding error detectable by adhering to [MSC00-C. Compile cleanly at high warning levels](#).

Compliant Solution (Unintentional Assignment)

When the assignment of b to a is not intended, the conditional block is now executed when a is equal to b:

```
if (a == b) {
    /* ... */
}
```

Compliant Solution (Intentional Assignment)

When the assignment is intended, this compliant solution explicitly uses inequality as the outermost expression while performing the assignment in the inner expression:

```
if ((a = b) != 0) {
    /* ... */
}
```

It is less desirable in general, depending on what was intended, because it mixes the assignment in the condition, but it is clear that the programmer intended the assignment to occur.

Noncompliant Code Example

In this noncompliant code example, the expression x = y is used as the controlling expression of the while statement:

```
do { /* ... */ } while (foo(), x = y);
```

The same result can be obtained using the for statement, which is specifically designed to evaluate an expression on each iteration of the loop, just before

performing the test in its controlling expression:

```
for (; x; foo(), x = y) { /* ... */ }
```

Compliant Solution [Unintentional Assignment]

When the assignment of `y` to `x` is not intended, the conditional block should be executed only when `x` is equal to `y`, as in this compliant solution:

```
do { /* ... */ } while (foo(), x == y);
```

Compliant Solution [Intentional Assignment]

When the assignment is intended, this compliant solution can be used:

```
do { /* ... */ } while (foo(), (x = y) != 0);
```

Noncompliant Code Example

In this noncompliant example, the expression `p = q` is used as the controlling expression of the `while` statement:

```
do { /* ... */ } while (x = y, p = q);
```

Compliant Solution

In this compliant solution, the expression `x = y` is not used as the controlling expression of the `while` statement:

```
do { /* ... */ } while (x = y, p == q);
```

Noncompliant Code Example

This noncompliant code example has a typo that results in an assignment rather than a comparison.

```
while (ch = '\t' && ch == ' ' && ch == '\n') {  
/* ... */  
}
```

Many compilers will warn about this condition. This coding error would typically be eliminated by adherence to [MSC00-C. Compile cleanly at high warning levels](#). Although this code compiles, it will cause [unexpected behavior](#) to an unsuspecting programmer. If the intent was to verify a string such as a password, user name, or group user ID, the code may produce significant [vulnerabilities](#) and require significant debugging.

Compliant Solution [RHS Variable]

When comparisons are made between a variable and a literal or const-qualified variable, placing the variable on the right of the comparison operation can prevent a spurious assignment.

In this code example, the literals are placed on the left-hand side of each comparison. If the programmer were to inadvertently use an assignment operator, the statement would assign `ch` to `'\t'`, which is invalid and produces a diagnostic message.

```
while ('\t' == ch && ' ' == ch && '\n' == ch) {  
/* ... */  
}
```

Due to the diagnostic, the typo will be easily spotted and fixed.

```
while ('\t' == ch && ' ' == ch && '\n' == ch) {  
/* ... */  
}
```

As a result, any mistaken use of the assignment operator that could otherwise create a [vulnerability](#) for operations such as string verification will result in a compiler diagnostic regardless of compiler, warning level, or [implementation](#).

Exceptions

EXP45-C-EX1: Assignment can be used where the result of the assignment is itself an operand to a comparison expression or relational expression. In this compliant example, the expression `x = y` is itself an operand to a comparison operation:

```
if ((x = y) != 0) { /* ... */ }
```

EXP45-C-EX2: Assignment can be used where the expression consists of a single primary expression. The following code is compliant because the expression `x = y` is a single primary expression:

```
if ((x = y)) { /* ... */ }
```

The following controlling expression is noncompliant because `&&` is not a comparison or relational operator and the entire expression is not primary:

```
if ((v = w) && flag) { /* ... */ }
```

When the assignment of `v` to `w` is not intended, the following controlling expression can be used to execute the conditional block when `v` is equal to `w`:

```
if ((v == w) && flag) { /* ... */ };
```

When the assignment is intended, the following controlling expression can be used:

```
if (((v = w) != 0) && flag) { /* ... */ };
```

EXP45-C-EX3: Assignment can be used in a function argument or array index. In this compliant solution, the expression `x = y` is used in a function argument:

```
if (foo(x = y)) { /* ... */ }
```

Risk Assessment

Errors of omission can result in unintended program flow.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
EXP45-C	Low	Likely	Medium	P6	L2

Related Guidelines

SEI CERT C++ Coding Standard	EXP19-CPP. Do not perform assignments in conditional expressions
CERT Oracle Secure Coding Standard for Java	EXP51-J. Do not perform assignments in conditional expressions
ISO/IEC TR 24772:2013	Likely Incorrect Expression [KOA]
ISO/IEC TS 17961	No assignment in conditional expressions [boolasgn]
MITRE CWE	CWE-480 , Use of Incorrect Operator

Bibliography

[Dutta 03]	"Best Practices for Programming in C"
[Hatton 1995]	Section 2.7.2, "Errors of Omission and Addition"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/nYFtAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
allow_in_extra_parens	Whether if <code>{(x = y)}</code> should be allowed (note the extra parens)	True
allow_in_relational_operator	Whether an assignment is allowed as operand of a comparison or relational operator, e.g. <code>(x = y) != 0</code>	True
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
assignment_inside_bool	Assignment inside boolean expression.

CertC-EXP46

Do not use a bitwise operator with a Boolean-like operand.

Input: IR

Source languages: C, C++

Details

Mixing bitwise and relational operators in the same full expression can be a sign of a logic error in the expression where a logical operator is usually the

intended operator. Do not use the bitwise AND (`&`), bitwise OR (`||`), or bitwise XOR (`^`) operators with an operand of type `_Bool`, or the result of a *relational-expression* or *equality-expression*. If the bitwise operator is intended, it should be indicated with use of a parenthesized expression.

Noncompliant Code Example

In this noncompliant code example, a bitwise `&` operator is used with the results of an *equality-expression*:

```
if (!(getuid() & geteuid() == 0)) {  
/* ... */  
}
```

Compliant Solution

This compliant solution uses the `&&` operator for the logical operation within the conditional expression:

```
if (!(getuid() && geteuid() == 0)) {  
/* ... */  
}
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP46-C	Low	Likely	Low	P9	L2

Related Guidelines

ISO/IEC TR 24772:2013	Likely Incorrect Expression [KOA]
MITRE CWE	CWE-480 , Use of incorrect operator

Bibliography

[Hatton 1995]	Section 2.7.2, "Errors of Omission and Addition"
-------------------------------	--

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/g4FtAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
allow_sizeof	Allows the use of <code>sizeof(bool_var)</code> . The Misra rule forbids all uses of expressions of type <code>bool</code> except for those explicitly allowed; and does not allow <code>sizeof</code> . However this is likely an oversight; there is no good reason to prevent the use of <code>sizeof(bool_var)</code> . This option also allows <code>alignof(bool_var)</code> in C++11.	True
allow_unevaluated_operator	Allows the use of boolean expressions in contexts where the expression is not evaluated. As of C++17, these are: <code>typeid(bool_var)</code> , <code>noexcept(bool_var)</code> and <code>decltype(bool_var)</code> . <code>sizeof(bool_var)</code> is controlled by the separate option 'allow_sizeof'.	True
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	9
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
<code>bool_operand_in_bad_operator</code>	Use of boolean operand in '{}' operator

CertC-EXP47

Do not call `va_arg` with an argument of the incorrect type.

Input: IR

Source languages: C, C++

Details

The variable arguments passed to a variadic function are accessed by calling the `va_arg()` macro. This macro accepts the `va_list` representing the variable arguments of the function invocation and the type denoting the expected argument type for the argument being retrieved. The macro is typically invoked within a loop, being called once for each expected argument. However, there are no type safety guarantees that the type passed to `va_arg` matches the type passed by the caller, and there are generally no compile-time checks that prevent the macro from being invoked with no argument available to the function call. The C Standard, 7.16.1.1, states [ISO/IEC 9899:2011], in part:

If there is no actual next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined, except for the following cases:

- one type is a signed integer type, the other type is the corresponding unsigned integer type, and the value is representable in both types;
- one type is pointer to `void` and the other is a pointer to a character type.

Ensure that an invocation of the `va_arg()` macro does not attempt to access an argument that was not passed to the variadic function. Further, the type passed to the `va_arg()` macro must match the type passed to the variadic function after default argument promotions have been applied. Either circumstance results in undefined behavior.

Noncompliant Code Example

This noncompliant code example attempts to read a variadic argument of type `unsigned char` with `va_arg()`. However, when a value of type `unsigned char` is passed to a variadic function, the value undergoes default argument promotions, resulting in a value of type `int` being passed.

```
#include <stdarg.h>
#include <stddef.h>

void func(size_t count, ...) {
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        unsigned char c = va_arg(ap, unsigned char);
        // ...
    }
    va_end(ap);
}

void f(void) {
    unsigned char c = 0x12;
    func(1, c);
}
```

Compliant Solution

The compliant solution accesses the variadic argument with type `int`, and then casts the resulting value to type `unsigned char`:

```
#include <stdarg.h>
#include <stddef.h>

void func(size_t count, ...) {
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        unsigned char c = (unsigned char)va_arg(ap, int);
        // ...
    }
    va_end(ap);
}

void f(void) {
    unsigned char c = 0x12;
    func(1, c);
}
```

Noncompliant Code Example

This noncompliant code example assumes that at least one variadic argument is passed to the function, and attempts to read it using the `va_arg()` macro. This pattern arises frequently when a variadic function uses a sentinel value to denote the end of the variable argument list. However, the caller does not pass any extra arguments to the function, resulting in undefined behavior.

```
#include <stdarg.h>

void func(const char *cp, ...) {
    va_list ap;
    va_start(ap, cp);
    int val = va_arg(ap, int);
    // ...
    va_end(ap);
}

void f(void) {
    func("The only argument");
}
```

Compliant Solution

It is not possible for the variadic function to determine how many arguments are actually provided to the function call; that information must be passed in an

out-of-band way. Oftentimes this results in the information being encoded in the initial parameter, as in this compliant solution:

```
#include <stdarg.h>
#include <stddef.h>

void func(const char *cp, size_t numArgs, ...) {
    va_list ap;
    va_start(ap, cp);
    if (numArgs > 0) {
        int val = va_arg(ap, int);
        // ...
    }
    va_end(ap);
}

void f(void) {
    func("The only argument", 0);
}
```

Risk Assessment

Incorrect use of `va_arg()` results in undefined behavior that can include accessing stack memory.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP47-C	Medium	Likely	High	P6	L2

Bibliography

[ISO/IEC 9899:2011]	Subclause 7.16, "Variable Arguments <stdarg.h>" Subclause 6.5.2.2, "Function calls"
---------------------	--

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/BYACQw>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
va_arg_missing	Call requires at least one argument for variadic parameter.
va_arg_with_unpromoted_type	Type {} does not match promoted type {}.

CertC-INT00

Understand the data model used by your implementation(s).

Input: IR

Source languages: C, C++

Details

A *data model* defines the sizes assigned to standard data types. It is important to understand the data models used by your [implementation](#). However, if your code depends on any assumptions not guaranteed by the standard, you should provide static assertions to ensure that your assumptions are valid. (See [DCL03-C. Use a static assertion to test the value of a constant expression](#).) Assumptions concerning integer sizes may become invalid, for example, when porting from a 32-bit architecture to a 64-bit architecture.

Common Data Models

Data Type	iAPX86	IA-32	IA-64	SPARC-64	ARM-32	Alpha	64-bit Linux, FreeBSD, NetBSD, and OpenBSD
char	8	8	8	8	8	8	8
short	16	16	16	16	16	16	16
int	16	32	32	32	32	32	32
long	32	32	32	64	32	64	64
long long	N/A	64	64	64	64	64	64
Pointer	16/32	32	64	64	32	64	64

Code frequently embeds assumptions about data models. For example, some code bases require pointer and `long` to have the same size, whereas other large code bases require `int` and `long` to be the same size [van de Voort 2007]. These types of assumptions, while common, make the code difficult to port and make the ports error prone. One solution is to avoid any [implementation-defined behavior](#). However, this practice can result in inefficient code. Another solution is to include either static or runtime assertions near any platform-specific assumptions, so they can be easily detected and corrected during porting.

<limits.h>

Possibly more important than knowing the number of bits for a given type is knowing that `limits.h` defines macros that can be used to determine the integral ranges of the standard integer types for any conforming implementation. For example, `UINT_MAX` is the largest possible value of an `unsigned int`, and `LONG_MIN` is the smallest possible value of a `long int`.

<stdint.h>

The `stdint.h` header introduces types with specific size restrictions that can be used to avoid dependence on a particular data model. For example, `int_least32_t` is the smallest signed integer type supported by the implementation that contains at least 32 bits. The type `uint_fast16_t` is the fastest unsigned integer type supported by the implementation that contains at least 16 bits. The type `intmax_t` is the largest signed integer, and `uintmax_t` is the largest unsigned type, supported by the implementation. The following types are required to be available on all implementations:

Smallest Types	Signed	Unsigned
8 bits	<code>int_least8_t</code>	<code>uint_least8_t</code>
16 bits	<code>int_least16_t</code>	<code>uint_least16_t</code>
32 bits	<code>int_least32_t</code>	<code>uint_least32_t</code>
64 bits	<code>int_least64_t</code>	<code>uint_least64_t</code>
Fastest Types	Signed	Unsigned
8 bits	<code>int_fast8_t</code>	<code>uint_fast8_t</code>
16 bits	<code>int_fast16_t</code>	<code>uint_fast16_t</code>
32 bits	<code>int_fast32_t</code>	<code>uint_fast32_t</code>
64 bits	<code>int_fast64_t</code>	<code>uint_fast64_t</code>
Largest Types	Signed	Unsigned
Maximum	<code>intmax_t</code>	<code>uintmax_t</code>

Additional types may be supported by an implementation, such as `int8_t`, a type of exactly 8 bits, and `uintptr_t`, a type large enough to hold a converted `void *` if such an integer exists in the implementation.

<inttypes.h>

The `inttypes.h` header declares functions for manipulating greatest-width integers and converting numeric character strings to greatest-width integers.

Noncompliant Code Example

This noncompliant example attempts to read a `long` into an `int`. This code works for models in which `sizeof(int) == sizeof(long)`. For others, it causes an unexpected memory write similar to a buffer overflow.

```
int f(void) {
    FILE *fp;
    int x;
/* ... */
    if (fscanf(fp, "%ld", &x) < 1) {
        return -1; /* Indicate failure */
    }
/* ... */
    return 0;
}
```

Some compilers can generate warnings if a constant format string does not match the argument types.

Compliant Solution

This compliant solution uses the correct format for the type being used:

```
int f(void) {
    FILE *fp;
    int x;
    /* Initialize fp */
    if (fscanf(fp, "%d", &x) < 1) {
        return -1; /* Indicate failure */
    }
    /* ... */
    return 0;
}
```

Noncompliant Code Example

This noncompliant code attempts to guarantee that all bits of a multiplication of two `unsigned int` values are retained by performing arithmetic in the type `unsigned long`. This practice works for some platforms, such as 64-bit Linux, but fails for others, such as 64-bit Microsoft Windows.

```
unsigned int a, b;
unsigned long c;
/* Initialize a and b */
c = (unsigned long)a * b; /* Not guaranteed to fit */
```

Compliant Solution

This compliant solution uses the largest unsigned integer type available if it is guaranteed to hold the result. If it is not, another solution must be found, as discussed in [INT32-C. Ensure that operations on signed integers do not result in overflow](#).

```
#if UINT_MAX > UINTMAX_MAX/UINT_MAX
#error No safe type is available.
#endif
/* ... */
unsigned int a, b;
uintmax_t c;
/* Initialize a and b */
c = (uintmax_t)a * b; /* Guaranteed to fit, verified above */
```

Risk Assessment

Understanding the data model used by your [implementation](#) is necessary to avoid making errors about the sizes of integer types and the range of values they can represent. Making assumptions about the sizes of data types may lead to buffer-overflow-style attacks.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
INT00-C	High	Unlikely	High	P3	L3

Related Guidelines

SEI CERT C++ Coding Standard	INT00-CPP. Understand the data model used by your implementation(s)
ISO/IEC TR 24772:2013	Bit Representations [STR]

Bibliography

[Open Group 1997a]
[van de Voort 2007]

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/FhE>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
cast_to_same_size	Cast between different but same-sized integral types, intended?

CertC-INT01

Use `rsize_t` or `size_t` for all integer values representing the size of an object.

Input: IR

Source languages: C, C++

Details

The `size_t` type is the unsigned integer type of the result of the `sizeof` operator. Variables of type `size_t` are guaranteed to be of sufficient precision to represent the size of an object. The limit of `size_t` is specified by the `SIZE_MAX` macro.

The type `size_t` generally covers the entire address space. The C Standard, Annex K (normative), "Bounds-checking interfaces," introduces a new type, `rsize_t`, defined to be `size_t` but explicitly used to hold the size of a single object [Meyers 2004]. In code that documents this purpose by using the type `rsize_t`, the size of an object can be checked to verify that it is no larger than `RSIZE_MAX`, the maximum size of a normal single object, which provides additional input validation for library functions. See [STR07-C. Use the bounds-checking interfaces for string manipulation](#) for additional discussion of C11 Annex K.

Any variable that is used to represent the size of an object, including integer values used as sizes, indices, loop counters, and lengths, should be declared `rsize_t`, if available. Otherwise, it should be declared `size_t`.

Noncompliant Code Example

In this noncompliant code example, the dynamically allocated buffer referenced by `p` overflows for values of `n > INT_MAX`:

```
char *copy(size_t n, const char *c_str) {
    int i;
    char *p;

    if (n == 0) {
        /* Handle unreasonable object size error */
    }
    p = (char *)malloc(n);
    if (p == NULL) {
        return NULL; /* Indicate malloc failure */
    }
    for (i = 0; i < n; ++i) {
        p[i] = *c_str++;
    }
    return p;
}

char c_str[] = "hi there";
char *p = copy(sizeof(c_str), c_str);
```

Signed integer overflow causes [undefined behavior](#). The following are two possible conditions under which this code constitutes a serious [vulnerability](#):

```
sizeof(size_t) == sizeof(int)
```

The unsigned `n` may contain a value greater than `INT_MAX`. Assuming quiet wraparound on signed overflow, the loop executes `n` times because the comparison `i < n` is an unsigned comparison. Once `i` is incremented beyond `INT_MAX`, `i` takes on negative values starting with `(INT_MIN)`. Consequently, the memory locations referenced by `p[i]` precede the memory referenced by `p`, and a write outside array bounds occurs.

```
sizeof(size_t) > sizeof(int)
```

For values of `n` where `0 < n <= INT_MAX`, the loop executes `n` times, as expected.

For values of `n` where `INT_MAX < n <= (size_t)INT_MIN`, the loop executes `INT_MAX` times. Once `i` becomes negative, the loop stops, and `i` remains in the range `0` through `INT_MAX`.

For values of `n` where `(size_t)INT_MIN < n <= SIZE_MAX`, `i` wraps and takes the values `INT_MIN` to `INT_MIN + (n - (size_t)INT_MIN - 1)`. Execution of the

loop overwrites memory from `p[INT_MIN]` through `p[INT_MIN + (n - (size_t)INT_MIN - 1)]`.

Compliant Solution [C11, Annex K]

Declaring `i` to be of type `rsize_t` eliminates the possible integer overflow condition [in this example]. Also, the argument `n` is changed to be of type `rsize_t` to document additional [validation](#) in the form of a check against `RSIZE_MAX`:

```
char *copy(rsize_t n, const char *c_str) {
    rsize_t i;
    char *p;

    if (n == 0 || n > RSIZE_MAX) {
        /* Handle unreasonable object size error */
    }
    p = (char *)malloc(n);
    if (p == NULL) {
        return NULL; /* Indicate malloc failure */
    }
    for (i = 0; i < n; ++i) {
        p[i] = *c_str++;
    }
    return p;
}

/* ... */

char c_str[] = "hi there";
char *p = copy(sizeof(c_str), c_str);
```

Noncompliant Code Example

In this noncompliant code example, the value of `length` is read from a network connection and passed as an argument to a wrapper to `malloc()` to allocate the appropriate data block. Provided that the size of an `unsigned long` is equal to the size of an `unsigned int`, and both sizes are equal to or smaller than the size of `size_t`, this code runs as expected. However, if the size of an `unsigned long` is greater than the size of an `unsigned int`, the value stored in `length` may be truncated when passed as an argument to `alloc()`.

```
void *alloc(unsigned int blocksize) {
    return malloc(blocksize);
}

int read_counted_string(int fd) {
    unsigned long length;
    unsigned char *data;

    if (read_integer_from_network(fd, &length) < 0) {
        return -1;
    }

    data = (unsigned char*)alloc(length);
    if (data == NULL) {
        return -1; /* Indicate failure */
    }

    if (read_network_data(fd, data, length) < 0) {
        free(data);
        return -1;
    }
    data[length-1] = '\0';

    /* ... */
    free( data );
    return 0;
}
```

Compliant Solution [C11, Annex K]

Declaring both `length` and the `blocksize` argument to `alloc()` as `rsize_t` eliminates the possibility of truncation. This compliant solution assumes that `read_integer_from_network()` and `read_network_data()` can also be modified to accept a `length` argument of type pointer to `rsize_t` and `rsize_t`, respectively. If these functions are part of an external library that cannot be updated, care must be taken when casting `length` into an `unsigned long` to ensure that integer truncation does not occur.

```
void *alloc(rsize_t blocksize) {
    if (blocksize == 0 || blocksize > RSIZE_MAX) {
        return NULL; /* Indicate failure */
    }
    return malloc(blocksize);
}

int read_counted_string(int fd) {
    rsize_t length;
    unsigned char *data;

    if (read_integer_from_network(fd, &length) < 0) {
        return -1;
    }

    data = (unsigned char*)alloc(length);
    if (data == NULL) {
        return -1; /* Indicate failure */
    }

    if (read_network_data(fd, data, length) < 0) {
        free(data);
    }
}
```

```

        return -1;
    }
    data[length-1] = '\0';

    /* ... */
    free( data );
    return 0;
}

```

Risk Assessment

The improper calculation or manipulation of an object's size can result in exploitable [vulnerabilities](#).

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
INT01-C	Medium	Probable	Medium	P8	L2

Related Guidelines

SEI CERT C++ Coding Standard	INT01-CPP. Use rsize_t or size_t for all integer values representing the size of an object
--	--

Bibliography

[Meyers 2004]	
-------------------------------	--

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/PwE>], Copyright [C] 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
object_allocators		dict(...)
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
use_rsize_t_or_size_t_for_both_ops	Use rsize_t or size_t for both arguments of the relational operator.
use_rsize_t_or_size_t_in_alloc	Use rsize_t or size_t for the argument of the call to a memory allocating function.

CertC-INT05

Do not use input functions to convert character data if they cannot handle all possible inputs.

Input: IR

Source languages: C, C++

Details

Do not use functions that input characters and convert them to integers if the functions cannot handle all possible inputs. For example, formatted input functions such as `scanf()`, `fscanf()`, `vscanf()`, and `vfscanf()` can be used to read string data from `stdin` or (in the cases of `fscanf()` and `vfscanf()`) other input streams. These functions work fine for valid integer values but lack robust error handling for invalid values.

Alternatively, input character data as a null-terminated byte string and convert to an integer value using `strtol()` or a related function. (See [ERR34-C. Detect errors when converting a string to a number](#).)

Noncompliant Code Example

This noncompliant code example uses the `scanf()` function to read a string from `stdin` and convert it to a `long`. The `scanf()` and `fscanf()` functions have [undefined behavior](#) if the value of the result of this operation cannot be represented as an integer.

```

long num_long;

if (scanf("%ld", &num_long) != 1) {

```

```
    /* Handle error */  
}
```

In general, do not use `scanf()` to parse integers or floating-point numbers from input strings because the input could contain numbers not representable by the argument type.

Compliant Solution (Linux)

This compliant example uses the Linux `scanf()` implementation's built-in error handling to validate input. On Linux platforms, `scanf()` sets `errno` to `ERANGE` if the result of integer conversion cannot be represented within the size specified by the format string [[Linux 2008](#)]. Note that this solution is platform dependent, so it should be used only where portability is not a concern.

```
long num_long;  
errno = 0;  
  
if (scanf("%ld", &num_long) != 1) {  
    /* Handle error */  
}  
else if (ERANGE == errno) {  
    if (puts("number out of range\n") == EOF) {  
        /* Handle error */  
    }  
}
```

Compliant Solution

This compliant example uses `fgets()` to input a string and `strtol()` to convert the string to an integer. Error checking is provided to make sure the value is a valid integer in the range of `long`.

```
char buff[25];  
char *end_ptr;  
long num_long;  
  
if (fgets(buff, sizeof(buff), stdin) == NULL) {  
    if (puts("EOF or read error\n") == EOF) {  
        /* Handle error */  
    }  
} else {  
    errno = 0;  
  
    num_long = strtol(buff, &end_ptr, 10);  
  
    if (ERANGE == errno) {  
        if (puts("number out of range\n") == EOF) {  
            /* Handle error */  
        }  
    }  
    else if (end_ptr == buff) {  
        if (puts("not valid numeric input\n") == EOF) {  
            /* Handle error */  
        }  
    }  
    else if ('\n' != *end_ptr && '\0' != *end_ptr) {  
        if (puts("extra characters on input line\n") == EOF) {  
            /* Handle error */  
        }  
    }  
}
```

Note that this solution treats any trailing characters, including whitespace characters, as an error condition.

Risk Assessment

Although it is relatively rare for a violation of this recommendation to result in a security [vulnerability](#), it can easily result in lost or misinterpreted data.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
INT05-C	Medium	Probable	High	P4	L3

Related Guidelines

SEI CERT C++ Coding Standard	INT05-CPP. Do not use input functions to convert character data if they cannot handle all possible inputs
MITRE CWE	CWE-192 , Integer coercion error CWE-197 , Numeric truncation error

Bibliography

[Klein 2002]	
[Linux 2008]	scanf(3)

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [https://www.securecoding.cert.org/confluence/x/_AQ], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	high
scanf_functions		dict{...}

Possible Messages

Name	Message
scanf_conversion_to_number	Potential numeric overflow: do not use functions of scanf() family to convert a string to number.

CertC-INT07

Use only explicitly signed or unsigned char type for numeric values.

Input: IR

Source languages: C, C++

Details

The three types `char`, `signed char`, and `unsigned char` are collectively called the *character types*. Compilers have the latitude to define `char` to have the same range, representation, and behavior as *either* `signed char` *or* `unsigned char`. Irrespective of the choice made, `char` is a separate type from the other two and is *not* compatible with either.

Use only `signed char` and `unsigned char` types for the storage and use of numeric values because it is the only portable way to guarantee the signedness of the character types (see [STR00-C. Represent characters using an appropriate type](#) for more information on representing characters).

Noncompliant Code Example

In this noncompliant code example, the `char`-type variable `c` may be signed or unsigned. Assuming 8-bit, two's complement character types, this code may print out either `i/c = 5` (`unsigned`) or `i/c = -17` (`signed`). It is much more difficult to reason about the correctness of a program without knowing if these integers are signed or unsigned.

```
char c = 200;
int i = 1000;
printf("i/c = %d\n", i/c);
```

Compliant Solution

In this compliant solution, the variable `c` is declared as `unsigned char`. The subsequent division operation is now independent of the signedness of `char` and consequently has a predictable result.

```
unsigned char c = 200;
int i = 1000;
printf("i/c = %d\n", i/c);
```

Exceptions

[INT07-C-EX1: void FI034-C. Use int to capture the return value of character I/O functions that might be used to check for end of file](#) mentions that certain character I/O functions return a value of type `int`. Despite being returned in an arithmetic type, the value is not actually numeric in nature, so it is acceptable to later store the result into a variable of type `char`.

Risk Assessment

This is a subtle error that results in a disturbingly broad range of potentially severe [vulnerabilities](#). At the very least, this error can lead to unexpected numerical results on different platforms. Unexpected arithmetic values when applied to arrays or pointers can yield buffer overflows or other invalid memory access.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
INT07-C	Medium	Probable	Medium	P8	L2

Related Guidelines

SEI CERT C++ Coding Standard	INT07-CPP. Use only explicitly signed or unsigned char type for numeric values
ISO/IEC TR 24772:2013	Bit Representations [STR]
MISRA C:2012	Rule 10.1 (required) Rule 10.3 (required) Rule 10.4 (required)
MITRE CWE	CWE-682 , Incorrect calculation

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/-As>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
char_used_numerically	Plain char datatype used for non-character data.

CertC-INT08

Verify that all integer values are in range.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Integer operations must result in an integer value within the range of the integer type (that is, the resulting value is the same as the result produced by unlimited-range integers). Frequently, the range is more restrictive depending on the use of the integer value, for example, as an index. Integer values can be verified by code review or by [static analysis](#).

Integer overflow is [undefined behavior](#), so a compiled program can do anything, including go off to play the Game of Life. Furthermore, a compiler may perform optimizations that assume an overflow will never occur, which can easily yield unexpected results. Compilers can optimize away `if` statements that check whether an overflow occurred. See [MSC15-C. Do not depend on undefined behavior](#) for an example.

Verifiably in-range operations are often preferable to treating out-of-range values as an error condition because the handling of these errors has been repeatedly shown to cause [denial-of-service](#) problems in actual applications. The quintessential example is the failure of the Ariane 5 launcher, which occurred because of an improperly handled conversion error that resulted in the processor being shut down [[Lions 1996](#)].

A program that detects an integer overflow to be imminent may do one of two things: (1) signal some sort of error condition or (2) produce an integer result that is within the range of representable integers on that system. Some situations can be handled by an error condition, where an overflow causes a change in control flow (such as the system complaining about bad input and requesting alternative input from the user). Others are better handled by the latter option because it allows the computation to proceed and generate an integer result, thereby avoiding a denial-of-service attack. However, when continuing to produce an integer result in the face of overflow, the question of what integer result to return to the user must be considered.

The saturation and modwrap algorithms and the technique of restricted range usage, defined in the following subsections, produce integer results that are always within a defined range. This range is between the integer values `MIN` and `MAX` (inclusive), where `MIN` and `MAX` are two representable integers with `MIN < MAX`.

Saturation Semantics

For saturation semantics, assume that the mathematical result of the computation is `result`. The value actually returned to the user is set out in the following table:

Range of Mathematical Result	Result Returned
MAX < result	MAX
MIN <= result <= MAX	result
result < MIN	MIN

Modwrap Semantics

In modwrap semantics (also called *modulo* arithmetic), integer values "wrap round." That is, adding 1 to MAX produces MIN. This is the defined behavior for unsigned integers in the C Standard, subclause 6.2.5, paragraph 9. It is frequently the behavior of signed integers, as well. However, it is more sensible in many applications to use saturation semantics instead of modwrap semantics. For example, in the computation of a size (using unsigned integers), it is often better for the size to stay at the maximum value in the event of overflow rather than to suddenly become a very small value.

Restricted Range Usage

Another technique for avoiding integer overflow is to use only half the range of signed integers. For example, when using an `int`, use only the range `[INT_MIN/2, INT_MAX/2]`. This practice has been a trick of the trade in Fortran for some time, and now that optimizing C compilers are more sophisticated, it can be valuable in C.

Consider subtraction. If the user types the expression `a - b`, where both `a` and `b` are in the range `[INT_MIN/2, INT_MAX/2]`, the result will be in the range `(INT_MIN, INT_MAX]` for a typical two's complement machine.

Now, if the user types `a < b`, an implicit subtraction often occurs. On a machine without condition codes, the compiler may simply issue a subtract instruction and check whether the result is negative. This behavior is allowed because the compiler is allowed to assume there is no overflow. If all explicitly user-generated values are kept in the range `[INT_MIN/2, INT_MAX/2]`, then comparisons will always work even if the compiler performs this optimization on such hardware.

Noncompliant Code Example

In this noncompliant example, `i + 1` will overflow on a 16-bit machine. The C Standard allows signed integers to overflow and produce incorrect results. Compilers can take advantage of this to produce faster code by assuming an overflow will not occur. As a result, the `if` statement that is intended to catch an overflow might be optimized away.

```
int i = /* Expression that evaluates to the value 32767 */;
/* ... */
if (i + 1 <= i) {
    /* Handle overflow */
}
/* Expression involving i + 1 */
```

Compliant Solution

Using a `long` instead of an `int` is guaranteed to accommodate the computed value:

```
long i = /* Expression that evaluates to the value 32767 */;
/* ... */
/* No test is necessary; i is known not to overflow */
/* Expression involving i + 1 */
```

Risk Assessment

Out-of-range integer values can result in reading from or writing to arbitrary memory locations and the execution of arbitrary code.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
INT08-C	Medium	Probable	High	P4	L3

Related Guidelines

SEI CERT C++ Coding Standard	INT08-CPP. Verify that all integer values are in range
ISO/IEC TR 24772:2013	Numeric Conversion Errors [FLC]

Bibliography

[Lions 1996]

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/JA4>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
abstract_interpretation_overflow	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
cast_overflow	Cast on result of arithmetic computation may cause overflow
cast_truncate	Cast may truncate value
cast_underflow	Cast on result of arithmetic computation may cause underflow
overflow	Arithmetic computation may cause overflow
underflow	Arithmetic computation may cause underflow

CertC-INT09

Ensure enumeration constants map to unique values.

Input: IR

Source languages: C, C++

Details

A C enumeration defines a type with a finite set of values represented by identifiers known as *enumeration constants*, or enumerators. An enumerator is a constant integer expression whose value is representable as an `int`. Although the language allows multiple enumerators of the same type to have the same value, it is a common expectation that all enumerators of the same type have distinct values. However, defining two or more enumerators of the same type to have the same value can lead to some nonobvious errors.

Noncompliant Code Example

In this noncompliant code example, two enumerators of type `Color` are assigned explicit values. It may not be obvious to the programmer that `yellow` and `indigo` have been declared to be identical values [6], as are `green` and `violet` [7]. Probably the least dangerous error that can result from such a definition is attempting to use the enumerators as labels of a `switch` statement. Because all labels in a `switch` statement are required to be unique, the following code violates this semantic constraint and is required to be diagnosed by a [conforming](#) compiler:

```
enum Color { red=4, orange, yellow, green, blue, indigo=6, violet };

const char* color_name(enum Color col) {
    switch (col) {
        case red: return "red";
        case orange: return "orange";
        case yellow: return "yellow";
        case green: return "green";
        case blue: return "blue";
        case indigo: return "indigo"; /* Error: duplicate label (yellow) */
        case violet: return "violet"; /* Error: duplicate label (green) */
    }
}
```

Compliant Solution

To prevent the error discussed of the noncompliant code example, enumeration type declarations must take one of the following forms:

- Provide no explicit integer assignments, as in this example:

```
enum Color { red, orange, yellow, green, blue, indigo, violet };
```

- Assign a value to the first member only (the rest are then sequential), as in this example:

```
enum Color { red=4, orange, yellow, green, blue, indigo, violet };
```

- Assign a value to all members so any equivalence is explicit, as in this example:

```
enum Color {
    red=4,
    orange=5,
    yellow=6,
```

```

green=7,
blue=8,
indigo=6,
violet=7
};

```

It is also advisable to provide a comment explaining why multiple enumeration type members are being assigned the same value so that future maintainers do not mistakenly identify this form as an error.

Of these three options, providing no explicit integer assignments is the simplest and consequently the preferred approach unless the first enumerator must have a nonzero value.

Exceptions

INT09-C-EX1: In cases where defining an enumeration with two or more enumerators with the same value is intended, the constant expression used to define the value of the duplicate enumerator should reference the enumerator rather than the original enumerator's value. This practice makes the intent clear to both human readers of the code and automated code analysis tools that detect violations of this guideline and would diagnose them otherwise. Note, however, that it does not make it possible to use such enumerators in contexts where unique values are required (such as in a `switch` statement, as discussed earlier).

```
enum Color { red, orange, yellow, green, blue, indigo, violet=indigo };
```

Risk Assessment

Failing to ensure that constants within an enumeration have unique values can result in unexpected results.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
INT09-C	Low	Probable	Medium	P4	L3

Related Guidelines

SEI CERT C++ Coding Standard	INT09-CPP. Ensure enumeration constants map to unique values
CERT Oracle Secure Coding Standard for Java	DCL56-J. Do not attach significance to the ordinal associated with an enum
ISO/IEC TR 24772:2013	Enumerator Issues [CCB]
MISRA C:2012	Rule 8.12 (required)

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/Rg4>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
duplicate_enum_value	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique

CertC-INT12

Do not make assumptions about the type of a plain int bit-field when used in an expression.

Input: IR

Source languages: C, C++

Details

Bit-fields can be used to allow flags or other integer values with small ranges to be packed together to save storage space.

It is [implementation-defined](#) whether the specifier `int` designates the same type as `signed int` or the same type as `unsigned int` for bit-fields. According to the C Standard [[ISO/IEC 9899:2011](#)], C integer promotions also require that "if an `int` can represent all values of the original type (as restricted by the

width, for a bit-field], the value is converted to an `int`; otherwise, it is converted to an `unsigned int`."

This issue is similar to the signedness of plain `char`, discussed in [INT07-C. Use only explicitly signed or unsigned char type for numeric values](#). A plain `int` bit-field that is treated as `unsigned` will promote to `int` as long as its field width is less than that of `int` because `int` can hold all values of the original type. This behavior is the same as that of a plain `char` treated as `unsigned`. However, a plain `int` bit-field treated as `unsigned` will promote to `unsigned int` if its field width is the same as that of `int`. This difference makes a plain `int` bit-field even trickier than a plain `char`.

Bit-field types other than `_Bool`, `int`, `signed int`, and `unsigned int` are implementation-defined. They still obey the integer promotions quoted previously when the specified width is at least as narrow as `CHAR_BIT*sizeof(int)`, but wider bit-fields are not portable.

Noncompliant Code Example

This noncompliant code depends on [implementation-defined behavior](#). It prints either `-1` or `255`, depending on whether a plain `int` bit-field is signed or `unsigned`.

```
struct {
    int a: 8;
} bits = {255};

int main(void) {
    printf("bits.a = %d.\n", bits.a);
    return 0;
}
```

Compliant Solution

This compliant solution uses an `unsigned int` bit-field and does not depend on implementation-defined behavior:

```
struct {
    unsigned int a: 8;
} bits = {255};

int main(void) {
    printf("bits.a = %d.\n", bits.a);
    return 0;
}
```

Risk Assessment

Making invalid assumptions about the type of a bit-field or its layout can result in unexpected program flow.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
INT12-C	Low	Unlikely	Medium	P2	L3

Related Guidelines

SEI CERT C++ Coding Standard	INT12-CPP. Do not make assumptions about the type of a plain int bit-field when used in an expression
ISO/IEC TR 24772:2013	Bit Representations [STR]
MISRA C:2012	Rule 10.1 (required)

Bibliography

[ISO/IEC 9899:2011]	Subclause 6.3.1.1, "Boolean, Characters, and Integers"
-------------------------------------	--

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/RAE>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
accept_system_typedef	Whether standard typedefs like <code>int32_t</code> are accepted if from a system header even when they do not use an explicit sign for the underlying type	True
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
missing_bitfield_sign	Bit-field type should be an explicitly unsigned or signed integral type.

CertC-INT13

Use bitwise operators only on unsigned operands.

Input: IR

Source languages: C, C++

Details

Bitwise operators include the complement operator `~`, bitwise shift operators `>>` and `<<`, bitwise AND operator `&`, bitwise exclusive OR operator `^`, bitwise inclusive OR operator `|` and compound assignment operators `>=`, `<<=`, `&=`, `^=` and `|=`. Bitwise operators should be used only with unsigned integer operands, as the results of bitwise operations on signed integers are [implementation-defined](#).

The C11 standard, section 6.5, paragraph 4 [[ISO/IEC 9899:2011](#)], states:

Some operators (the unary operator `~`, and the binary operators `<<`, `>>`, `&`, `^`, and `|`, collectively described as bitwise operators) shall have operands that have integral type. These operators return values that depend on the internal representations of integers, and thus have implementation-defined and undefined aspects for signed types.

Furthermore, the bitwise shift operators `<<` and `>>` are undefined under many circumstances, and are implementation-defined for signed integers for more circumstances; see rule [INT34-C. Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand](#) for more information.

Implementation details

The Microsoft C compiler documentation says that:

Bitwise operations on signed integers work the same as bitwise operations on unsigned integers.

On-line GCC documentation about the implementation of bitwise operations on signed integers says:

Bitwise operators act on the representation of the value including both the sign and value bits, where the sign bit is considered immediately above the highest-value value bit.

Noncompliant Code Example (Right Shift)

The right-shift operation may be implemented as either an arithmetic (signed) shift or a logical (unsigned) shift. If `E1` in the expression `E1 >> E2` has a signed type and a negative value, the resulting value is implementation-defined. Also, a bitwise shift can result in [undefined behavior](#). (See [INT34-C. Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand](#).)

This noncompliant code example can result in an error condition on [implementations](#) in which an arithmetic shift is performed, and the sign bit is propagated as the number is shifted [[Dowd 2006](#)]:

```
int rc = 0;
int stringify = 0x80000000;
char buf[sizeof("256")];
rc = sprintf(buf, sizeof(buf), "%u", stringify >> 24);
if (rc == -1 || rc >= sizeof(buf)) {
    /* Handle error */
}
```

In this example, `stringify >> 24` evaluates to `0xFFFFFFF80`, or 4,294,967,168. When converted to a string, the resulting value "4294967168" is too large to store in `buf` and is truncated by `sprintf()`.

If this code had been implemented using `sprintf()` instead of `snprintf()`, this noncompliant code example would have resulted in a buffer overflow.

Compliant Solution (Right Shift)

In this compliant solution, `stringify` is declared as an unsigned integer. The value of the result of the right-shift operation is the integral part of the quotient of `stringify / 2 ^ 24`:

```
int rc = 0;
unsigned int stringify = 0x80000000;
char buf[sizeof("256")];
rc = sprintf(buf, sizeof(buf), "%u", stringify >> 24);
if (rc == -1 || rc >= sizeof(buf)) {
    /* Handle error */
}
```

Also, consider using the `sprintf_s()` function, defined in ISO/IEC TR 24731-1, instead of `snprintf()` to provide some additional checks. (See [STR07-C. Use the bounds-checking interfaces for string manipulation](#).)

Exceptions

INT13-C-EX1: When used as bit flags, it is acceptable to use preprocessor macros or enumeration constants as arguments to the `&` and `|` operators even if the value is not explicitly declared as unsigned.

```
fd = open(file_name, UO_WRONLY | UO_CREAT | UO_EXCL | UO_TRUNC, 0600);
```

INT13-C-EX2: If the right-side operand to a shift operator is known at compile time, it is acceptable for the value to be represented with a signed type provided it is positive.

```
#define SHIFT 24
foo = 15u >> SHIFT;
```

Risk Assessment

Performing bitwise operations on signed numbers can lead to buffer overflows and the execution of arbitrary code by an attacker in some cases, unexpected or implementation-defined behavior in others.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
INT13-C	High	Unlikely	Medium	P6	L2

Related Guidelines

SEI CERT C++ Coding Standard	INT13-CPP. Use bitwise operators only on unsigned operands
ISO/IEC TR 24772:2013	Bit Representations [STR] Arithmetic Wrap-around Error [FIF] Sign Extension Error [XZI]
MITRE CWE	CWE-682 , Incorrect calculation

Bibliography

[Dowd 2006]	Chapter 6, "C Language Issues"
[C99 Rationale 2003]	Subclause 6.5.7, "Bitwise Shift Operators"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/BoAD>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
bitop_on_enum	Bitwise operator on enum underlying type
bitop_on_signed	Bitwise operator on signed underlying type

CertC-INT15

Use `intmax_t` or `uintmax_t` for formatted I/O on programmer-defined integer types.

Input: IR

Source languages: C, C++

Details

Few programmers consider the issues around formatted I/O and type definitions. A programmer-defined integer type might be any type supported by the [implementation](#), even a type larger than `unsigned long long`. For example, given an implementation that supports 128-bit unsigned integers and provides a `uint_fast128_t` type, a programmer may define the following type:

```
typedef uint_fast128_t mytypedef_t;
```

Furthermore, the definition of programmer-defined types may change, which creates a problem when these types are used with formatted output functions, such as `printf()`, and formatted input functions, such as `scanf()`. (See [FI047-C. Use valid format strings](#).)

The C `intmax_t` and `uintmax_t` types can represent any value representable by any other integer types of the same signedness. (See [INT00-C. Understand the data model used by your implementation\(s\)](#).) This capability allows conversion between programmer-defined integer types (of the same signedness) and `intmax_t` and `uintmax_t`:

```
mytypedef_t x;
uintmax_t temp;

temp = x; /* Always secure if mytypedef_t is unsigned*/

/* ... Change the value of temp ... */

if (temp <= MYTYPEDEF_MAX) {
    x = temp;
}
```

Formatted I/O functions can be used to input and output greatest-width integer typed values. The `j` length modifier in a format string indicates that the following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier will apply to an argument with type `intmax_t` or `uintmax_t`. C also specifies the `z` length modifier for use with arguments of type `size_t` and the `t` length modifier for arguments of type `ptrdiff_t`.

In addition to programmer-defined types, there is no requirement that an implementation provide format-length modifiers for [implementation-defined](#) integer types. For example, a machine with an implementation-defined 48-bit integer type may not provide format-length modifiers for the type. Such a machine still must have a 64-bit `long long`, with `intmax_t` being at least that large.

Noncompliant Code Example (`printf()`)

This noncompliant code example prints the value of `x` as an `unsigned long long` value even though the value is of a programmer-defined integer type:

```
#include <stdio.h>

mytypedef_t x;

/* ... */

printf("%llu", (unsigned long long) x);
```

There is no guarantee that this code prints the correct value of `x`, as `x` may be too large to represent as an `unsigned long long`.

Compliant Solution (`printf()`)

The C `intmax_t` and `uintmax_t` can be safely used to perform formatted I/O with programmer-defined integer types by converting signed programmer-defined integer types to `intmax_t` and unsigned programmer-defined integer types to `uintmax_t`, then outputting these values using the `j` length modifier. Similarly, programmer-defined integer types can be input to variables of `intmax_t` or `uintmax_t` (whichever matches the signedness of the programmer-defined integer type) and then converted to programmer-defined integer types using appropriate range checks.

This compliant solution guarantees that the correct value of `x` is printed, regardless of its length, provided that `mytypedef_t` is an unsigned type:

```
#include <stdio.h>
#include <inttypes.h>

mytypedef_t x;

/* ... */

printf("%ju", (uintmax_t) x);
```

Compliant Solution (Microsoft `printf()`)

Visual Studio 2012 and earlier versions do not support the standard `j` length modifier and do not have a nonstandard analog. Consequently, the programmer must hard code the knowledge that `intmax_t` is `int64_t` and `uintmax_t` is `uint64_t` for Microsoft Visual Studio versions.

```
#include <stdio.h>
#include <inttypes.h>

mytypedef_t x;

/* ... */

#ifndef _MSC_VER
    printf("%llu", (uintmax_t) x);
#else
    printf("%ju", (uintmax_t) x);
#endif
```

A feature request has been submitted to Microsoft to add support for the `j` length modifier to a future release of Microsoft Visual Studio.

Noncompliant Code Example (`scanf()`)

This noncompliant code example reads an `unsigned long long` value from standard input and stores the result in `x`, which is of a programmer-defined integer type:

```
#include <stdio.h>

mytypedef_t x;
/* ... */
if (scanf("%llu", &x) != 1) {
    /* Handle error */
```

}

This noncompliant code example can result in a buffer overflow if the size of `mytypedef_t` is smaller than `unsigned long long`, or it might result in an incorrect value if the size of `mytypedef_t` is larger than `unsigned long long`. Moreover, `scanf()` lacks the error checking capabilities of alternative conversion routines, such as `strtol()`. For more information, see [INT06-C. Use strtol\(\) or a related function to convert a string token to an integer](#).

Compliant Solution [`strtoumax()`]

This compliant solution guarantees that a correct value in the range of `mytypedef_t` is read, or an error condition is detected, assuming the value of `MYTYPEDEF_MAX` is correct as the largest value representable by `mytypedef_t`: The `strtoumax()` function is used instead of `scanf()` as it provides enhanced error checking functionality. The `fgets()` function is used to read input from `stdin`.

```
#include <stdio.h>
#include <inttypes.h>
#include <errno.h>

mytypedef_t x;
uintmax_t temp;

/* ... */
if (fgets(buff, sizeof(buff), stdin) == NULL) {
    if (puts("EOF or read error\n") == EOF) {
        /* Handle error */
    }
} else {
    /* Check for errors in the conversion */
    errno = 0;
    temp = strtoumax(buff, &end_ptr, 10);
    if (ERANGE == errno) {
        if (puts("number out of range\n") == EOF) {
            /* Handle error */
        }
    } else if (end_ptr == buff) {
        if (puts("not valid numeric input\n") == EOF) {
            /* Handle error */
        }
    } else if ('\n' != *end_ptr && '\0' != *end_ptr) {
        if (puts("extra characters on input line\n") == EOF) {
            /* Handle error */
        }
    }
}

/* No conversion errors, attempt to store the converted value into x */
if (temp > MYTYPEDEF_MAX) {
    /* Handle error */
} else {
    x = temp;
}
```

Risk Assessment

Failure to use an appropriate conversion specifier when inputting or outputting programmer-defined integer types can result in buffer overflow and lost or misinterpreted data.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
INT15-C	High	Unlikely	Medium	P6	L2

Related Guidelines

SEI CERT C++ Coding Standard	INT15-CPP. Use intmax_t or uintmax_t for formatted I/O on programmer-defined integer types
MITRE CWE	CWE-681 , Incorrect conversion between numeric types

Bibliography

[Saks 2007c]	Standard C's Pointer Difference Type
------------------------------	--------------------------------------

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/LIAc>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
allow_gnu_extensions		True
check_for_j_modifier		False
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
use_intmax_t	Use intmax_t or uintmax_t for formatted IO on programmer-defined integer types.

CertC-INT17

Define integer constants in an implementation-independent manner.

Input: IR

Source languages: C, C++

Details

Integer constants are often used as masks or specific bit values. Frequently, these constants are expressed in hexadecimal form to indicate to the programmer how the data might be represented in the machine. However, hexadecimal integer constants are frequently used in a nonportable manner.

Noncompliant Code Example

In this pedagogical noncompliant code example, the `flipbits()` function complements the value stored in `x` by performing a bitwise exclusive OR against a mask with all bits set to 1. For implementations where `unsigned long` is represented by a 32-bit value, each bit of `x` is correctly complemented.

```
/* (Incorrect) Set all bits in mask to 1 */
const unsigned long mask = 0xFFFFFFFF;

unsigned long flipbits(unsigned long x) {
    return x ^ mask;
}
```

However, on implementations where values of type `unsigned long` are represented by greater than 32 bits, `mask` will have leading 0s. For example, on implementations where values of type `unsigned long` are 64 bits long, `mask` is assigned the value `0x00000000FFFFFFFF`. Consequently, only the lower-order bits of `x` are complemented.

Compliant Solution (-1)

In this compliant solution, the integer constant `-1` is used to set all bits in `mask` to 1. The integer constant `-1` is of type `signed int`. Because `-1` cannot be represented by a variable of type `unsigned long`, it is converted to a representable number according to the rule in subclause 6.3.1.3, paragraph 2, of the C Standard [[ISO/IEC 9899:2011](#)]:

[If the value can't be represented by the new type and] if the new type is `unsigned`, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.

"One more than the maximum value that can be represented in the new type," `ULONG_MAX + 1`, is added to `-1`, resulting in a right-side value of `ULONG_MAX`. The representation of `ULONG_MAX` is guaranteed to have all bits set to 1 by subclause 6.2.6.2, paragraph 1:

For unsigned integer types other than `unsigned char`, the bits of the object representation shall be divided into two groups: value bits and padding bits (there need not be any of the latter). If there are N value bits, each bit shall represent a different power of 2 between 1 and $2^N - 1$, so that objects of that type shall be capable of representing values from 0 to $2^N - 1$ using a **pure binary representation**; this shall be known as the value representation. The values of any padding bits are unspecified.

By the same reasoning, `-1` is suitable for setting all bits to one of any unsigned integer variable. Subclause 6.2.6.1, paragraph 3, guarantees the same results for `unsigned char`:

Values stored in unsigned bit-fields and objects of type `unsigned char` shall be represented using a **pure binary notation**.

```
/* (Correct) Set all bits in mask to 1 */
const unsigned long mask = -1;

unsigned long flipbits(unsigned long x) {
    return x ^ mask;
}
```

Noncompliant Code Example

In this noncompliant code example, a programmer attempts to set the most significant bit:

```
const unsigned long mask = 0x80000000;
unsigned long x;

/* Initialize x */
x |= mask;
```

This code has the desired effect for implementations where `unsigned long` has a precision of 32 bits but not for implementations where `unsigned long` has a precision of 64 bits.

Compliant Solution

A portable (and safer) way of setting the high-order bit is to use a shift expression, as in this compliant solution:

```
const unsigned long mask = ~(ULONG_MAX >> 1);
unsigned long x;

/* Initialize x */
x |= mask;
```

Risk Assessment

[Vulnerabilities](#) are frequently introduced while porting code. A buffer overflow vulnerability may result, for example, if an incorrectly defined integer constant is used to determine the size of a buffer. It is always best to write portable code, especially when there is no performance overhead for doing so.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
INT17-C	High	Probable	Low	P18	L1

Related Guidelines

SEI CERT C++ Coding Standard	INT17-CPP. Define integer constants in an implementation-independent manner
--	---

Bibliography

[Dewhurst 2002]	Gotcha #25, "#define Literals"
[ISO/IEC 9899:2011]	Subclause 6.2.6, "Representations of Types" Subclause 6.3.1.3, "Signed and Unsigned Integers"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/agHsAQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
non_portable_int	This constant definition may be non-portable.

CertC-INT34

Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand.

Input: IR

Source languages: C, C++

Details

Bitwise shifts include left-shift operations of the form *shift-expression* \ll *additive-expression* and right-shift operations of the form *shift-expression* \gg *additive-expression*. The standard integer promotions are first performed on the operands, each of which has an integer type. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is [undefined](#). (See [undefined behavior 51](#).)

Do not shift an expression by a negative number of bits or by a number greater than or equal to the *precision* of the promoted left operand. The precision of an integer type is the number of bits it uses to represent values, excluding any sign and padding bits. For unsigned integer types, the width and the precision are the same; whereas for signed integer types, the width is one greater than the precision. This rule uses precision instead of width because, in almost every case, an attempt to shift by a number of bits greater than or equal to the precision of the operand indicates a bug (logic error). A logic error is different from overflow, in which there is simply a representational deficiency. In general, shifts should be performed only on unsigned operands. (See [INT13-C. Use bitwise operators only on unsigned operands](#).)

Noncompliant Code Example [Left Shift, Unsigned Type]

The result of `E1 << E2` is `E1` left-shifted `E2` bit positions; vacated bits are filled with zeros.

According to the C Standard, if `E1` has an unsigned type, the value of the result is $E1 * 2^{E2}$, reduced modulo 1 more than the maximum value representable in the result type.

This noncompliant code example fails to ensure that the right operand is less than the precision of the promoted left operand:

```
void func(unsigned int ui_a, unsigned int ui_b) {
    unsigned int urestult = ui_a << ui_b;
    /* ... */
}
```

Compliant Solution [Left Shift, Unsigned Type]

This compliant solution eliminates the possibility of shifting by greater than or equal to the number of bits that exist in the precision of the left operand:

```
#include <limits.h>
#include <stddef.h>
#include <inttypes.h>

extern size_t __popcount(uintmax_t);
#define PRECISION(x) __popcount(x)

void func(unsigned int ui_a, unsigned int ui_b) {
    unsigned int urestult = 0;
    if (ui_b >= PRECISION(UINT_MAX)) {
        /* Handle error */
    } else {
        urestult = ui_a << ui_b;
    }
    /* ... */
}
```

The `PRECISION()` macro and `popcount()` function provide the correct precision for any integer type. (See [INT35-C. Use correct integer precisions](#).)

Modulo behavior resulting from left-shifting an unsigned integer type is permitted by exception INT30-EX3 to [INT30-C. Ensure that unsigned integer operations do not wrap](#).

Noncompliant Code Example [Left Shift, Signed Type]

The result of `E1 << E2` is `E1` left-shifted `E2` bit positions; vacated bits are filled with zeros. If `E1` has a signed type and nonnegative value, and $E1 * 2^{E2}$ is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

This noncompliant code example fails to ensure that left and right operands have nonnegative values and that the right operand is less than the precision of the promoted left operand. This example does check for signed integer overflow in compliance with [INT32-C. Ensure that operations on signed integers do not result in overflow](#).

```
#include <limits.h>
#include <stddef.h>
#include <inttypes.h>

void func(signed long si_a, signed long si_b) {
    signed long result;
    if (si_a > (LONG_MAX >> si_b)) {
        /* Handle error */
    } else {
        result = si_a << si_b;
    }
    /* ... */
}
```

Shift operators and other bitwise operators should be used only with unsigned integer operands in accordance with [INT13-C. Use bitwise operators only on unsigned operands](#).

Compliant Solution [Left Shift, Signed Type]

In addition to the check for overflow, this compliant solution ensures that both the left and right operands have nonnegative values and that the right operand is less than the precision of the promoted left operand:

```
#include <limits.h>
#include <stddef.h>
#include <inttypes.h>
```

```

extern size_t __popcount(uintmax_t);
#define PRECISION(x) __popcount(x)

void func(signed long si_a, signed long si_b) {
    signed long result;
    if ((si_a < 0) || (si_b < 0) ||
        (si_b >= PRECISION(ULONG_MAX)) ||
        (si_a > (LONG_MAX >> si_b))) {
        /* Handle error */
    } else {
        result = si_a << si_b;
    }
    /* ... */
}

```

Noncompliant Code Example (Right Shift)

The result 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 nonnegative value, the value of the result is the integral part of the quotient of `E1 / 2E2`. If `E1` has a signed type and a negative value, the resulting value is [implementation-defined](#) and can be either an arithmetic (signed) shift or a logical (unsigned) shift.

This noncompliant code example fails to test whether the right operand is greater than or equal to the precision of the promoted left operand, allowing undefined behavior:

```

void func(unsigned int ui_a, unsigned int ui_b) {
    unsigned int urestult = ui_a >> ui_b;
    /* ... */
}

```

When working with signed operands, making assumptions about whether a right shift is implemented as an arithmetic (signed) shift or a logical (unsigned) shift can also lead to [vulnerabilities](#). (See [INT13-C. Use bitwise operators only on unsigned operands](#).)

Compliant Solution (Right Shift)

This compliant solution eliminates the possibility of shifting by greater than or equal to the number of bits that exist in the precision of the left operand:

```

#include <limits.h>
#include <stddef.h>
#include <inttypes.h>

extern size_t __popcount(uintmax_t);
#define PRECISION(x) __popcount(x)

void func(unsigned int ui_a, unsigned int ui_b) {
    unsigned int urestult = 0;
    if (ui_b >= PRECISION(UINT_MAX)) {
        /* Handle error */
    } else {
        urestult = ui_a >> ui_b;
    }
    /* ... */
}

```

Implementation Details

GCC has no options to handle shifts by negative amounts or by amounts outside the width of the type predictably or to trap on them; they are always treated as undefined. Processors may reduce the shift amount modulo the width of the type. For example, 32-bit right shifts are implemented using the following instruction on x86-32:

```
sarl %cl, %eax
```

The `sarl` instruction takes a bit mask of the least significant 5 bits from `%cl` to produce a value in the range [0, 31] and then shift `%eax` that many bits:

```
// 64-bit right shifts on IA-32 platforms become
shrdl %edx, %eax
sarl %cl, %edx
```

where `%eax` stores the least significant bits in the doubleword to be shifted, and `%edx` stores the most significant bits.

Risk Assessment

Although shifting a negative number of bits or shifting a number of bits greater than or equal to the width of the promoted left operand is undefined behavior in C, the risk is generally low because processors frequently reduce the shift amount modulo the width of the type.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT34-C	Low	Unlikely	Medium	P2	L3

Related Guidelines

SEI CERT C Coding Standard	INT13-C. Use bitwise operators only on unsigned operands INT35-C. Use correct integer precisions INT32-C. Ensure that operations on signed integers do not result in overflow
ISO/IEC TR 24772:2013	Arithmetic Wrap-Around Error [FIF]

Bibliography

[C99 Rationale 2003]	6.5.7, "Bitwise Shift Operators"
[Dowd 2006]	Chapter 6, "C Language Issues"
[Seacord 2013b]	Chapter 5, "Integer Security"
[Viega 2005]	Section 5.2.7, "Integer Overflow"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/IRe>], Copyright [C] 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
report_nonconst_cases	Whether a message should be printed for non-constant cases missing a range-checking if statement	True
use_essential_type	Whether essential type should be computed for allowed value range.	False

Possible Messages

Name	Message
excessive_shift	Right operand of shift operator has value {}, but should be in range 0 to {}
possible_excessive_shift	Right operand of shift operator should be checked to be in range 0 to {}
shift_right_negative	If first operand is negative, using this bitwise operator relies on implementation-defined behaviour

CertC-INT36

Converting a pointer to integer or integer to pointer.

Input: IR

Source languages: C, C++

Details

Although programmers often use integers and pointers interchangeably in C, pointer-to-integer and integer-to-pointer conversions are [implementation-defined](#).

Conversions between integers and pointers can have undesired consequences depending on the [implementation](#). According to the C Standard, subclause 6.3.2.3 [[ISO/IEC 9899:2011](#)],

An integer may be converted to any pointer type. Except as previously specified, the result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation.

Any pointer type may be converted to an integer type. Except as previously specified, the result is implementation-defined. If the result cannot be represented in the integer type, the behavior is undefined. The result need not be in the range of values of any integer type.

Do not convert an integer type to a pointer type if the resulting pointer is incorrectly aligned, does not point to an entity of the referenced type, or is a [trap representation](#).

Do not convert a pointer type to an integer type if the result cannot be represented in the integer type. (See [undefined behavior 24](#).)

The mapping between pointers and integers must be consistent with the addressing structure of the execution environment. Issues may arise, for example, on architectures that have a segmented memory model.

Noncompliant Code Example

The size of a pointer can be greater than the size of an integer, such as in an implementation where pointers are 64 bits and unsigned integers are 32 bits. This code example is noncompliant on such implementations because the result of converting the 64-bit `ptr` cannot be represented in the 32-bit integer type:

```
void f(void) {
    char *ptr;
    /* ... */
    unsigned int number = (unsigned int)ptr;
    /* ... */
}
```

Compliant Solution

Any valid pointer to `void` can be converted to `intptr_t` or `uintptr_t` and back with no change in value. (See [INT36-EX2](#).) The C Standard guarantees that a pointer to `void` may be converted to or from a pointer to any object type and back again and that the result must compare equal to the original pointer. Consequently, converting directly from a `char *` pointer to a `uintptr_t`, as in this compliant solution, is allowed on implementations that support the `uintptr_t` type.

```
#include <stdint.h>

void f(void) {
    char *ptr;
    /* ... */
    uintptr_t number = (uintptr_t)ptr;
    /* ... */
}
```

Noncompliant Code Example

In this noncompliant code example, the pointer `ptr` is converted to an integer value. The high-order 9 bits of the number are used to hold a flag value, and the result is converted back into a pointer. This example is noncompliant on an implementation where pointers are 64 bits and unsigned integers are 32 bits because the result of converting the 64-bit `ptr` cannot be represented in the 32-bit integer type.

```
void func(unsigned int flag) {
    char *ptr;
    /* ... */
    unsigned int number = (unsigned int)ptr;
    number = (number & 0x7fffff) | (flag << 23);
    ptr = (char *)number;
}
```

A similar scheme was used in early versions of Emacs, limiting its portability and preventing the ability to edit files larger than 8MB.

Compliant Solution

This compliant solution uses a `struct` to provide storage for both the pointer and the flag value. This solution is portable to machines of different word sizes, both smaller and larger than 32 bits, working even when pointers cannot be represented in any integer type.

```
struct ptrflag {
    char *pointer;
    unsigned int flag : 9;
} ptrflag;

void func(unsigned int flag) {
    char *ptr;
    /* ... */
    ptrflag.pointer = ptr;
    ptrflag.flag = flag;
}
```

Noncompliant Code Example

It is sometimes necessary to access memory at a specific location, requiring a literal integer to pointer conversion. In this noncompliant code, a pointer is set directly to an integer constant, where it is unknown whether the result will be as intended:

```
unsigned int *g(void) {
    unsigned int *ptr = 0xdeadbeef;
    /* ... */
    return ptr;
}
```

The result of this assignment is [implementation-defined](#), might not be correctly aligned, might not point to an entity of the referenced type, and might be a [trap representation](#).

Compliant Solution

Adding an explicit cast may help the compiler convert the integer value into a valid pointer. A common technique is to assign the integer to a volatile-qualified object of type `intptr_t` or `uintptr_t` and then assign the integer value to the pointer:

```
unsigned int *g(void) {
    volatile uintptr_t iptr = 0xdeadbeef;
    unsigned int *ptr = (unsigned int *)iptr;
    /* ... */
    return ptr;
}
```

Exceptions

INT36-C-EX1: A null pointer can be converted to an integer; it takes on the value 0. Likewise, the integer value 0 can be converted to a pointer; it becomes the null pointer.

INT36-C-EX2: Any valid pointer to void can be converted to intptr_t or uintptr_t or their underlying types and back again with no change in value. Use of underlying types instead of intptr_t or uintptr_t is discouraged, however, because it limits portability.

```
#include <assert.h>
#include <stdint.h>

void h(void) {
    intptr_t i = (intptr_t)(void *)&i;
    uintptr_t j = (uintptr_t)(void *)&j;

    void *ip = (void *)i;
    void *jp = (void *)j;

    assert(ip == &i);
    assert(jp == &j);
}
```

Risk Assessment

Converting from pointer to integer or vice versa results in code that is not portable and may create unexpected pointers to invalid memory locations.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT36-C	Low	Probable	High	P2	L3

Related Guidelines

SEI CERT C++ Coding Standard	INT11-CPP. Take care when converting from pointer to integer or integer to pointer
ISO/IEC TR 24772:2013	Pointer Casting and Pointer Type Changes [HFC]
ISO/IEC TS 17961:2013	Converting a pointer to integer or integer to pointer [intptrconv]
MITRE CWE	CWE-466 , Return of Pointer Value Outside of Expected Range CWE-587 , Assignment of a Fixed Address to a Pointer

Bibliography

[ISO/IEC 9899:2011]	6.3.2.3, "Pointers"
-------------------------------------	---------------------

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/XAAV>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high
reported_messages	If provided, only messages of these types are reported.	[767, 152, 1053]
reported_severities	List of severities to display.	('error', 'warning', 'remark')
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

CertC-FLP02

Avoid using floating-point numbers when precise computation is needed.

Input: IR

Details

Computers can represent only a finite number of digits. It is therefore impossible to precisely represent repeating binary-representation values such as 1/3 or 1/5 with the most common floating-point representation: binary floating point.

When precise computation is necessary, use alternative representations that can accurately represent the values. For example, if you are performing arithmetic on decimal values and need an exact decimal rounding, represent the values in binary-coded decimal instead of using floating-point values. Another option is decimal floating-point arithmetic, as specified by ANSI/IEEE 754-2007. ISO/IEC WG14 has drafted a proposal to add support for decimal floating-point arithmetic to the C language [[ISO/IEC DTR 24732](#)].

When precise computation is necessary, carefully and methodically estimate the maximum cumulative error of the computations, regardless of whether decimal or binary is used, to ensure that the resulting error is within tolerances. Consider using numerical analysis to properly understand the problem. An introduction can be found in David Goldberg's "What Every Computer Scientist Should Know about Floating-Point Arithmetic" [[Goldberg 1991](#)].

Noncompliant Code Example

This noncompliant code example takes the mean of 10 identical numbers and checks to see if the mean matches this number. It should match because the 10 numbers are all 10.1. Yet, because of the imprecision of floating-point arithmetic, the computed mean does not match this number.

```
#include <stdio.h>

/* Returns the mean value of the array */
float mean(float array[], int size) {
    float total = 0.0;
    size_t i;
    for (i = 0; i < size; i++) {
        total += array[i];
        printf("array[%zu] = %f and total is %f\n", i, array[i], total);
    }
    if (size != 0)
        return total / size;
    else
        return 0.0;
}

enum { array_size = 10 };
float array_value = 10.1;

int main(void) {
    float array[array_size];
    float avg;
    size_t i;
    for (i = 0; i < array_size; i++) {
        array[i] = array_value;
    }

    avg = mean(array, array_size);
    printf("mean is %f\n", avg);
    if (avg == array[0])
        printf("array[0] is the mean\n");
    else
        printf("array[0] is not the mean\n");
    return 0;
}
```

On a 64-bit Linux machine using GCC 4.1, this program yields the following output:

```
array[0] = 10.100000 and total is 10.100000
array[1] = 10.100000 and total is 20.200001
array[2] = 10.100000 and total is 30.300001
array[3] = 10.100000 and total is 40.400002
array[4] = 10.100000 and total is 50.500000
array[5] = 10.100000 and total is 60.599998
array[6] = 10.100000 and total is 70.699997
array[7] = 10.100000 and total is 80.799995
array[8] = 10.100000 and total is 90.899994
array[9] = 10.100000 and total is 100.999992
mean is 10.099999
array[0] is not the mean
```

Compliant Solution

The noncompliant code can be fixed by replacing the floating-point numbers with integers for the internal additions. Floats are used only when printing results and when doing the division to compute the mean.

```
#include <stdio.h>

/* Returns the mean value of the array */
float mean(int array[], int size) {
    int total = 0;
    size_t i;
    for (i = 0; i < size; i++) {
        total += array[i];
        printf("array[%zu] = %f and total is %f\n", i, array[i] / 100.0, total / 100.0);
    }
    if (size != 0)
        return ((float) total) / size;
    else
        return 0.0;
```

```

}

enum {array_size = 10};
int array_value = 1010;

int main(void) {
    int array[array_size];
    float avg;
    size_t i;
    for (i = 0; i < array_size; i++) {
        array[i] = array_value;
    }

    avg = mean(array, array_size);
    printf("mean is %f\n", avg / 100.0);
    if (avg == array[0]) {
        printf("array[0] is the mean\n");
    } else {
        printf("array[0] is not the mean\n");
    }
    return 0;
}

```

On a 64-bit Linux machine using GCC 4.1, this program yields the following expected output:

```

array[0] = 10.100000 and total is 10.100000
array[1] = 10.100000 and total is 20.200000
array[2] = 10.100000 and total is 30.300000
array[3] = 10.100000 and total is 40.400000
array[4] = 10.100000 and total is 50.500000
array[5] = 10.100000 and total is 60.600000
array[6] = 10.100000 and total is 70.700000
array[7] = 10.100000 and total is 80.800000
array[8] = 10.100000 and total is 90.900000
array[9] = 10.100000 and total is 101.000000
mean is 10.100000
array[0] is the mean

```

Risk Assessment

Using a representation other than floating point may allow for more accurate results.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
FLP02-C	Low	Probable	High	P2	L3

Related Guidelines

SEI CERT C++ Coding Standard	FLP02-CPP. Avoid using floating-point numbers when precise computation is needed
CERT Oracle Secure Coding Standard for Java	NUM04-J. Do not use floating-point numbers if precise computation is required
ISO/IEC TR 24772:2013	Floating-point Arithmetic [PLF]

Bibliography

[Goldberg 1991]
[IEEE 754 2006]
[ISO/IEC DTR 24732]
[ISO/IEC JTC1/SC22/WG11]

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/DgU>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
float_equality_comparison	Testing a float for (in)equality.
float_indirect_equality_comparison	Indirectly testing a float for (in)equality.

CertC-FLP04

Check floating-point inputs for exceptional values.

Input: IR

Source languages: C, C++

Details

Floating-point numbers can take on two classes of exceptional values; infinity and NaN (not-a-number). These values are returned as the result of exceptional or otherwise unresolvable floating-point operations. (See also [FLP32-C. Prevent or detect domain and range errors in math functions](#).) Additionally, they can be directly input by a user by `scanf` or similar functions. Failure to detect and handle such values can result in [undefined behavior](#).

NaN values are particularly problematic because the expression `NaN == NaN` (for every possible value of NaN) returns false. Any comparisons made with NaN as one of the arguments returns false, and all arithmetic functions on NaNs simply propagate them through the code. Hence, a NaN entered in one location in the code and not properly handled could potentially cause problems in other, more distant sections.

Formatted-input functions such as `scanf` will accept the values `INF`, `INFINITY`, or `NAN` (case insensitive) as valid inputs for the `%f` format specification, allowing malicious users to feed them directly to a program. Programs should therefore check to ensure that all input floating-point values (especially those controlled by the user) have neither of these values if doing so would be inappropriate. The `<math.h>` library provides two macros for this purpose: `isinf` and `isnan`.

`isinf` and `isnan`

The `isinf` macro tests an input floating-point value for infinity. `isinf(val)` is nonzero if `val` is an infinity (positive or negative), and 0 otherwise.

`isnan` tests if an input is NaN. `isnan(val)` is nonzero if `val` is a NaN, and 0 otherwise.

If infinity or NaN values are not acceptable inputs in a program, these macros should be used to ensure they are not passed to vulnerable functions.

Noncompliant Code Example

This noncompliant code example accepts user data without first validating it:

```
float currentBalance; /* User's cash balance */
void doDeposit() {
    float val;

    scanf("%f", &val);

    if(val >= MAX_VALUE - currentBalance) {
        /* Handle range error */
    }

    currentBalance += val;
}
```

This can be a problem if an invalid value is entered for `val` and subsequently used for calculations or as control values. The user could, for example, input the strings `"INF"`, `"INFINITY"`, or `"NAN"` (case insensitive) on the command line, which would be parsed by `scanf` into the floating-point representations of infinity and NaN. All subsequent calculations using these values would be invalid, possibly crashing the program and enabling a [denial-of-service attack](#).

Here, for example, entering `"nan"` for `val` would force `currentBalance` to also equal `"nan"`, corrupting its value. If this value is used elsewhere for calculations, every resulting value would also be a NaN, possibly destroying important data.

Implementation Details

The following code was run on 32-bit GNU Linux using the GCC 3.4.6 compiler. On this platform, `FLT_MAX` has the value 340282346638528859811704183484516925440.000000.

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    float val, currentBalance=0;
    scanf("%f", &val);
    currentBalance+=val;
    printf("%f\n", currentBalance);
    return 0;
}
```

The following table shows the value of `currentBalance` returned for various arguments:

Input	currentBalance
25	25.00000
infinity	inf
inf	inf
-infinity	-inf
NaN	nan
nan	nan
1e9999	inf
-1e9999	-inf

As this example demonstrates, the user can enter the exceptional values `infinity` and `NaN`, as well as force a float's value to be infinite, by entering out-of-range floats. These entries subsequently corrupt the value of `currentBalance`. So by entering exceptional floats, an attacker can corrupt the program data, possibly leading to a crash.

Compliant Solution

This compliant solution first validates the input float before using it. The value is tested to ensure that it is neither an infinity nor a NaN.

```
float currentBalance; /* User's cash balance */

void doDeposit() {
    float val;

    scanf("%f", &val);
    if (isinf(val)) {
        /* Handle infinity error */
    }
    if (isnan(val)) {
        /* Handle NaN error */
    }
    if (val >= MAX_VALUE - currentBalance) {
        /* Handle range error */
    }

    currentBalance += val;
}
```

Exceptions

Occasionally, NaN or infinity may be acceptable or expected inputs to a program. If this is the case, then explicit checks may not be necessary. Such programs must, however, be prepared to handle these inputs gracefully and not blindly use them in mathematical expressions where they are not appropriate.

Risk Assessment

Inappropriate floating-point inputs can result in invalid calculations and unexpected results, possibly leading to crashing and providing a [denial-of-service](#) opportunity.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
FLP04-C	Low	Probable	High	P2	L3

Related Guidelines

SEI CERT C++ Coding Standard	FLP04-CPP. Check floating-point inputs for exceptional values
CERT Oracle Secure Coding Standard for Java	FLP06-J. Check floating-point inputs for exceptional values

Bibliography

[IEEE 754]
[IEEE Std 1003.1:2013]

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/XAGyAQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	high
scanf_functions		dict{...}

Possible Messages

Name	Message
uncleared_float_value	Check floating-point inputs for exceptional values before use.

CertC-FLP06

Convert integers to floating point for floating-point operations.

Input: IR

Source languages: C, C++

Details

Using integer arithmetic to calculate a value for assignment to a floating-point variable may lead to loss of information. This problem can be avoided by converting one of the integers in the expression to a floating type.

When converting integers to floating-point values, and vice versa, it is important to carry out proper range checks to avoid undefined behavior (see [FLP34-C. Ensure that floating-point conversions are within range of the new type](#)).

Noncompliant Code Example

In this noncompliant code example, the division and multiplication operations take place on integers and are then converted to floating point. Consequently, floating-point variables `d`, `e`, and `f` are not initialized correctly because the operations take place before the values are converted to floating-point values. The results are truncated to the nearest integer or may overflow.

```
void func(void) {
    short a = 533;
    int b = 6789;
    long c = 466438237;

    float d = a / 7; /* d is 76.0 */
    double e = b / 30; /* e is 226.0 */
    double f = c * 789; /* f may be negative due to overflow */
}
```

Compliant Solution (Floating-Point Literal)

In this compliant solution, the decimal error in initialization is eliminated by ensuring that at least one of the operands to the division operation is floating point:

```
void func(void) {
    short a = 533;
    int b = 6789;
    long c = 466438237;

    float d = a / 7.0f; /* d is 76.14286 */
    double e = b / 30.; /* e is 226.3 */
    double f = (double)c * 789; /* f is 368019768993.0 */
}
```

Compliant Solution (Conversion)

In this compliant solution, the decimal error in initialization is eliminated by first storing the integer in the floating-point variable and then performing the arithmetic operation. This practice ensures that at least one of the operands is a floating-point number and that the subsequent arithmetic operation is performed on floating-point operands.

```
void func(void) {
    short a = 533;
    int b = 6789;
    long c = 466438237;

    float d = a;
    double e = b;
    double f = c;

    d /= 7; /* d is 76.14286 */
}
```

```

e /= 30; /* e is 226.3 */
f *= 789; /* f is 368019768993.0 */
}

```

Exceptions

FLP06-C-EX0: It may be desirable to have the operation take place as integers before the conversion (obviating the need for a call to `trunc()`, for example). If this is the programmer's intention, it should be clearly documented to help future maintainers understand that this behavior is intentional.

Risk Assessment

Improper conversions between integers and floating-point values may yield unexpected results, especially loss of precision. Additionally, these unexpected results may actually involve overflow, or undefined behavior.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
FLP06-C	Low	Probable	Low	P6	L2

Related Guidelines

CERT C Secure Coding Standard	FLP34-C. Ensure that floating-point conversions are within range of the new type
SEI CERT C++ Coding Standard	FLP05-CPP. Convert integers to floating point for floating point operations
CERT Oracle Secure Coding Standard for Java	NUM50-J. Convert integers to floating point for floating-point operations
MITRE CWE	CWE-681 , Incorrect conversion between numeric types CWE-682 , Incorrect calculation

Bibliography

[Hatton 1995]	Section 2.7.3, "Floating-Point Misbehavior"
-------------------------------	---

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/YAAV>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, BoolTypes, EnumTypes, CharacterTypes], [FloatingTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Implicit conversion from integral to float type

CertC-FLP07

Cast the return value of a function that returns a floating-point type.

Input: IR

Source languages: C, C++

Details

Cast the return value of a function that returns a floating point type to ensure predictable program execution.

Subclause 6.8.6.4, paragraph 3, of the C Standard [[ISO/IEC 9899:2011](#)] states:

If a return statement with an expression is executed, the value of the expression is returned to the caller as the value of the function call expression. If the expression has a type different from the return type of the function in which it appears, the value is converted as if by assignment to an object having the return type of the function.

This paragraph is annotated [footnote 160] as follows:

The return statement is not an assignment. The overlap restriction of subclause 6.5.16.1 does not apply to the case of function return. The representation of floating-point values may have wider range or precision than implied by the type; a cast may be used to remove this extra range and precision.

Conversion as if by assignment to the type of the function is required if the return expression has a different type than the function but not if the return expression has a wider value only because of wide evaluation. This allows seemingly inconsistent and confusing behavior. Consider the following code as an example:

```
float f(float x) {
    return x * 0.1f;
}

float g(float x) {
    return x * 0.1;
}
```

Function `f()` is allowed to return a value wider than `float`, but function `g()` (which uses the wider constant) is not.

Although the standard does not require narrowing return expressions of the same type as the function, it does not clearly state what is allowed. Is it allowed to narrow the result? Is it allowed to narrow the result sometimes but not always? Is it allowed to partially narrow the result (for example, if the application binary interface [ABI] returns floats in double format, but a float function has a float return expression evaluated to wider than double)? An aggressive implementation could argue yes to all these questions, though the resulting behavior would complicate debugging and error analysis.

Footnote 160 in the C Standard says a cast may be used to remove extra range and precision from the return expression. This means that a predictable program must have casts on all function calls that return floating-point values (except where the function directly feeds an operator-like assignment that implies the conversion). With type-generic math (`tgmath.h`), the programmer has to reason through the `tgmath.h` resolution rules to determine which casts to apply. These are significant obstacles to writing predictable code.

The C Standard, subclause F.6 [[ISO/IEC 9899:2011](#)], states:

If the return expression is evaluated in a floating-point format different from the return type, the expression is converted as if by assignment³⁶² to the return type of the function and the resulting value is returned to the caller.

362) Assignment removes any extra range and precision.

This applies only to implementations that conforms to the optional Annex F, "IEC 60559 Floating-Point Arithmetic." The macro `__STDC_IEC_559__` can be used to determine whether an implementation conforms to Annex F.

Noncompliant Code Example

This noncompliant code example fails to cast the result of the expression in the return statement and thereby guarantee that the range or precision is no wider than expected. The uncertainty in this example is introduced by the constant `0.1f`. This constant may be stored with a range or precision that is greater than that of `float`. Consequently, the result of `x * 0.1f` may also have a range or precision greater than that of `float`. As described previously, this range or precision may not be reduced to that of a `float`, so the caller of `calcPercentage()` may receive a value that is more precise than expected. This may lead to inconsistent program execution across different platforms.

```
float calc_percentage(float value) {
    return value * 0.1f;
}

void float_routine(void) {
    float value = 99.0f;
    long double percentage;

    percentage = calc_percentage(value);
}
```

Compliant Solution (within the Function)

This compliant solution casts the value of the expression in the return statement. It forces the return value to have the expected range and precision, as described in subclause 5.2.4.2.2, paragraph 9, of the C Standard [[ISO/IEC 9899:2011](#)].

```

float calc_percentage(float value) {
    return (float)(value * 0.1f);
}

void float_routine(void) {
    float value = 99.0f;
    long double percentage;

    percentage = calc_percentage(value);
}

```

Forcing the range and precision inside the `calcPercentage()` function is a good way to fix the problem once without having to apply fixes in multiple locations (every time `calcPercentage()` is called).

Compliant Solution (Outside the Function)

Source code to the called function may not always be available. This compliant solution casts the return value of the `calcPercentage()` function to `float` to force the correct range and precision when the source of the called function cannot be modified.

```

void float_routine(void) {
    float value = 99.0f;
    long double percentage;

    percentage = (float) calc_percentage(value);
}

```

Risk Assessment

Failure to follow this guideline can lead to inconsistent results across different platforms.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FLP07-C	Low	Probable	Medium	P4	L3

Bibliography

[ISO/IEC 9899:2011]	Subclause 6.8.6.4, "The <code>return</code> Statement" Annex F.6, "The <code>return</code> Statement"
[WG14/N1396]	

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/6oAtAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
wide_float_call	Function may return floating value wider than declared type, use cast to {}.
wide_float_return	Returned floating value may be wider than declared type, use cast to {}.

CertC-FLP30

Do not use floating-point variables as loop counters.

Input: IR

Source languages: C, C++

Details

Because floating-point numbers represent real numbers, it is often mistakenly assumed that they can represent any simple fraction exactly. Floating-point

numbers are subject to representational limitations just as integers are, and binary floating-point numbers cannot represent all real numbers exactly, even if they can be represented in a small number of decimal digits.

In addition, because floating-point numbers can represent large values, it is often mistakenly assumed that they can represent all significant digits of those values. To gain a large dynamic range, floating-point numbers maintain a fixed number of precision bits (also called the significand) and an exponent, which limit the number of significant digits they can represent.

Different implementations have different precision limitations, and to keep code portable, floating-point variables must not be used as the loop induction variable. See Goldberg's work for an introduction to this topic [[Goldberg 1991](#)].

For the purpose of this rule, a *loop counter* is an induction variable that is used as an operand of a comparison expression that is used as the controlling expression of a `do`, `while`, or `for` loop. An *induction variable* is a variable that gets increased or decreased by a fixed amount on every iteration of a loop [[Aho 1986](#)]. Furthermore, the change to the variable must occur directly in the loop body (rather than inside a function executed within the loop).

Noncompliant Code Example

In this noncompliant code example, a floating-point variable is used as a loop counter. The decimal number `0.1` is a repeating fraction in binary and cannot be exactly represented as a binary floating-point number. Depending on the implementation, the loop may iterate 9 or 10 times.

```
void func(void) {
    for (float x = 0.1f; x <= 1.0f; x += 0.1f) {
        /* Loop may iterate 9 or 10 times */
    }
}
```

For example, when compiled with GCC or Microsoft Visual Studio 2013 and executed on an x86 processor, the loop is evaluated only nine times.

Compliant Solution

In this compliant solution, the loop counter is an integer from which the floating-point value is derived:

```
#include <stddef.h>

void func(void) {
    for (size_t count = 1; count <= 10; ++count) {
        float x = count / 10.0f;
        /* Loop iterates exactly 10 times */
    }
}
```

Noncompliant Code Example

In this noncompliant code example, a floating-point loop counter is incremented by an amount that is too small to change its value given its precision:

```
void func(void) {
    for (float x = 100000001.0f; x <= 100000010.0f; x += 1.0f) {
        /* Loop may not terminate */
    }
}
```

On many implementations, this produces an infinite loop.

Compliant Solution

In this compliant solution, the loop counter is an integer from which the floating-point value is derived. The variable `x` is assigned a computed value to reduce compounded rounding errors that are present in the noncompliant code example.

```
void func(void) {
    for (size_t count = 1; count <= 10; ++count) {
        float x = 100000000.0f + (count * 1.0f);
        /* Loop iterates exactly 10 times */
    }
}
```

Risk Assessment

The use of floating-point variables as loop counters can result in [unexpected behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FLP30-C	Low	Probable	Low	P6	L2

Related Guidelines

SEI CERT C++ Coding Standard	FLP30-CPP. Do not use floating-point variables as loop counters
CERT Oracle Secure Coding Standard for Java	NUM09-J. Do not use floating-point variables as loop counters
ISO/IEC TR 24772:2013	Floating-Point Arithmetic [PLF]
MISRA C:2012	Directive 1.1 (required) Rule 14.1 (required)

Bibliography

[Aho 1986]	
[Goldberg 1991]	
[Lockheed Martin 05]	AV Rule 197

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [https://www.securecoding.cert.org/confluence/x/AoG_/], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
loop_counter_model		CertLoopCounters
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
float_loop_counter	Use of floating-point loop counter.

CertC-FLP32

Prevent or detect domain and range errors in math functions.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

The C Standard, 7.12.1 [[ISO/IEC 9899:2011](#)], defines three types of errors that relate specifically to math functions in `<math.h>`. Paragraph 2 states

A domain error occurs if an input argument is outside the domain over which the mathematical function is defined.

Paragraph 3 states

A pole error (also known as a singularity or infinitary) occurs if the mathematical function has an exact infinite result as the finite input argument(s) are approached in the limit.

Paragraph 4 states

A range error occurs if the mathematical result of the function cannot be represented in an object of the specified type, due to extreme magnitude.

An example of a domain error is the square root of a negative number, such as `sqrt(-1.0)`, which has no meaning in real arithmetic. Contrastingly, 10 raised to the 1-millionth power, `pow(10., 1e6)`, cannot be represented in many floating-point [implementations](#) because of the limited range of the type `double` and consequently constitutes a range error. In both cases, the function will return some value, but the value returned is not the correct result of the computation. An example of a pole error is `log(0.0)`, which results in negative infinity.

Programmers can prevent domain and pole errors by carefully bounds-checking the arguments before calling mathematical functions and taking alternative action if the bounds are violated.

Range errors usually cannot be prevented because they are dependent on the implementation of floating-point numbers as well as on the function being applied. Instead of preventing range errors, programmers should attempt to detect them and take alternative action if a range error occurs.

The following table lists the `double` forms of standard mathematical functions, along with checks that should be performed to ensure a proper input domain, and indicates whether they can also result in range or pole errors, as reported by the C Standard. Both `float` and `long double` forms of these functions also exist but are omitted from the table for brevity. If a function has a specific domain over which it is defined, the programmer must check its input values. The programmer must also check for range errors where they might occur. The standard math functions not listed in this table, such as `fabs()`, have no domain restrictions and cannot result in range or pole errors.

Function	Domain	Range	Pole
acos(x)	-1 <= x && x <= 1	No	No
asin(x)	-1 <= x && x <= 1	Yes	No
atan(x)	None	Yes	No
atan2(y, x)	x != 0 && y != 0	No	No
acosh(x)	x >= 1	Yes	No
asinh(x)	None	Yes	No
atanh(x)	-1 < x && x < 1	Yes	Yes
cosh(x), sinh(x)	None	Yes	No
exp(x), exp2(x), expm1(x)	None	Yes	No
ldexp(x, exp)	None	Yes	No
log(x), log10(x), log2(x)	x >= 0	No	Yes
log1p(x)	x >= -1	No	Yes
ilogb(x)	x != 0 && !isinf(x) && !isnan(x)	Yes	No
logb(x)	x != 0	Yes	Yes
scalbn(x, n), scalbln(x, n)	None	Yes	No
hypot(x, y)	None	Yes	No
pow(x,y)	x > 0 (x == 0 && y > 0) (x < 0 && y is an integer)	Yes	Yes
sqrt(x)	x >= 0	No	No
erf(x)	None	Yes	No
erfc(x)	None	Yes	No
lgamma(x), tgamma(x)	x != 0 && [x < 0 && x is an integer]	Yes	Yes
lrint(x), lround(x)	None	Yes	No
fmod(x, y), remainder(x, y), remquo(x, y, quo)	y != 0	Yes	No
nextafter(x, y), nexttoward(x, y)	None	Yes	No
fdim(x,y)	None	Yes	No
fma(x,y,z)	None	Yes	No

Domain and Pole Checking

The most reliable way to handle domain and pole errors is to prevent them by checking arguments beforehand, as in the following exemplar:

```
double safe_sqrt(double x) {
    if (x < 0) {
        fprintf(stderr, "sqrt requires a nonnegative argument");
        /* Handle domain / pole error */
    }
    return sqrt (x);
}
```

Range Checking

Programmers usually cannot prevent range errors, so the most reliable way to handle them is to detect when they have occurred and act accordingly.

The exact treatment of error conditions from math functions is tedious. The C Standard, 7.12.1 [\[ISO/IEC 9899:2011\]](#), defines the following behavior for floating-point overflow:

A floating result overflows if the magnitude of the mathematical result is finite but so large that the mathematical result cannot be represented without extraordinary roundoff error in an object of the specified type. If a floating result overflows and default rounding is in effect, then the

function returns the value of the macro `HUGE_VAL`, `HUGE_VALF`, or `HUGE_VALL` according to the return type, with the same sign as the correct value of the function; if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, the integer expression `errno` acquires the value `ERANGE`; if the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero, the "overflow" floating-point exception is raised.

It is preferable not to check for errors by comparing the returned value against `HUGE_VAL` or `0` for several reasons:

- These are, in general, valid (albeit unlikely) data values.
- Making such tests requires detailed knowledge of the various error returns for each math function.
- Multiple results aside from `HUGE_VAL` and `0` are possible, and programmers must know which are possible in each case.
- Different versions of the library have varied in their error-return behavior.

It can be unreliable to check for math errors using `errno` because an [implementation](#) might not set `errno`. For real functions, the programmer determines if the implementation sets `errno` by checking whether `math_errhandling & MATH_ERRNO` is nonzero. For complex functions, the C Standard, 7.3.2, paragraph 1, simply states that "an implementation may set `errno` but is not required to" [\[ISO/IEC 9899:2011\]](#).

The obsolete *System V Interface Definition* (SVID3) [\[UNIX 1992\]](#) provides more control over the treatment of errors in the math library. The programmer can define a function named `matherr()` that is invoked if errors occur in a math function. This function can print diagnostics, terminate the execution, or specify the desired return value. The `matherr()` function has not been adopted by C or POSIX, so it is not generally portable.

The following error-handling template uses C Standard functions for floating-point errors when the C macro `math_errhandling` is defined and indicates that they should be used; otherwise, it examines `errno`:

```
#include <math.h>
#include <fenv.h>
#include <errno.h>

/* ... */
/* Use to call a math function and check errors */
{
    #pragma STDC FENV_ACCESS ON

    if ((math_errhandling & MATH_ERREXCEPT) {
        feclearexcept(FE_ALL_EXCEPT);
    }
    errno = 0;

    /* Call the math function */

    if (((math_errhandling & MATH_ERRNO) && errno != 0) {
        /* Handle range error */
    } else if (((math_errhandling & MATH_ERREXCEPT) &&
               fetestexcept(FE_INVALID | FE_DIVBYZERO |
                           FE_OVERFLOW | FE_UNDERFLOW) != 0) {
        /* Handle range error */
    }
}
```

See [FLP03-C. Detect and handle floating-point errors](#) for more details on how to detect floating-point errors.

Subnormal Numbers

A subnormal number is a nonzero number that does not use all of its precision bits [\[IEEE 754 2006\]](#). These numbers can be used to represent values that are closer to 0 than the smallest normal number (one that uses all of its precision bits). However, the `asin()`, `asinh()`, `atan()`, `atanh()`, and `erf()` functions may produce range errors, specifically when passed a subnormal number. When evaluated with a subnormal number, these functions can produce an inexact, subnormal value, which is an underflow error. The C Standard, 7.12.1, paragraph 6 [\[ISO/IEC 9899:2011\]](#), defines the following behavior for floating-point underflow:

The result underflows if the magnitude of the mathematical result is so small that the mathematical result cannot be represented, without extraordinary roundoff error, in an object of the specified type. If the result underflows, the function returns an implementation-defined value whose magnitude is no greater than the smallest normalized positive number in the specified type; if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, whether `errno` acquires the value `ERANGE` is implementation-defined; if the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero, whether the "underflow" floating-point exception is raised is implementation-defined.

Implementations that support floating-point arithmetic but do not support subnormal numbers, such as IBM S/360 hex floating-point or nonconforming IEEE-754 implementations that skip subnormals (or support them by flushing them to zero), can return a range error when calling one of the following families of functions with the following arguments:

- `fmod((min+subnorm), min)`
- `remainder((min+subnorm), min)`
- `remquo((min+subnorm), min, quo)`

where `min` is the minimum value for the corresponding floating point type and `subnorm` is a subnormal value.

If Annex F is supported and subnormal results are supported, the returned value is exact and a range error cannot occur. The C Standard, F.10.7.1 [\[ISO/IEC 9899:2011\]](#), specifies the following for the `fmod()`, `remainder()`, and `remquo()` functions:

When subnormal results are supported, the returned value is exact and is independent of the current rounding direction mode.

Annex F, subclause F.10.7.2, paragraph 2, and subclause F.10.7.3, paragraph 2, of the C Standard identify when subnormal results are supported.

Noncompliant Code Example (`sqrt()`)

This noncompliant code example determines the square root of `x`:

```
#include <math.h>
void func(double x) {
```

```
    double result;
    result = sqrt(x);
}
```

However, this code may produce a domain error if x is negative.

Compliant Solution (`sqrt()`)

Because this function has domain errors but no range errors, bounds checking can be used to prevent domain errors:

```
#include <math.h>

void func(double x) {
    double result;

    if (isless(x, 0.0)) {
        /* Handle domain error */
    }

    result = sqrt(x);
}
```

Noncompliant Code Example (`sinh()`, Range Errors)

This noncompliant code example determines the hyperbolic sine of x :

```
#include <math.h>

void func(double x) {
    double result;
    result = sinh(x);
}
```

This code may produce a range error if x has a very large magnitude.

Compliant Solution (`sinh()`, Range Errors)

Because this function has no domain errors but may have range errors, the programmer must detect a range error and act accordingly:

```
#include <math.h>
#include <fenv.h>
#include <errno.h>

void func(double x) {
    double result;
    {

        #pragma STDC FENV_ACCESS ON
        if (math_errhandling & MATH_ERREXCEPT) {
            feclearexcept(FE_ALL_EXCEPT);
        }
        errno = 0;

        result = sinh(x);

        if ((math_errhandling & MATH_ERRNO) && errno != 0) {
            /* Handle range error */
        } else if ((math_errhandling & MATH_ERREXCEPT) &&
                   fetestexcept(FE_INVALID | FE_DIVBYZERO |
                               FE_OVERFLOW | FE_UNDERFLOW) != 0) {
            /* Handle range error */
        }
    }

    /* Use result... */
}
```

Noncompliant Code Example (`pow()`)

This noncompliant code example raises x to the power of y :

```
#include <math.h>

void func(double x, double y) {
    double result;
    result = pow(x, y);
}
```

This code may produce a domain error if x is negative and y is not an integer value or if x is 0 and y is 0. A domain error or pole error may occur if x is 0 and y is negative, and a range error may occur if the result cannot be represented as a `double`.

Compliant Solution (`pow()`)

Because the `pow()` function can produce domain errors, pole errors, and range errors, the programmer must first check that x and y lie within the proper domain and do not generate a pole error and then detect whether a range error occurs and act accordingly:

```
#include <math.h>
#include <fenv.h>
```

```

#include <errno.h>

void func(double x, double y) {
    double result;

    if (((x == 0.0f) && islesseq(y, 0.0)) || isless(x, 0.0)) {
        /* Handle domain or pole error */
    }

    {
        #pragma STDC FENV_ACCESS ON
        if (math_errhandling & MATH_ERREXCEPT) {
            feclearexcept(FE_ALL_EXCEPT);
        }
        errno = 0;

        result = pow(x, y);

        if ((math_errhandling & MATH_ERRNO) && errno != 0) {
            /* Handle range error */
        } else if ((math_errhandling & MATH_ERREXCEPT) &&
                   fetestexcept(FE_INVALID | FE_DIVBYZERO |
                               FE_OVERFLOW | FE_UNDERFLOW) != 0) {
            /* Handle range error */
        }
    }

    /* Use result... */
}

```

Noncompliant Code Example {asin(), Subnormal Number}

This noncompliant code example determines the inverse sine of x:

```

#include <math.h>

void func(float x) {
    float result = asin(x);
    /* ... */
}

```

Compliant Solution {asin(), Subnormal Number}

Because this function has no domain errors but may have range errors, the programmer must detect a range error and act accordingly:

```

#include <math.h>
#include <fenv.h>
#include <errno.h>
void func(float x) {
    float result;

    {
        #pragma STDC FENV_ACCESS ON
        if (math_errhandling & MATH_ERREXCEPT) {
            feclearexcept(FE_ALL_EXCEPT);
        }
        errno = 0;

        result = asin(x);

        if ((math_errhandling & MATH_ERRNO) && errno != 0) {
            /* Handle range error */
        } else if ((math_errhandling & MATH_ERREXCEPT) &&
                   fetestexcept(FE_INVALID | FE_DIVBYZERO |
                               FE_OVERFLOW | FE_UNDERFLOW) != 0) {
            /* Handle range error */
        }
    }

    /* Use result... */
}

```

Risk Assessment

Failure to prevent or detect domain and range errors in math functions may cause unexpected results.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FLP32-C	Medium	Probable	Medium	P8	L2

Related Guidelines

CERT C Secure Coding Standard	FLP03-C. Detect and handle floating-point errors
MITRE CWE	CWE-682, Incorrect Calculation

Bibliography

[ISO/IEC 9899:2011]	7.3.2, "Conventions" 7.12.1, "Treatment of Error Conditions" F.10.7, "Remainder Functions"
[IEEE 754 2006]	
[Plum 1985]	Rule 2-2
[Plum 1989]	Topic 2.10, "conv-Conversions and Overflow"
[UNIX 1992]	<i>System V Interface Definition</i> (SVID3)

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/rqQ>], Copyright [C] 1995-2016 Carnegie Mellon University. See *axivion_copyright_guide.pdf* for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
math_functions	Functions which require a proper input domain check.	dict{...}
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
domain_check_expected	Prevent or detect domain and range errors in math functions.
literal_outside_domain	Literal used is outside of math function's domain.

CertC-FLP37

Do not use object representations to compare floating-point values.

Input: IR

Source languages: C, C++

Details

The object representation for floating-point values is implementation defined. However, an implementation that defines the `__STDC_IEC_559__` macro shall conform to the IEC 60559 floating-point standard and uses what is frequently referred to as IEEE 754 floating-point arithmetic [[ISO/IEC 9899:2011](#)]. The floating-point object representation used by IEC 60559 is one of the most common floating-point object representations in use today.

All floating-point object representations use specific bit patterns to encode the value of the floating-point number being represented. However, equivalence of floating-point values is not encoded solely by the bit pattern used to represent the value. For instance, if the floating-point format supports negative zero values (as IEC 60559 does), the values `-0.0` and `0.0` are equivalent and will compare as equal, but the bit patterns used in the object representation are not identical. Similarly, if two floating-point values are both (the same) NaN, they will not compare as equal, despite the bit patterns being identical, because they are not equivalent.

Do not compare floating-point object representations directly, such as by calling `memcmp()` or its moral equivalents. Instead, the equality operators (`==` and `!=`) should be used to determine if two floating-point values are equivalent.

Noncompliant Code Example

In this noncompliant code example, `memcmp()` is used to compare two structures for equality. However, since the structure contains a floating-point object, this code may not behave as the programmer intended.

```
#include <stdbool.h>
#include <string.h>

struct S {
    int i;
    float f;
};

bool are_equal(const struct S *s1, const struct S *s2) {
    if (!s1 && !s2)
        return true;
    else if (!s1 || !s2)
        return false;
    return 0 == memcmp(s1, s2, sizeof(struct S));
}
```

}

Compliant Solution

In this compliant solution, the structure members are compared individually:

```
#include <stdbool.h>
#include <string.h>

struct S {
    int i;
    float f;
};

bool are_equal(const struct S *s1, const struct S *s2) {
    if (!s1 && !s2)
        return true;
    else if (!s1 || !s2)
        return false;
    return s1->i == s2->i &&
           s1->f == s2->f;
}
```

Risk Assessment

Using the object representation of a floating-point value for comparisons can lead to incorrect equality results, which can lead to unexpected behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FLP37-C	Low	Unlikely	Medium	P2	L3

Bibliography

[ISO/IEC 9899:2011] Annex F, "IEC 60559 floating-point arithmetic"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/J4DK>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
allow_char	Whether to allow memcmp on 'char' type.	True
allow_composites_without_padding	Whether to allow using memcmp on structs and unions that have no padding bytes.	True
allow_float	Whether to allow memcmp on floating point types.	False
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
disallowed_memcmp_pointer_arg	Disallowed type of pointer argument. (disabled)
memcmp_char_pointer_arg	memcmp shall not be used with char pointer argument, use strncmp instead.
memcmp_float	memcmp shall not be used to compare floats as the same value may be stored using different representations.
memcmp_padding	memcmp shall not be used to compare structs with padding. (disabled)
memcmp_struct_pointer_arg	memcmp shall not be used with struct pointer argument as it would compare padding as well. (disabled)
memcmp_union_pointer_arg	memcmp shall not be used with union pointer argument as it would compare padding and different kinds of representation. (disabled)

CertC-ARR01

Do not apply the `sizeof` operator to a pointer when taking the size of an array.

Input: IR

Source languages: C, C++

Details

The `sizeof` operator yields the size (in bytes) of its operand, which can be an expression or the parenthesized name of a type. However, using the `sizeof` operator to determine the size of arrays is error prone.

The `sizeof` operator is often used in determining how much memory to allocate via `malloc()`. However using an incorrect size is a violation of [MEM35-C. Allocate sufficient memory for an object](#).

Noncompliant Code Example

In this noncompliant code example, the function `clear()` zeros the elements in an array. The function has one parameter declared as `int array[]` and is passed a static array consisting of 12 `int` as the argument. The function `clear()` uses the idiom `sizeof(array) / sizeof(array[0])` to determine the number of elements in the array. However, `array` has a pointer type because it is a parameter. As a result, `sizeof(array)` is equal to the `sizeof(int *)`. For example, on an architecture (such as IA-32) where the `sizeof(int) == 4` and the `sizeof(int *) == 4`, the expression `sizeof(array) / sizeof(array[0])` evaluates to 1, regardless of the length of the array passed, leaving the rest of the array unaffected.

```
void clear(int array[]) {
    for (size_t i = 0; i < sizeof(array) / sizeof(array[0]); ++i) {
        array[i] = 0;
    }
}

void dowork(void) {
    int dis[12];

    clear(dis);
    /* ... */
}
```

Footnote 103 in subclause 6.5.3.4 of the C Standard [[ISO/IEC 9899:2011](#)] applies to all array parameters:

When applied to a parameter declared to have array or function type, the `sizeof` operator yields the size of the adjusted (pointer) type.

Compliant Solution

In this compliant solution, the size of the array is determined inside the block in which it is declared and passed as an argument to the function:

```
void clear(int array[], size_t len) {
    for (size_t i = 0; i < len; i++) {
        array[i] = 0;
    }
}

void dowork(void) {
    int dis[12];

    clear(dis, sizeof(dis) / sizeof(dis[0]));
    /* ... */
}
```

This `sizeof(array) / sizeof(array[0])` idiom will succeed provided the original definition of `array` is visible.

Noncompliant Code Example

In this noncompliant code example, `sizeof(a)` does not equal `100 * sizeof(int)`, because the `sizeof` operator, when applied to a parameter declared to have array type, yields the size of the adjusted (pointer) type even if the parameter declaration specifies a length:

```
enum {ARR_LEN = 100};

void clear(int a[ARR_LEN]) {
    memset(a, 0, sizeof(a)); /* Error */
}

int main(void) {
    int b[ARR_LEN];
    clear(b);
    assert(b[ARR_LEN / 2]==0); /* May fail */
    return 0;
}
```

Compliant Solution

In this compliant solution, the size is specified using the expression `len * sizeof(int)`:

```
enum {ARR_LEN = 100};

void clear(int a[], size_t len) {
    memset(a, 0, len * sizeof(int));
}

int main(void) {
    int b[ARR_LEN];
```

```

    clear(b, ARR_LEN);
    assert(b[ARR_LEN / 2]==0); /* Cannot fail */
    return 0;
}

```

Risk Assessment

Incorrectly using the `sizeof` operator to determine the size of an array can result in a buffer overflow, allowing the execution of arbitrary code.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
ARR01-C	High	Probable	Low	P18	L1

Related Guidelines

SEI CERT C++ Coding Standard	CTR01-CPP. Do not apply the sizeof operator to a pointer when taking the size of an array
MITRE CWE	CWE-467 , Use of sizeof() on a pointer type
ISO/IEC TS 17961	Taking the size of a pointer to determine the size of the pointed-to type [sizeofptr]

Bibliography

[Drepper 2006]	Section 2.1.1, "Respecting Memory Bounds"
[ISO/IEC 9899:2011]	Subclause 6.5.3.4, "The sizeof and _Alignof Operators"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/6wE>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
sizeof_on_array_param	Operand of "sizeof" shall not be an array parameter

CertC-ARR02

Explicitly specify array bounds, even if implicitly defined by an initializer.

Input: IR

Source languages: C, C++

Details

The C Standard allows an array variable to be declared both with a bound and with an initialization literal. The initialization literal also implies an array bound in the number of elements specified.

The size implied by an initialization literal is usually specified by the number of elements,

```
int array[] = {1, 2, 3}; /* 3-element array */
```

but it is also possible to use designators to initialize array elements in a noncontiguous fashion. Subclause 6.7.9, Example 12, of the C Standard [[ISO/IEC 9899:2011](#)] states:

Space can be "allocated" from both ends of an array by using a single designator:

```
int a[MAX] = {
    1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0
};
```

In the above, if `MAX` is greater than ten, there will be some zero-valued elements in the middle; if it is less than ten, some of the values provided by the first five initializers will be overridden by the second five.

The C Standard also dictates how array initialization is handled when the number of initialization elements does not equal the explicit array bound. Subclause 6.7.9, paragraphs 21 and 22, state:

If there are fewer initializers in a brace-enclosed list than there are elements or members of an aggregate, or fewer characters in a string literal used to initialize an array of known size than there are elements in the array, the remainder of the aggregate shall be initialized implicitly the same as objects that have static storage duration.

If an array of unknown size is initialized, its size is determined by the largest indexed element with an explicit initializer. The array type is completed at the end of its initializer list.

Although compilers can compute the size of an array on the basis of its initialization list, explicitly specifying the size of the array provides a redundancy check, ensuring that the array's size is correct. It also enables compilers to emit warnings if the array's size is less than the size implied by the initialization.

Note that this recommendation does not apply (in all cases) to character arrays initialized with string literals. See [STR11-C. Do not specify the bound of a character array initialized with a string literal](#) for more information.

Noncompliant Code Example (Incorrect Size)

This noncompliant code example initializes an array of integers using an initialization with too many elements for the array:

```
int a[3] = {1, 2, 3, 4};
```

The size of the array `a` is 3, although the size of the initialization is 4. The last element of the initialization (4) is ignored. Most compilers will diagnose this error.

Implementation Details

This noncompliant code example generates a warning in GCC. Microsoft Visual Studio generates a fatal diagnostic: `error C2078: too many initializers`.

Noncompliant Code Example (Implicit Size)

In this example, the compiler allocates an array of four integer elements and, because an array bound is not explicitly specified by the programmer, sets the array bound to 4. However, if the initializer changes, the array bound may also change, causing unexpected results.

```
int a[] = {1, 2, 3, 4};
```

Compliant Solution

This compliant solution explicitly specifies the array bound:

```
int a[4] = {1, 2, 3, 4};
```

Explicitly specifying the array bound, although it is implicitly defined by an initializer, allows a compiler or other static analysis tool to issue a diagnostic if these values do not agree.

Exceptions

ARR02-C-EX1: [STR11-C. Do not specify the bound of a character array initialized with a string literal](#) is a specific exception to this recommendation; it requires that the bound of a character array initialized with a string literal is unspecified.

Risk Assessment

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
ARR02-C	Medium	Unlikely	Low	P6	L2

Related Guidelines

SEI CERT C++ Coding Standard	CTR02-CPP. Explicitly specify array bounds, even if implicitly defined by an initializer
MITRE CWE	CWE-665 , Incorrect or incomplete initialization
MISRA C:2012	Rule 8.11 (advisory) Rule 9.5 (required)

Bibliography

[ISO/IEC 9899:2011]	Subclause 6.7.9, "Initialization"
-------------------------------------	-----------------------------------

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/HQEQAQ>], Copyright [C] 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
only_extern	Whether to consider only extern declared arrays	False
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low
report_definitions	Whether definitions of array variables should also be reported	True
reported_messages	If provided, only messages of these types are reported.	[1162, 146]
reported_severities	List of severities to display.	('error', 'warning', 'remark')
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

Possible Messages

Name	Message
unbounded_array	Array declaration with unknown size.

CertC-ARR30

Do not form or use out-of-bounds pointers or array subscripts.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

The C Standard identifies the following distinct situations in which undefined behavior (UB) can arise as a result of invalid pointer operations:

UB	Description	Example Code
46	Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object.	Forming Out-of-Bounds Pointer , Null Pointer Arithmetic
47	Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated.	Dereferencing Past the End Pointer , Using Past the End Index
49	An array subscript is out of range, even if an object is apparently accessible with the given subscript, for example, in the lvalue expression a[1] [7] given the declaration int a[4] [5].	Apparently Accessible Out-of-Range Index
62	An attempt is made to access, or generate a pointer to just past, a flexible array member of a structure when the referenced object provides no elements for that array.	Pointer Past Flexible Array Member

Noncompliant Code Example [Forming Out-of-Bounds Pointer]

In this noncompliant code example, the function `f()` attempts to validate the `index` before using it as an offset to the statically allocated `table` of integers. However, the function fails to reject negative `index` values. When `index` is less than zero, the behavior of the addition expression in the return statement of the function is [undefined behavior 46](#). On some implementations, the addition alone can trigger a hardware trap. On other implementations, the addition may produce a result that when dereferenced triggers a hardware trap. Other implementations still may produce a dereferenceable pointer that points to an object distinct from `table`. Using such a pointer to access the object may lead to information exposure or cause the wrong object to be modified.

```
enum { TABLESIZE = 100 };

static int table[TABLESIZE];

int *f(int index) {
```

```

if (index < TABLESIZE) {
    return table + index;
}
return NULL;
}

```

Compliant Solution

One compliant solution is to detect and reject invalid values of `index` if using them in pointer arithmetic would result in an invalid pointer:

```

enum { TABLESIZE = 100 };

static int table[TABLESIZE];

int *f(int index) {
    if (index >= 0 && index < TABLESIZE) {
        return table + index;
    }
    return NULL;
}

```

Compliant Solution

Another slightly simpler and potentially more efficient compliant solution is to use an unsigned type to avoid having to check for negative values while still rejecting out-of-bounds positive values of `index`:

```

#include <stddef.h>

enum { TABLESIZE = 100 };

static int table[TABLESIZE];

int *f(size_t index) {
    if (index < TABLESIZE) {
        return table + index;
    }
    return NULL;
}

```

Noncompliant Code Example (Dereferencing Past-the-End Pointer)

This noncompliant code example shows the flawed logic in the Windows Distributed Component Object Model (DCOM) Remote Procedure Call (RPC) interface that was exploited by the W32.Blaster.Worm. The error is that the `while` loop in the `GetMachineName()` function (used to extract the host name from a longer string) is not sufficiently bounded. When the character array pointed to by `pwszTemp` does not contain the backslash character among the first `MAX_COMPUTERNAME_LENGTH_FQDN + 1` elements, the final valid iteration of the loop will dereference past the end pointer, resulting in exploitable [undefined behavior 47](#). In this case, the actual exploit allowed the attacker to inject executable code into a running program. Economic damage from the Blaster worm has been estimated to be at least \$525 million [[Pethia 2003](#)].

For a discussion of this programming error in the Common Weakness Enumeration database, see [CWE-119](#), "Improper Restriction of Operations within the Bounds of a Memory Buffer," and [CWE-121](#), "Stack-based Buffer Overflow" [[MITRE 2013](#)].

```

error_status_t _RemoteActivation(
    /* ... */, WCHAR *pwszObjectName, ... ) {
    phr = GetServerPath(
        pwszObjectName, &pwszObjectName);
    /* ... */
}

HRESULT GetServerPath(
    WCHAR *pwszPath, WCHAR **pwszServerPath ){
    WCHAR *pwszFinalPath = pwszPath;
    WCHAR wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN+1];
    hr = GetMachineName(pwszPath, wszMachineName);
    *pwszServerPath = pwszFinalPath;
}

HRESULT GetMachineName(
    WCHAR *pwszPath,
    WCHAR wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN+1])
{
    pwszServerName = wszMachineName;
    LPWSTR pwszTemp = pwszPath + 2;
    while (*pwszTemp != L'\\')
        *pwszServerName++ = *pwszTemp++;
    /* ... */
}

```

Compliant Solution

In this compliant solution, the `while` loop in the `GetMachineName()` function is bounded so that the loop terminates when a backslash character is found, the null-termination character [`L'\0'`] is discovered, or the end of the buffer is reached. This code does not result in a buffer overflow even if no backslash character is found in `wszMachineName`.

```

HRESULT GetMachineName(
    wchar_t *pwszPath,
    wchar_t wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN+1])
{
    wchar_t *pwszServerName = wszMachineName;
    wchar_t *pwszTemp = pwszPath + 2;
    wchar_t *end_addr
        = pwszServerName + MAX_COMPUTERNAME_LENGTH_FQDN;
    while ( (*pwszTemp != L'\\') ||
            (*pwszTemp == L'\0') )

```

```

    && ((*pwszTemp != L'\0'))
    && (pwszServerName < end_addr) )
{
    *pwszServerName++ = *pwszTemp++;
}

/* ... */
}

```

This compliant solution is for illustrative purposes and is not necessarily the solution implemented by Microsoft. This particular solution may not be correct because there is no guarantee that a backslash is found.

Noncompliant Code Example [Using Past-the-End Index]

Similar to the [dereferencing-past-the-end-pointer](#) error, the function `insert_in_table()` in this noncompliant code example uses an otherwise valid index to attempt to store a value in an element just past the end of an array.

First, the function incorrectly validates the index `pos` against the size of the buffer. When `pos` is initially equal to `size`, the function attempts to store `value` in a memory location just past the end of the buffer.

Second, when the index is greater than `size`, the function modifies `size` before growing the size of the buffer. If the call to `realloc()` fails to increase the size of the buffer, the next call to the function with a value of `pos` equal to or greater than the original value of `size` will again attempt to store `value` in a memory location just past the end of the buffer or beyond.

Third, the function violates [INT30-C. Ensure that unsigned integer operations do not wrap](#), which could lead to wrapping when 1 is added to `pos` or when `size` is multiplied by the size of `int`.

For a discussion of this programming error in the Common Weakness Enumeration database, see [CWE-122](#), "Heap-based Buffer Overflow," and [CWE-129](#), "Improper Validation of Array Index" [MITRE 2013].

```

#include <stdlib.h>

static int *table = NULL;
static size_t size = 0;

int insert_in_table(size_t pos, int value) {
    if (size < pos) {
        int *tmp;
        size = pos + 1;
        tmp = (int *)realloc(table, sizeof(*table) * size);
        if (tmp == NULL) {
            return -1; /* Failure */
        }
        table = tmp;
    }

    table[pos] = value;
    return 0;
}

```

Compliant Solution

This compliant solution correctly validates the index `pos` by using the `<=` relational operator, ensures the multiplication will not overflow, and avoids modifying `size` until it has verified that the call to `realloc()` was successful:

```

#include <stdint.h>
#include <stdlib.h>

static int *table = NULL;
static size_t size = 0;

int insert_in_table(size_t pos, int value) {
    if (size <= pos) {
        if ((SIZE_MAX - 1 < pos) ||
            (pos + 1) > SIZE_MAX / sizeof(*table))) {
            return -1;
        }

        int *tmp = (int *)realloc(table, sizeof(*table) * (pos + 1));
        if (tmp == NULL) {
            return -1;
        }
        /* Modify size only after realloc() succeeds */
        size = pos + 1;
        table = tmp;
    }

    table[pos] = value;
    return 0;
}

```

Noncompliant Code Example [Apparently Accessible Out-of-Range Index]

This noncompliant code example declares `matrix` to consist of 7 rows and 5 columns in row-major order. The function `init_matrix` iterates over all 35 elements in an attempt to initialize each to the value given by the function argument `x`. However, because multidimensional arrays are declared in C in row-major order, the function iterates over the elements in column-major order, and when the value of `j` reaches the value `cols` during the first iteration of the outer loop, the function attempts to access element `matrix[0][5]`. Because the type of `matrix` is `int[7][5]`, the `j` subscript is out of range, and the access has [undefined behavior 49](#).

```

#include <stddef.h>
#define COLS 5
#define ROWS 7

```

```

static int matrix[ROWS][COLS];

void init_matrix(int x) {
    for (size_t i = 0; i < COLS; i++) {
        for (size_t j = 0; j < ROWS; j++) {
            matrix[i][j] = x;
        }
    }
}

```

Compliant Solution

This compliant solution avoids using out-of-range indices by initializing `matrix` elements in the same row-major order as multidimensional objects are declared in C:

```

#include <stddef.h>
#define COLS 5
#define ROWS 7
static int matrix[ROWS][COLS];

void init_matrix(int x) {
    for (size_t i = 0; i < ROWS; i++) {
        for (size_t j = 0; j < COLS; j++) {
            matrix[i][j] = x;
        }
    }
}

```

Noncompliant Code Example [Pointer Past Flexible Array Member]

In this noncompliant code example, the function `find()` attempts to iterate over the elements of the flexible array member `buf`, starting with the second element. However, because function `g()` does not allocate any storage for the member, the expression `first++` in `find()` attempts to form a pointer just past the end of `buf` when there are no elements. This attempt is [undefined behavior 62](#). (See [MSC21-C. Use robust loop termination conditions](#) for more information.)

```

#include <stdlib.h>

struct S {
    size_t len;
    char buf[]; /* Flexible array member */
};

const char *find(const struct S *s, int c) {
    const char *first = s->buf;
    const char *last = s->buf + s->len;

    while (first++ != last) { /* Undefined behavior */
        if (*first == (unsigned char)c) {
            return first;
        }
    }
    return NULL;
}

void g(void) {
    struct S *s = (struct S *)malloc(sizeof(struct S));
    if (s == NULL) {
        /* Handle error */
    }
    s->len = 0;
    find(s, 'a');
}

```

Compliant Solution

This compliant solution avoids incrementing the pointer unless a value past the pointer's current value is known to exist:

```

#include <stdlib.h>

struct S {
    size_t len;
    char buf[]; /* Flexible array member */
};

const char *find(const struct S *s, int c) {
    const char *first = s->buf;
    const char *last = s->buf + s->len;

    while (first != last) { /* Avoid incrementing here */
        if (*++first == (unsigned char)c) {
            return first;
        }
    }
    return NULL;
}

void g(void) {
    struct S *s = (struct S *)malloc(sizeof(struct S));
    if (s == NULL) {
        /* Handle error */
    }
    s->len = 0;
    find(s, 'a');
}

```

Noncompliant Code Example (Null Pointer Arithmetic)

This noncompliant code example is similar to an [Adobe Flash Player vulnerability](#) that was first exploited in 2008. This code allocates a block of memory and initializes it with some data. The data does not belong at the beginning of the block, which is left uninitialized. Instead, it is placed `offset` bytes within the block. The function ensures that the data fits within the allocated block.

```
#include <string.h>
#include <stdlib.h>

char *init_block(size_t block_size, size_t offset,
                 char *data, size_t data_size) {
    char *buffer = malloc(block_size);
    if (data_size > block_size || block_size - data_size < offset) {
        /* Data won't fit in buffer, handle error */
    }
    memcpy(buffer + offset, data, data_size);
    return buffer;
}
```

This function fails to check if the allocation succeeds, which is a violation of [ERR33-C. Detect and handle standard library errors](#). If the allocation fails, then `malloc()` returns a null pointer. The null pointer is added to `offset` and passed as the destination argument to `memcpy()`. Because a null pointer does not point to a valid object, the result of the pointer arithmetic is [undefined behavior 46](#).

An attacker who can supply the arguments to this function can exploit it to execute arbitrary code. This can be accomplished by providing an overly large value for `block_size`, which causes `malloc()` to fail and return a null pointer. The `offset` argument will then serve as the destination address to the call to `memcpy()`. The attacker can specify the `data` and `data_size` arguments to provide the address and length of the address, respectively, that the attacker wishes to write into the memory referenced by `offset`. The overall result is that the call to `memcpy()` can be exploited by an attacker to overwrite an arbitrary memory location with an attacker-supplied address, typically resulting in arbitrary code execution.

Compliant Solution (Null Pointer Arithmetic)

This compliant solution ensures that the call to `malloc()` succeeds:

```
#include <string.h>
#include <stdlib.h>

char *init_block(size_t block_size, size_t offset,
                 char *data, size_t data_size) {
    char *buffer = malloc(block_size);
    if (NULL == buffer) {
        /* Handle error */
    }
    if (data_size > block_size || block_size - data_size < offset) {
        /* Data won't fit in buffer, handle error */
    }
    memcpy(buffer + offset, data, data_size);
    return buffer;
}
```

Risk Assessment

Writing to out-of-range pointers or array subscripts can result in a buffer overflow and the execution of arbitrary code with the permissions of the vulnerable process. Reading from out-of-range pointers or array subscripts can result in unintended information disclosure.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ARR30-C	High	Likely	High	P9	L2

Related Guidelines

ISO/IEC TR 24772:2013	Arithmetic Wrap-Around Error [FIF] Unchecked Array Indexing [XYZ]
ISO/IEC TS 17961	Forming or using out-of-bounds pointers or array subscripts [invptr]
MITRE CWE	CWE-119 , Improper Restriction of Operations within the Bounds of a Memory Buffer CWE-122 , Heap-based Buffer Overflow CWE-123 , Write-what-where Condition CWE-125 , Out-of-bounds Read CWE-129 , Improper Validation of Array Index CWE-788 , Access of Memory Location after End of Buffer
MISRA C:2012	Rule 18.1 (required)

Bibliography

[Finlay 2003]	
[Microsoft 2003]	
[Pethia 2003]	
[Seacord 2013b]	Chapter 1, "Running with Scissors"
[Viega 2005]	Section 5.2.13, "Unchecked Array Indexing"
[xorl 2009]	"CVE-2008-1517: Apple Mac OS X [XNU] Missing Array Index Validation"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/DYDXAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	9
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
arithmetic_out_of_bounds	Pointer arithmetic on {1} creates pointer outside array bounds of {0}
out_of_bounds	Access into array is out of bounds
possible_indirect_out_of_bounds	Pointer-indirect access through {1} might be out of bounds accessing {0}
possible_out_of_bounds	Access into array might be out of bounds

CertC-ARR36

Do not subtract or compare two pointers that do not refer to the same array.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

When two pointers are subtracted, both must point to elements of the same array object or just one past the last element of the array object (C Standard, 6.5.6 [[ISO/IEC 9899:2011](#)]); the result is the difference of the subscripts of the two array elements. Otherwise, the operation is [undefined behavior](#). (See [undefined behavior 48](#).)

Similarly, comparing pointers using the relational operators <, <=, >=, and > gives the positions of the pointers relative to each other. Subtracting or comparing pointers that do not refer to the same array is undefined behavior. (See [undefined behavior 48](#) and [undefined behavior 53](#).)

Comparing pointers using the equality operators == and != has well-defined semantics regardless of whether or not either of the pointers is null, points into the same object, or points one past the last element of an array object or function.

Noncompliant Code Example

In this noncompliant code example, pointer subtraction is used to determine how many free elements are left in the `nums` array:

```
#include <stddef.h>
enum { SIZE = 32 };

void func(void) {
    int nums[SIZE];
    int end;
    int *next_num_ptr = nums;
    size_t free_elements;

    /* Increment next_num_ptr as array fills */
    free_elements = &end - next_num_ptr;
```

This program incorrectly assumes that the `nums` array is adjacent to the `end` variable in memory. A compiler is permitted to insert padding bits between these two variables or even reorder them in memory.

Compliant Solution

In this compliant solution, the number of free elements is computed by subtracting `next_num_ptr` from the address of the pointer past the `nums` array. While this pointer may not be dereferenced, it may be used in pointer arithmetic.

```
#include <stddef.h>
enum { SIZE = 32 };

void func(void) {
    int nums[SIZE];
    int *next_num_ptr = nums;
    size_t free_elements;

    /* Increment next_num_ptr as array fills */
    free_elements = &(nums[SIZE]) - next_num_ptr;
}
```

Exceptions

ARR36-C-EX1: Comparing two pointers to distinct members of the same `struct` object is allowed. Pointers to structure members declared later in the structure compare greater-than pointers to members declared earlier in the structure.

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ARR36-C	Medium	Probable	Medium	P8	L2

Related Guidelines

SEI CERT C++ Coding Standard	CTR54-CPP. Do not subtract iterators that do not refer to the same container
ISO/IEC TS 17961	Subtracting or comparing two pointers that do not refer to the same array [ptrob]
MITRE CWE	CWE-469 , Use of Pointer Subtraction to Determine Size

Bibliography

[Banahan 2003]	Section 5.3, "Pointers" Section 5.7, "Expressions Involving Pointers"
[ISO/IEC 9899:2011]	6.5.6, "Additive Operators"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/LIDp>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
possible_unrelated_ptr_comparison	Comparing possibly unrelated pointers
possible_unrelated_ptr_subtraction	Subtracting possibly unrelated pointers
unrelated_ptr_comparison	Comparing unrelated pointers
unrelated_ptr_subtraction	Subtracting unrelated pointers

CertC-ARR37

Do not add or subtract an integer to a pointer to a non-array object.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Pointer arithmetic must be performed only on pointers that reference elements of array objects.

The C Standard, 6.5.6 [ISO/IEC 9899:2011], states the following about pointer arithmetic:

When an expression that has integer 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 integer expression.

Noncompliant Code Example

This noncompliant code example attempts to access structure members using pointer arithmetic. This practice is dangerous because structure members are not guaranteed to be contiguous.

```
struct numbers {
    short num_a, num_b, num_c;
};

int sum_numbers(const struct numbers *numb) {
    int total = 0;
    const short *numb_ptr;

    for (numb_ptr = &numb->num_a;
        numb_ptr <= &numb->num_c;
        numb_ptr++) {
        total += *(numb_ptr);
    }

    return total;
}

int main(void) {
    struct numbers my_numbers = { 1, 2, 3 };
    sum_numbers(&my_numbers);
    return 0;
}
```

Compliant Solution

It is possible to use the `->` operator to dereference each structure member:

```
total = numb->num_a + numb->num_b + numb->num_c;
```

However, this solution results in code that is hard to write and hard to maintain (especially if there are many more structure members), which is exactly what the author of the noncompliant code example was likely trying to avoid.

Compliant Solution

A better solution is to define the structure to contain an array member to store the numbers in an array rather than a structure, as in this compliant solution:

```
#include <stddef.h>

struct numbers {
    short a[3];
};

int sum_numbers(const short *numb, size_t dim) {
    int total = 0;
    for (size_t i = 0; i < dim; ++i) {
        total += numb[i];
    }

    return total;
}

int main(void) {
    struct numbers my_numbers = { .a[0]= 1, .a[1]= 2, .a[2]= 3 };
    sum_numbers(&my_numbers,
```

```

    my_numbers.a,
    sizeof(my_numbers.a)/sizeof(my_numbers.a[0])
);
return 0;
}

```

Array elements are guaranteed to be contiguous in memory, so this solution is completely portable.

Exceptions

ARR37-C-EX1: Any non-array object in memory can be considered an array consisting of one element. Adding one to a pointer for such an object yields a pointer one element past the array, and subtracting one from that pointer yields the original pointer. This allows for code such as the following:

```

#include <stdlib.h>
#include <string.h>

struct s {
    char *c_str;
    /* Other members */
};

struct s *create_s(const char *c_str) {
    struct s *ret;
    size_t len = strlen(c_str) + 1;

    ret = (struct s *)malloc(sizeof(struct s) + len);
    if (ret != NULL) {
        ret->c_str = (char *)(ret + 1);
        memcpy(ret + 1, c_str, len);
    }
    return ret;
}

```

A more general and safer solution to this problem is to use a flexible array member that guarantees the array that follows the structure is properly aligned by inserting padding, if necessary, between it and the member that immediately precedes it.

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ARR37-C	Medium	Probable	Medium	P8	L2

Related Guidelines

MITRE CWE	CWE-469 , Use of Pointer Subtraction to Determine Size
---------------------------	--

Bibliography

[Banahan 2003]	Section 5.3, "Pointers" Section 5.7, "Expressions Involving Pointers"
[ISO/IEC 9899:2011]	6.5.6, "Additive Operators"
[VU#162289]	

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/UgHm>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
arithmetic_on_nonarray	Pointer arithmetic on {1} operates on non-array target {0}

CertC-ARR39

Do not add or subtract a scaled integer to a pointer.

Input: IR

Source languages: C, C++

Details

Pointer arithmetic is appropriate only when the pointer argument refers to an array (see [ARR37-C. Do not add or subtract an integer to a pointer to a non-array object](#)), including an array of bytes. When performing pointer arithmetic, the size of the value to add to or subtract from a pointer is automatically scaled to the size of the type of the referenced array object. Adding or subtracting a scaled integer value to or from a pointer is invalid because it may yield a pointer that does not point to an element within or one past the end of the array. (See [ARR30-C. Do not form or use out-of-bounds pointers or array subscripts](#).)

Adding a pointer to an array of a type other than character to the result of the `sizeof` operator or `offsetof` macro, which returns a size and an offset, respectively, violates this rule. However, adding an array pointer to the number of array elements, for example, by using the `arr[sizeof(arr)/sizeof(arr[0])]` idiom, is allowed provided that `arr` refers to an array and not a pointer.

Noncompliant Code Example

In this noncompliant code example, `sizeof(buf)` is added to the array `buf`. This example is noncompliant because `sizeof(buf)` is scaled by `int` and then scaled again when added to `buf`.

```
enum { INTBUFSIZE = 80 };

extern int getdata(void);
int buf[INTBUFSIZE];

void func(void) {
    int *buf_ptr = buf;

    while (buf_ptr < (buf + sizeof(buf))) {
        *buf_ptr++ = getdata();
    }
}
```

Compliant Solution

This compliant solution uses an unscaled integer to obtain a pointer to the end of the array:

```
enum { INTBUFSIZE = 80 };

extern int getdata(void);
int buf[INTBUFSIZE];

void func(void) {
    int *buf_ptr = buf;

    while (buf_ptr < (buf + INTBUFSIZE)) {
        *buf_ptr++ = getdata();
    }
}
```

Noncompliant Code Example

In this noncompliant code example, `skip` is added to the pointer `s`. However, `skip` represents the byte offset of `ull_b` in `struct big`. When added to `s`, `skip` is scaled by the size of `struct big`.

```
#include <string.h>
#include <stdlib.h>
#include <stddef.h>

struct big {
    unsigned long long ull_a;
    unsigned long long ull_b;
    unsigned long long ull_c;
    int si_e;
    int si_f;
};

void func(void) {
    size_t skip = offsetof(struct big, ull_b);
    struct big *s = (struct big *)malloc(sizeof(struct big));
    if (s == NULL) {
        /* Handle malloc() error */
    }

    memset(s + skip, 0, sizeof(struct big) - skip);
    /* ... */
    free(s);
    s = NULL;
}
```

Compliant Solution

This compliant solution uses an `unsigned char *` to calculate the offset instead of using a `struct big *`, which would result in scaled arithmetic:

```

#include <string.h>
#include <stdlib.h>
#include <stddef.h>

struct big {
    unsigned long long ull_a;
    unsigned long long ull_b;
    unsigned long long ull_c;
    int si_d;
    int si_e;
};

void func(void) {
    size_t skip = offsetof(struct big, ull_b);
    unsigned char *ptr = (unsigned char *)malloc(
        sizeof(struct big)
    );
    if (ptr == NULL) {
        /* Handle malloc() error */
    }

    memset(ptr + skip, 0, sizeof(struct big) - skip);
    /* ... */
    free(ptr);
    ptr = NULL;
}

```

Noncompliant Code Example

In this noncompliant code example, `wcslen(error_msg) * sizeof(wchar_t)` bytes are scaled by the size of `wchar_t` when added to `error_msg`:

```

#include <wchar.h>
#include <stdio.h>

enum { WCHAR_BUF = 128 };

void func(void) {
    wchar_t error_msg[WCHAR_BUF];

    wcscpy(error_msg, L"Error: ");
    fgetws(error_msg + wcslen(error_msg) * sizeof(wchar_t),
           WCHAR_BUF - 7, stdin);
    /* ... */
}

```

Compliant Solution

This compliant solution does not scale the length of the string; `wcslen()` returns the number of characters and the addition to `error_msg` is scaled:

```

#include <wchar.h>
#include <stdio.h>

enum { WCHAR_BUF = 128 };
const wchar_t ERROR_PREFIX[7] = L"Error: ";

void func(void) {
    const size_t prefix_len = wcslen(ERROR_PREFIX);
    wchar_t error_msg[WCHAR_BUF];

    wcscpy(error_msg, ERROR_PREFIX);
    fgetws(error_msg + prefix_len,
           WCHAR_BUF - prefix_len, stdin);
    /* ... */
}

```

Risk Assessment

Failure to understand and properly use pointer arithmetic can allow an attacker to execute arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ARR39-C	High	Probable	High	P6	L2

Related Guidelines

CERT C Secure Coding Standard	ARR30-C. Do not form or use out-of-bounds pointers or array subscripts ARR37-C. Do not add or subtract an integer to a pointer to a non-array object
ISO/IEC TR 24772:2013	Pointer Casting and Pointer Type Changes [HFC] Pointer Arithmetic [RVG]
MISRA C:2012	Rule 18.1 (required) Rule 18.2 (required) Rule 18.3 (required) Rule 18.4 (advisory)
MITRE CWE	CWE-468 , Incorrect Pointer Scaling

Bibliography

[Dowd 2006]	Chapter 6, "C Language Issues"
[Murenin 07]	

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/HADXAQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
scaled_arith	Do not add or subtract a scaled integer to a pointer.

CertC-STR04

Use plain `char` for characters in the basic character set.

Input: IR

Source languages: C, C++

Details

There are three *character types*: `char`, `signed char`, and `unsigned char`. Compilers have the latitude to define `char` to have the same range, representation, and behavior as either `signed char` or `unsigned char`. Irrespective of the choice made, `char` is a separate type from the other two and is not compatible with either.

For characters in the basic character set, it does not matter which data type is used, except for type compatibility. Consequently, it is best to use plain `char` for character data for compatibility with standard string-handling functions.

In most cases, the only portable operators on plain `char` types are assignment and equality operators (`=`, `==`, `!=`). An exception is the translation to and from digits. For example, if the `char c` is a digit, `c - '\0'` is a value between 0 and 9.

Noncompliant Code Example

This noncompliant code example simply shows the standard string-handling function `strlen()` being called with a plain character string, a signed character string, and an unsigned character string. The `strlen()` function takes a single argument of type `const char *`:

```
size_t len;
char cstr[] = "char string";
signed char scstr[] = "signed char string";
unsigned char ucstr[] = "unsigned char string";

len = strlen(cstr);
len = strlen(scstr); /* Warns when char is unsigned */
len = strlen(ucstr); /* Warns when char is signed */
```

Compiling at high warning levels in compliance with [MSC00-C. Compile cleanly at high warning levels](#) causes warnings to be issued when

- Converting from `unsigned char[]` to `const char *` when `char` is signed
- Converting from `signed char[]` to `const char *` when `char` is defined to be unsigned

Casts are required to eliminate these warnings, but excessive casts can make code difficult to read and hide legitimate warning messages.

If this C code were compiled using a C++ compiler, conversions from `unsigned char[]` to `const char *` and from `signed char[]` to `const char *` would be flagged as errors requiring casts.

Compliant Solution

The compliant solution uses plain `char` for character data:

```
size_t len;
char cstr[] = "char string";
len = strlen(cstr);
```

Conversions are not required, and the code compiles cleanly at high warning levels without casts.

Risk Assessment

Failing to use plain `char` for characters in the basic character set can lead to excessive casts and less effective compiler diagnostics.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
STR04-C	Low	Unlikely	Low	P3	L3

Related Guidelines

SEI CERT C++ Coding Standard	STR04-CPP. Use plain char for characters in the basic character set
MISRA C:2012	Rule 10.1 (required) Rule 10.2 (required) Rule 10.3 (required) Rule 10.4 (required)

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/JABi>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
numeric_char_used_as_character	Signed/unsigned char datatype used for character data.

CertC-STR05

Use pointers to `const` when referring to string literals.

Input: IR

Source languages: C, C++

Details

The type of a narrow string literal is an array of `char`, and the type of a wide string literal is an array of `wchar_t`. However, string literals (of both types) are notionally constant and should consequently be protected by `const` qualification. This recommendation is a specialization of [DCL00-C. Const-qualify immutable objects](#) and also supports [STR30-C. Do not attempt to modify string literals](#).

Adding `const` qualification may propagate through a program; as `const` qualifiers are added, still more become necessary. This phenomenon is sometimes called *const-poisoning*. Const-poisoning can frequently lead to violations of [EXP05-C. Do not cast away a const qualification](#). Although `const` qualification is a good idea, the costs may outweigh the value in the remediation of existing code.

Noncompliant Code Example (Narrow String Literal)

In this noncompliant code example, the `const` keyword has been omitted:

```
char *c = "Hello";
```

If a statement such as `c[0] = '\c'` were placed following the declaration in the noncompliant code example, the code is likely to compile cleanly, but the result of the assignment would be [undefined](#) because string literals are considered constant.

Compliant Solution (Immutable Strings)

In this compliant solution, the characters referred to by the pointer `c` are `const`-qualified, meaning that any attempt to assign them to different values is an error:

```
const char *c = "Hello";
```

Compliant Solution (Mutable Strings)

In cases where the string is meant to be modified, use initialization instead of assignment. In this compliant solution, `c` is a modifiable `char` array that has been initialized using the contents of the corresponding string literal:

```
char c[] = "Hello";
```

Consequently, a statement such as `c[0] = '\c'` is valid and behaves as expected.

Noncompliant Code Example (Wide String Literal)

In this noncompliant code example, the `const` keyword has been omitted:

```
wchar_t *c = L"Hello";
```

If a statement such as `c[0] = L'\c'` were placed following this declaration, the code is likely to compile cleanly, but the result of the assignment would be [undefined](#) because string literals are considered constant.

Compliant Solution (Immutable Strings)

In this compliant solution, the characters referred to by the pointer `c` are `const`-qualified, meaning that any attempt to assign them to different values is an error:

```
wchar_t const *c = L"Hello";
```

Compliant Solution (Mutable Strings)

In cases where the string is meant to be modified, use initialization instead of assignment. In this compliant solution, `c` is a modifiable `wchar_t` array that has been initialized using the contents of the corresponding string literal:

```
wchar_t c[] = L"Hello";
```

Consequently, a statement such as `c[0] = L'\c'` is valid and behaves as expected.

Risk Assessment

Modifying string literals causes [undefined behavior](#), resulting in [abnormal program termination](#) and [denial-of-service vulnerabilities](#).

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
STR05-C	Low	Unlikely	Low	P3	L3

Bibliography

[Corfield 1993]	
[Lockheed Martin 2005]	AV Rule 151.1

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/mwAV>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
nonconst_string_literal	String literal should only be used as 'const char*'

CertC-STR07

Use the bounds-checking interfaces for string manipulation.

Input: IR

Source languages: C, C++

Details

The C Standard, Annex K (normative), defines alternative versions of standard string-handling functions designed to be safer replacements for existing functions. For example, it defines the `strncpy_s()`, `strcat_s()`, `strncat_s()`, and `strncpy_s()` functions as replacements for `strcpy()`, `strcat()`, `strncpy()`, and `strncat()`, respectively.

The Annex K functions were created by Microsoft to help retrofit its existing legacy code base in response to numerous, well-publicized security incidents over the past decade. These functions were subsequently proposed to the international standardization working group for the programming language C (ISO/IEC JTC1/SC22/WG14) for standardization.

The `strcpy_s()` function, for example, has this signature:

```
errno_t strcpy_s(
    char * restrict s1,
    rsize_t s1max,
    const char * restrict s2
);
```

The signature is similar to `strcpy()` but takes an extra argument of type `rsize_t` that specifies the maximum length of the destination buffer. Functions that accept parameters of type `rsize_t` diagnose a constraint violation if the values of those parameters are greater than `RSIZE_MAX`. Extremely large object sizes are frequently a sign that an object's size was calculated incorrectly. For example, negative numbers appear as very large positive numbers when converted to an unsigned type like `size_t`. For those reasons, it is sometimes beneficial to restrict the range of object sizes to detect errors. For machines with large address spaces, the C Standard, Annex K, recommends that `RSIZE_MAX` be defined as the smaller of the size of the largest object supported or `(SIZE_MAX >> 1)`, even if this limit is smaller than the size of some legitimate, but very large, objects (see also [INT01-C. Use rsize_t or size_t for all integer values representing the size of an object](#)).

The semantics of `strcpy_s()` are similar to the semantics of `strcpy()`. When there are no input validation errors, the `strcpy_s()` function copies characters from a source string to a destination character array up to and including the terminating null character. The function returns 0 on success.

The `strcpy_s()` function succeeds only when the source string can be fully copied to the destination without overflowing the destination buffer. Specifically, the following checks are made:

- The source and destination pointers are checked to see if they are `NULL`.
- The maximum length of the destination buffer is checked to see if it is equal to 0, greater than `RSIZE_MAX`, or less than or equal to the length of the source string.
- Copying is not allowed between objects that overlap.

When a runtime-constraint violation is detected, the destination string is set to the null string (as long as it is not a null pointer, and the maximum length of the destination buffer is greater than 0 and not greater than `RSIZE_MAX`), and the function returns a nonzero value. In the following example, the `strcpy_s()` function is used to copy `src1` to `dst1`:

```
char src1[100] = "hello";
char src2[8] = {'g','o','o','d','b','y','e','\0'};
char dst1[6];
char dst2[5];
int r1;
int r2;

r1 = strcpy_s(dst1, sizeof(dst1), src1);
r2 = strcpy_s(dst2, sizeof(dst2), src2);
```

However, the call to copy `src2` to `dst2` fails because insufficient space is available to copy the entire string, which consists of eight characters, to the destination buffer. As a result, `r2` is assigned a nonzero value and `dst2[0]` is set to the null character.

Users of the C Standard Annex K functions are less likely to introduce a [security flaw](#) because the size of the destination buffer and the maximum number of characters to append must be specified. ISO/IEC TR 24731 Part II [[ISO/IEC TR 24731-2:2010](#)] offers another approach, supplying functions that allocate

enough memory for their results. ISO/IEC TR 24731 Part II functions also ensure null termination of the destination string.

The C Standard Annex K functions are still capable of overflowing a buffer if the maximum length of the destination buffer and number of characters to copy are incorrectly specified. ISO/IEC TR 24731 Part II functions can make it more difficult to keep track of memory that must be freed, leading to memory leaks. As a result, the C Standard Annex K and the ISO/IEC TR 24731 Part II functions are not particularly secure but may be useful in preventive maintenance to reduce the likelihood of vulnerabilities in an existing legacy code base.

Noncompliant Code Example

This noncompliant code overflows its buffer if `msg` is too long, and it has [undefined behavior](#) if `msg` is a null pointer:

```
void complain(const char *msg) {
    static const char prefix[] = "Error: ";
    static const char suffix[] = "\n";
    char buf[BUFSIZ];

    strcpy(buf, prefix);
    strcat(buf, msg);
    strcat(buf, suffix);
    fputs(buf, stderr);
}
```

Compliant Solution (Runtime)

This compliant solution will not overflow its buffer:

```
void complain(const char *msg) {
    errno_t err;
    static const char prefix[] = "Error: ";
    static const char suffix[] = "\n";
    char buf[BUFSIZ];

    err = strcpy_s(buf, sizeof(buf), prefix);
    if (err != 0) {
        /* Handle error */
    }

    err = strcat_s(buf, sizeof(buf), msg);
    if (err != 0) {
        /* Handle error */
    }

    err = strcat_s(buf, sizeof(buf), suffix);
    if (err != 0) {
        /* Handle error */
    }

    fputs(buf, stderr);
}
```

Compliant Solution (Partial Compile Time)

This compliant solution performs some of the checking at compile time using a static assertion (see [DCL03-C. Use a static assertion to test the value of a constant expression](#)).

```
void complain(const char *msg) {
    errno_t err;
    static const char prefix[] = "Error: ";
    static const char suffix[] = "\n";
    char buf[BUFSIZ];

    /*
     * Ensure that more than one character
     * is available for msg
     */
    static_assert(sizeof(buf) > sizeof(prefix) + sizeof(suffix),
                 "Buffer for complain() is too small");
    strcpy(buf, prefix);

    err = strcat_s(buf, sizeof(buf), msg);
    if (err != 0) {
        /* Handle error */
    }

    err = strcat_s(buf, sizeof(buf), suffix);
    if (err != 0) {
        /* Handle error */
    }
    fputs(buf, stderr);
}
```

Risk Assessment

String-handling functions defined in the C Standard, subclause 7.24, and elsewhere are susceptible to common programming errors that can lead to serious, [exploitable vulnerabilities](#). Proper use of the C11 Annex K functions can eliminate most of these issues.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
STR07-C	High	Probable	Medium	P12	L1

Related Guidelines

ISO/IEC TR 24731-2:2010	
ISO/IEC TR 24772:2013	Use of Libraries [TRJ]

Bibliography

[Seacord 2005b]	"Managed String Library for C, C/C++"
[Seacord 2013]	Chapter 2, "Strings"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/QwY>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
blacklist	Dictionary of header globbing to (list of) function name globbing(s) of forbidden functions. These functions should be replaced by their _s counterpart (strlen and strnlen both map to strnlen_s)	dict{...}
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
forbidden_libfunc_call	Use the bounds-checking interface instead of this function.

CertC-STR09

Don't assume numeric values for expressions with type plan character.

Input: IR

Source languages: C, C++

Details

For portable applications, use only the assignment = operator, the equality operators == and !=, and the unary & operator on plain-character-typed or plain-wide-character-typed expressions.

This practice is recommended because the C Standard requires only the digit characters (0-9) to have consecutive numerical values. Consequently, operations that rely on expected values for plain-character- or plain-wide-character-typed expressions can lead to unexpected behavior.

However, because of the requirement for digit characters, other operators can be used for them according to the following restrictions:

- The binary + operator may be used to add integer values 0 through 9 to '\0'.
- The binary - operator may be used to subtract character 0.
- Relational operators <, <=, >, and >= can be used to check whether a character or wide character is a digit.

Character types should be chosen and used in accordance with [STR04-C. Use plain char for characters in the basic character set](#).

Noncompliant Code Example

This noncompliant code example attempts to determine if the value of a character variable is between '\a\' and '\c\' inclusive. However, because the C Standard does not require the letter characters to be in consecutive or alphabetic order, the check might not work as expected.

```
char ch = '\b\';
if ((ch >= '\a\') && (ch <= '\c\')) {
/* ... */
```

Compliant Solution

In this example, the specific check is enforced using compliant operations on character expressions:

```
char ch = '\t\';
if ((ch == '\a\') || (ch == '\b\') || (ch == '\c\')) {
/* ... */
```

Exceptions

STR09-C-EX1: Consecutive values for characters like `a~z` can be assumed on platforms where ASCII or Unicode is used. This recommendation is primarily concerned with platform portability, for example, if code is migrated from ASCII systems to non-ASCII systems.

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR09-C	Low	Unlikely	Low	P3	L3

Related Guidelines

[SEI CERT C++ Coding Standard](#) | [STR07-CPP. Don't assume numeric values for expressions with type plain character](#)

Bibliography

[[Jones 2009](#)] Section 5.2.1, "Character Sets"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/b4AzAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
char_operand_outside_comparison	Use of character operand in forbidden context

CertC-STR10

Do not concatenate different type of string literals.

Input: IR

Source languages: C, C++

Details

According to [MISRA 2008](#), concatenation of wide and narrow string literals leads to [undefined behavior](#). This was once considered implicitly undefined behavior until C90 [[ISO/IEC 9899:1990](#)]. However, C99 defined this behavior [[ISO/IEC 9899:1999](#)], and C11 further explains in subclause 6.4.5, paragraph 5 [[ISO/IEC 9899:2011](#)]:

In translation phase 6, the multibyte character sequences specified by any sequence of adjacent character and identically-prefixed string literal tokens are concatenated into a single multibyte character sequence. If any of the tokens has an encoding prefix, the resulting multibyte character sequence is treated as having the same prefix; otherwise, it is treated as a character string literal. Whether differently-prefixed wide string literal tokens can be concatenated and, if so, the treatment of the resulting multibyte character sequence are implementation-defined.

Nonetheless, it is recommended that string literals that are concatenated should all be the same type so as not to rely on [implementation-defined behavior](#) or [undefined behavior](#) if compiled on a platform that supports only C90.

Noncompliant Code Example (C90)

This noncompliant code example concatenates wide and narrow string literals. Although the behavior is undefined in C90, the programmer probably intended to create a wide string literal.

```
wchar_t *msg = L"This message is very long, so I want to divide it "
                 "into two parts.";
```

Compliant Solution (C90, Wide String Literals)

If the concatenated string needs to be a wide string literal, each element in the concatenation must be a wide string literal, as in this compliant solution:

```
wchar_t *msg = L"This message is very long, so I want to divide it "
L"into two parts.";
```

Compliant Solution [C90, Narrow String Literals]

If wide string literals are unnecessary, it is better to use narrow string literals, as in this compliant solution:

```
char *msg = "This message is very long, so I want to divide it "
"into two parts.;"
```

Risk Assessment

The concatenation of wide and narrow string literals could lead to undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR10-C	Low	Probable	Medium	P4	L3

Related Guidelines

MISRA C++:2008	Rule 2-13-5
--------------------------------	-------------

Bibliography

[ISO/IEC 9899:2011]	Section 6.4.5, "String Literals"
-------------------------------------	----------------------------------

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/QIEzAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
cpp11_mode	Use rules as defined in the C++11 standard.	False
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
mixed_string_concatenation	Concatenation of mixed string encodings
narrow_wide_concat	Concatenation of narrow and wide string literal

CertC-STR11

Do not specify the bound of a character array initialized with a string literal.

Input: IR

Source languages: C, C++

Details

The C Standard allows an array variable to be declared both with a bound index and with an initialization literal. The initialization literal also implies an array size in the number of elements specified. For strings, the size specified by a string literal is the number of characters in the literal plus one for the terminating null character.

It is common for an array variable to be initialized by a string literal and declared with an explicit bound that matches the number of characters in the string literal. Subclause 6.7.9, paragraph 14, of the C Standard [[ISO/IEC 9899:2011](#)], says:

An array of character type may be initialized by a character string literal or UTF-8 string literal, optionally enclosed in braces. Successive bytes of the string literal (including the terminating null character if there is room or if the array is of unknown size) initialize the elements of the array.

However, if the string is intended to be used as a null-terminated byte string, then the array will have one too few characters to hold the string because it does not account for the terminating null character. Such a sequence of characters has limited utility and has the potential to cause [vulnerabilities](#) if a null-

terminated byte string is assumed.

A better approach is to not specify the bound of a string initialized with a string literal because the compiler will automatically allocate sufficient space for the entire string literal, including the terminating null character. This rule is a specific exception to [ARR02-C. Explicitly specify array bounds, even if implicitly defined by an initializer](#).

Noncompliant Code Example

This noncompliant code example initializes an array of characters using a string literal that defines one character more (counting the terminating '\0') than the array can hold:

```
const char s[3] = "abc";
```

The size of the array `s` is 3, although the size of the string literal is 4. Any subsequent use of the array as a null-terminated byte string can result in a vulnerability, because `s` is not properly null-terminated. (See [STR32-C. Do not pass a non-null-terminated character sequence to a library function that expects a string](#).)

Implementation Details

This code compiles with no warning with Visual Studio 2013 and GCC 4.8.1. It produces a three-character array with no terminating null character, as specified by the standard.

Compliant Solution

This compliant solution does not specify the bound of a character array in the array declaration. If the array bound is omitted, the compiler allocates sufficient storage to store the entire string literal, including the terminating null character.

```
const char s[] = "abc";
```

This approach is preferred because the size of the array can always be derived even if the size of the string literal changes.

Exceptions

STR11-C-EX1: If the intention is to create a character array and *not* a null-terminated byte string, initializing to fit exactly without a null byte is allowed but not recommended. The preferred approach to create an array containing just the three characters 'a', 'b', and 'c', for example, is to declare each character literal as a separate element as follows:

```
char s[3] = { 'a', 'b', 'c' }; /* NOT a string */
```

Also, you should make clear in comments or documentation if a character array is, in fact, not a null-terminated byte string.

STR11-C-EX2: If the character array must be larger than the string literal it is initialized with, you may explicitly specify an array bounds. This is particularly important if the array's contents might change during program execution.

```
#include <string.h>
void func(void) {
    char s[10] = "abc";
    strcpy(&s[3], "def");
}
```

Risk Assessment

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
STR11-C	Low	Probable	Low	P6	L2

Related Guidelines

CERT C Secure Coding Standard	ARR02-C. Explicitly specify array bounds, even if implicitly defined by an initializer STR32-C. Do not pass a non-null-terminated character sequence to a library function that expects a string
SEI CERT C++ Coding Standard	STR08-CPP. Do not specify the bound of a character array initialized with a string literal
ISO/IEC TR 24772:2013	String Termination [CJM]

Bibliography

[ECTC 1998]	Section A.8, "Character Array Initialization"
[ISO/IEC 9899:2011]	Subclause 6.7.9, "Initialization"
[Seacord 2013]	Chapter 2, "Strings"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/GoEAAQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
bounded_string_array	Avoid explicit array bound for arrays initialized with a string literal.
string_array_too_small	Array bound {} is too small for the string, should at least be {}.

CertC-STR30

Do not attempt to modify string literals.

Input: IR

Source languages: C, C++

Details

According to the C Standard, 6.4.5, paragraph 3 [[ISO/IEC 9899:2011](#)]:

A *character string literal* is a sequence of zero or more multibyte characters enclosed in double-quotes, as in "xyz". A UTF-8 string literal is the same, except prefixed by `u8`. A wide string literal is the same, except prefixed by the letter `L`, `u`, or `U`.

At compile time, string literals are used to create an array of static storage duration of sufficient length to contain the character sequence and a terminating null character. String literals are usually referred to by a pointer to (or array of) characters. Ideally, they should be assigned only to pointers to (or arrays of) `const char` or `const wchar_t`. It is unspecified whether these arrays of string literals are distinct from each other. The behavior is [undefined](#) if a program attempts to modify any portion of a string literal. Modifying a string literal frequently results in an access violation because string literals are typically stored in read-only memory. (See [undefined behavior 33](#).)

Avoid assigning a string literal to a pointer to `non-const` or casting a string literal to a pointer to `non-const`. For the purposes of this rule, a pointer to (or array of) `const` characters must be treated as a string literal. Similarly, the returned value of the following library functions must be treated as a string literal if the first argument is a string literal:

- `strpbrk()`, `strchr()`, `strrchr()`, `strstr()`
- `wcsnbrk()`, `wcschr()`, `wcsrchr()`, `wcsstr()`
- `memchr()`, `wmemchr()`

This rule is a specific instance of [EXP40-C. Do not modify constant objects](#).

Noncompliant Code Example

In this noncompliant code example, the `char` pointer `p` is initialized to the address of a string literal. Attempting to modify the string literal is [undefined behavior](#):

```
char *p = "string literal";
p[0] = 'S';
```

Compliant Solution

As an array initializer, a string literal specifies the initial values of characters in an array as well as the size of the array. (See [STR11-C. Do not specify the bound of a character array initialized with a string literal](#).) This code creates a copy of the string literal in the space allocated to the character array `a`. The string stored in `a` can be modified safely.

```
char a[] = "string literal";
a[0] = 'S';
```

Noncompliant Code Example (POSIX)

In this noncompliant code example, a string literal is passed to the (pointer to `non-const`) parameter of the POSIX function `mkstemp()`, which then modifies the characters of the string literal:

```
#include <stdlib.h>

void func(void) {
    mkstemp("/tmp/edXXXXXX");
}
```

The behavior of `mkstemp()` is described in more detail in [F1021-C. Do not create temporary files in shared directories](#).

Compliant Solution (POSIX)

This compliant solution uses a named array instead of passing a string literal:

```
#include <stdlib.h>

void func(void) {
    static char fname[] = "/tmp/edXXXXXX";
    mkstemp(fname);
}
```

Noncompliant Code Example (Result of `strrchr()`)

In this noncompliant example, the `char *` result of the `strrchr()` function is used to modify the object pointed to by `pathname`. Because the argument to `strrchr()` points to a string literal, the effects of the modification are undefined.

```
#include <stdio.h>
#include <string.h>

const char *get dirname(const char *pathname) {
    char *slash;
    slash = strrchr(pathname, '/');
    if (slash) {
        *slash = '\0'; /* Undefined behavior */
    }
    return pathname;
}

int main(void) {
    puts(get dirname(__FILE__));
    return 0;
}
```

Compliant Solution (Result of `strrchr()`)

This compliant solution avoids modifying a `const` object, even if it is possible to obtain a non-`const` pointer to such an object by calling a standard C library function, such as `strrchr()`. To reduce the risk to callers of `get dirname()`, a buffer and length for the directory name are passed into the function. It is insufficient to change `pathname` to require a `char *` instead of a `const char *` because conforming compilers are not required to diagnose passing a string literal to a function accepting a `char *`.

```
#include <stddef.h>
#include <stdio.h>
#include <string.h>

char *get dirname(const char *pathname, char *dirname, size_t size) {
    const char *slash;
    slash = strrchr(pathname, '/');
    if (slash) {
        ptrdiff_t slash_idx = slash - pathname;
        if ((size_t)slash_idx < size) {
            memcpy(dirname, pathname, slash_idx);
            dirname[slash_idx] = '\0';
            return dirname;
        }
    }
    return 0;
}

int main(void) {
    char dirname[260];
    if (get dirname(__FILE__, dirname, sizeof(dirname))) {
        puts(dirname);
    }
    return 0;
}
```

Risk Assessment

Modifying string literals can lead to [abnormal program termination](#) and possibly [denial-of-service attacks](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR30-C	Low	Likely	Low	P9	L2

Related Guidelines

CERT C Secure Coding Standard	EXP05-C. Do not cast away a const qualification STR11-C. Do not specify the bound of a character array initialized with a string literal
ISO/IEC TS 17961:2013	Modifying string literals [strmod]

Bibliography

[ISO/IEC 9899:2011]	6.4.5, "String Literals"
[Plum 1991]	Topic 1.26, "Strings-String Literals"
[Summit 1995]	comp.lang.c FAQ List, Question 1.32

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/TQE>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
funcs		{'strpbrk', 'strchr', 'strrchr', 'strstr', 'wcspbrk', 'wcschr', 'wcsrchr', 'wcsstr', 'memchr', 'wmemchr'}
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	9
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
call_with_const	Result of call to {}() with '{}' input should be used as 'const {}*'.
call_with_literal	Result of call to {}() with string literal should be used as 'const {}*'.
nonconst_string_literal	String literal should only be used as 'const char*'

CertC-STR31

Guarantee that storage for strings has sufficient space for character data and the null terminator.

Input: IR

Source languages: C, C++

Details

Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. Buffer overflows occur frequently when manipulating strings [[Seacord 2013b](#)]. To prevent such errors, either limit copies through truncation or, preferably, ensure that the destination is of sufficient size to hold the character data to be copied and the null-termination character. (See [STR03-C. Do not inadvertently truncate a string](#).)

When strings live on the heap, this rule is a specific instance of [MEM35-C. Allocate sufficient memory for an object](#). Because strings are represented as arrays of characters, this rule is related to both [ARR30-C. Do not form or use out-of-bounds pointers or array subscripts](#) and [ARR38-C. Guarantee that library functions do not form invalid pointers](#).

Noncompliant Code Example (Off-by-One Error)

This noncompliant code example demonstrates an *off-by-one* error [[Dowd 2006](#)]. The loop copies data from `src` to `dest`. However, because the loop does not account for the null-termination character, it may be incorrectly written 1 byte past the end of `dest`.

```
#include <stddef.h>

void copy(size_t n, char src[n], char dest[n]) {
    size_t i;

    for (i = 0; src[i] && (i < n); ++i) {
        dest[i] = src[i];
    }
    dest[i] = '\0';
}
```

Compliant Solution (Off-by-One Error)

In this compliant solution, the loop termination condition is modified to account for the null-termination character that is appended to `dest`:

```
#include <stddef.h>
```

```

void copy(size_t n, char src[n], char dest[n]) {
    size_t i;

    for (i = 0; src[i] && (i < n - 1); ++i) {
        dest[i] = src[i];
    }
    dest[i] = '\0';
}

```

Noncompliant Code Example (`gets()`)

The `gets()` function, which was deprecated in the C99 Technical Corrigendum 3 and removed from C11, is inherently unsafe and should never be used because it provides no way to control how much data is read into a buffer from `stdin`. This noncompliant code example assumes that `gets()` will not read more than `BUFFER_SIZE - 1` characters from `stdin`. This is an invalid assumption, and the resulting operation can result in a buffer overflow.

The `gets()` function reads characters from the `stdin` into a destination array until end-of-file is encountered or a newline character is read. Any newline character is discarded, and a null character is written immediately after the last character read into the array.

```

#include <stdio.h>

#define BUFFER_SIZE 1024

void func(void) {
    char buf[BUFFER_SIZE];
    if (gets(buf) == NULL) {
        /* Handle error */
    }
}

```

See also [MSC24-C. Do not use deprecated or obsolescent functions](#).

Compliant Solution (`fgets()`)

The `fgets()` function reads, at most, one less than the specified number of characters from a stream into an array. This solution is compliant because the number of characters copied from `stdin` to `buf` cannot exceed the allocated memory:

```

#include <stdio.h>
#include <string.h>

enum { BUFSIZE = 32 };

void func(void) {
    char buf[BUFSIZE];
    int ch;

    if (fgets(buf, sizeof(buf), stdin)) {
        /* fgets() succeeded; scan for newline character */
        char *p = strchr(buf, '\n');
        if (p) {
            *p = '\0';
        } else {
            /* Newline not found; flush stdin to end of line */
            while ((ch = getchar()) != '\n' && ch != EOF)
                ;
            if (ch == EOF && !feof(stdin) && !ferror(stdin)) {
                /* Character resembles EOF; handle error */
            }
        }
    } else {
        /* fgets() failed; handle error */
    }
}

```

The `fgets()` function is not a strict replacement for the `gets()` function because `fgets()` retains the newline character (if read) and may also return a partial line. It is possible to use `fgets()` to safely process input lines too long to store in the destination array, but this is not recommended for performance reasons. Consider using one of the following compliant solutions when replacing `gets()`.

Compliant Solution (`gets_s()`)

The `gets_s()` function reads, at most, one less than the number of characters specified from the stream pointed to by `stdin` into an array.

The C Standard, Annex K [[ISO/IEC 9899:2011](#)], states

No additional characters are read after a new-line character (which is discarded) or after end-of-file. The discarded new-line character does not count towards number of characters read. A null character is written immediately after the last character read into the array.

If end-of-file is encountered and no characters have been read into the destination array, or if a read error occurs during the operation, then the first character in the destination array is set to the null character and the other elements of the array take unspecified values:

```

#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>

enum { BUFSIZE = 32 };

void func(void) {
    char buf[BUFSIZE];

    if (gets_s(buf, sizeof(buf)) == NULL) {
        /* Handle error */
    }
}

```

```
}
```

Compliant Solution (`getline()`, POSIX)

The `getline()` function is similar to the `fgets()` function but can dynamically allocate memory for the input buffer. If passed a null pointer, `getline()` dynamically allocates a buffer of sufficient size to hold the input. If passed a pointer to dynamically allocated storage that is too small to hold the contents of the string, the `getline()` function resizes the buffer, using `realloc()`, rather than truncating the input. If successful, the `getline()` function returns the number of characters read, which can be used to determine if the input has any null characters before the newline. The `getline()` function works only with dynamically allocated buffers. Allocated memory must be explicitly deallocated by the caller to avoid memory leaks. (See [MEM31-C. Free dynamically allocated memory when no longer needed](#).)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void func(void) {
    int ch;
    size_t buffer_size = 32;
    char *buffer = malloc(buffer_size);

    if (!buffer) {
        /* Handle error */
        return;
    }

    if ((ssize_t size = getline(&buffer, &buffer_size, stdin))
        == -1) {
        /* Handle error */
    } else {
        char *p = strchr(buffer, '\n');
        if (p) {
            *p = '\0';
        } else {
            /* Newline not found; flush stdin to end of line */
            while ((ch = getchar()) != '\n' && ch != EOF)
                ;
            if (ch == EOF && !feof(stdin) && !ferror(stdin)) {
                /* Character resembles EOF; handle error */
            }
        }
    }
    free (buffer);
}
```

Note that the `getline()` function uses an [in-band error indicator](#), in violation of [ERR02-C. Avoid in-band error indicators](#).

Noncompliant Code Example (`getchar()`)

Reading one character at a time provides more flexibility in controlling behavior, though with additional performance overhead. This noncompliant code example uses the `getchar()` function to read one character at a time from `stdin` instead of reading the entire line at once. The `stdin` stream is read until end-of-file is encountered or a newline character is read. Any newline character is discarded, and a null character is written immediately after the last character read into the array. Similar to the noncompliant code example that invokes `gets()`, there are no guarantees that this code will not result in a buffer overflow.

```
#include <stdio.h>

enum { BUFSIZE = 32 };

void func(void) {
    char buf[BUFSIZE];
    char *p;
    int ch;
    p = buf;
    while ((ch = getchar()) != '\n' && ch != EOF) {
        *p++ = (char)ch;
    }
    *p++ = 0;
    if (ch == EOF) {
        /* Handle EOF or error */
    }
}
```

After the loop ends, if `ch == EOF`, the loop has read through to the end of the stream without encountering a newline character, or a read error occurred before the loop encountered a newline character. To conform to [FI034-C. Distinguish between characters read from a file and EOF or WEOF](#), the error-handling code must verify that an end-of-file or error has occurred by calling `feof()` or `ferror()`.

Compliant Solution (`getchar()`)

In this compliant solution, characters are no longer copied to `buf` once `index == BUFSIZE - 1`, leaving room to null-terminate the string. The loop continues to read characters until the end of the line, the end of the file, or an error is encountered. When `chars_read > index`, the input string has been truncated.

```
#include <stdio.h>

enum { BUFSIZE = 32 };

void func(void) {
    char buf[BUFSIZE];
    int ch;
    size_t index = 0;
```

```

size_t chars_read = 0;

while ((ch = getchar()) != '\n' && ch != EOF) {
    if (index < sizeof(buf) - 1) {
        buf[index++] = (char)ch;
    }
    chars_read++;
}

buf[index] = '\0'; /* Terminate string */
if (ch == EOF) {
    /* Handle EOF or error */
}
if (chars_read > index) {
    /* Handle truncation */
}
}

```

Noncompliant Code Example (`fscanf()`)

In this noncompliant example, the call to `fscanf()` can result in a write outside the character array `buf`:

```

#include <stdio.h>

enum { BUF_LENGTH = 1024 };

void get_data(void) {
    char buf[BUF_LENGTH];
    if (1 != fscanf(stdin, "%s", buf)) {
        /* Handle error */
    }

    /* Rest of function */
}

```

Compliant Solution (`fscanf()`)

In this compliant solution, the call to `fscanf()` is constrained not to overflow `buf`:

```

#include <stdio.h>

enum { BUF_LENGTH = 1024 };

void get_data(void) {
    char buf[BUF_LENGTH];
    if (1 != fscanf(stdin, "%1023s", buf)) {
        /* Handle error */
    }

    /* Rest of function */
}

```

Noncompliant Code Example (`argv`)

In a [hosted environment](#), arguments read from the command line are stored in process memory. The function `main()`, called at program startup, is typically declared as follows when the program accepts command-line arguments:

```
int main(int argc, char *argv[]) { /* ... */ }
```

Command-line arguments are passed to `main()` as pointers to strings in the array members `argv[0]` through `argv[argc - 1]`. If the value of `argc` is greater than 0, the string pointed to by `argv[0]` is, by convention, the program name. If the value of `argc` is greater than 1, the strings referenced by `argv[1]` through `argv[argc - 1]` are the program arguments.

[Vulnerabilities](#) can occur when inadequate space is allocated to copy a command-line argument or other program input. In this noncompliant code example, an attacker can manipulate the contents of `argv[0]` to cause a buffer overflow:

```

#include <string.h>

int main(int argc, char *argv[]) {
    /* Ensure argv[0] is not null */
    const char *const name = (argc && argv[0]) ? argv[0] : "";
    char prog_name[128];
    strcpy(prog_name, name);

    return 0;
}

```

Compliant Solution (`argv`)

The `strlen()` function can be used to determine the length of the strings referenced by `argv[0]` through `argv[argc - 1]` so that adequate memory can be dynamically allocated.

```

#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    /* Ensure argv[0] is not null */
    const char *const name = (argc && argv[0]) ? argv[0] : "";
    char *prog_name = (char *)malloc(strlen(name) + 1);

```

```

if (prog_name != NULL) {
    strcpy(prog_name, name);
} else {
    /* Handle error */
}
free(prog_name);
return 0;
}

```

Remember to add a byte to the destination string size to accommodate the null-termination character.

Compliant Solution (`argv`)

The `strcpy_s()` function provides additional safeguards, including accepting the size of the destination buffer as an additional argument. (See [STR07-C. Use the bounds-checking interfaces for string manipulation](#).)

```

#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    /* Ensure argv[0] is not null */
    const char *const name = (argc && argv[0]) ? argv[0] : "";
    char *prog_name;
    size_t prog_size;

    prog_size = strlen(name) + 1;
    prog_name = (char *)malloc(prog_size);

    if (prog_name != NULL) {
        if (strcpy_s(prog_name, prog_size, name)) {
            /* Handle error */
        }
    } else {
        /* Handle error */
    }
    /* ... */
    free(prog_name);
    return 0;
}

```

The `strcpy_s()` function can be used to copy data to or from dynamically allocated memory or a statically allocated array. If insufficient space is available, `strcpy_s()` returns an error.

Compliant Solution (`argv`)

If an argument will not be modified or concatenated, there is no reason to make a copy of the string. Not copying a string is the best way to prevent a buffer overflow and is also the most efficient solution. Care must be taken to avoid assuming that `argv[0]` is non-null.

```

int main(int argc, char *argv[]) {
    /* Ensure argv[0] is not null */
    const char * const prog_name = (argc && argv[0]) ? argv[0] : "";
    /* ... */
    return 0;
}

```

Noncompliant Code Example (`getenv()`)

According to the C Standard, 7.22.4.6 [[ISO/IEC 9899:2011](#)]

The `getenv` function searches an environment list, provided by the host environment, for a string that matches the string pointed to by `name`. The set of environment names and the method for altering the environment list are implementation defined.

Environment variables can be arbitrarily large, and copying them into fixed-length arrays without first determining the size and allocating adequate storage can result in a buffer overflow.

```

#include <stdlib.h>
#include <string.h>

void func(void) {
    char buff[256];
    char *editor = getenv("EDITOR");
    if (editor == NULL) {
        /* EDITOR environment variable not set */
    } else {
        strcpy(buff, editor);
    }
}

```

Compliant Solution (`getenv()`)

Environmental variables are loaded into process memory when the program is loaded. As a result, the length of these strings can be determined by calling the `strlen()` function, and the resulting length can be used to allocate adequate dynamic memory:

```

#include <stdlib.h>
#include <string.h>

void func(void) {

```

```

char *buff;
char *editor = getenv("EDITOR");
if (editor == NULL) {
    /* EDITOR environment variable not set */
} else {
    size_t len = strlen(editor) + 1;
    buff = (char *)malloc(len);
    if (buff == NULL) {
        /* Handle error */
    }
    memcpy(buff, editor, len);
    free(buff);
}

```

Noncompliant Code Example (`sprintf()`)

In this noncompliant code example, `name` refers to an external string; it could have originated from user input, the file system, or the network. The program constructs a file name from the string in preparation for opening the file.

```

#include <stdio.h>

void func(const char *name) {
    char filename[128];
    sprintf(filename, "%s.txt", name);
}

```

Because the `sprintf()` function makes no guarantees regarding the length of the generated string, a sufficiently long string in `name` could generate a buffer overflow.

Compliant Solution (`sprintf()`)

The buffer overflow in the preceding noncompliant example can be prevented by adding a precision to the `%s` conversion specification. If the precision is specified, no more than that many bytes are written. The precision `123` in this compliant solution ensures that `filename` can contain the first 123 characters of `name`, the `.txt` extension, and the null terminator.

```

#include <stdio.h>

void func(const char *name) {
    char filename[128];
    sprintf(filename, ".%123s.txt", name);
}

```

Compliant Solution (`snprintf()`)

A more general solution is to use the `snprintf()` function:

```

#include <stdio.h>

void func(const char *name) {
    char filename[128];
    snprintf(filename, sizeof(filename), "%s.txt", name);
}

```

Risk Assessment

Copying string data to a buffer that is too small to hold that data results in a buffer overflow. Attackers can exploit this condition to execute arbitrary code with the permissions of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR31-C	High	Likely	Medium	P18	L1

Related Guidelines

CERT C Secure Coding Standard	STR03-C. Do not inadvertently truncate a string STR07-C. Use the bounds-checking interfaces for remediation of existing string manipulation code MSC24-C. Do not use deprecated or obsolescent functions MEM00-C. Allocate and free memory in the same module, at the same level of abstraction FI034-C. Distinguish between characters read from a file and EOF or WEOF
ISO/IEC TR 24772:2013	String Termination [CJM] Buffer Boundary Violation (Buffer Overflow) [HCB] Unchecked Array Copying [XYW]
ISO/IEC TS 17961:2013	Using a tainted value to write to an object using a formatted input or output function [taintformatio] Tainted strings are passed to a string copying function [taintstrcpy]
MITRE CWE	CWE-119 , Improper Restriction of Operations within the Bounds of a Memory Buffer CWE-120 , Buffer Copy without Checking Size of Input ("Classic Buffer Overflow") CWE-123 , Write-what-where Condition CWE-125 , Out-of-bounds Read CWE-193 , Off-by-one Error

Bibliography

[Dowd 2006]	Chapter 7, "Program Building Blocks" ("Loop Constructs," pp. 327-336)
[Drepper 2006]	Section 2.1.1, "Respecting Memory Bounds"
[ISO/IEC 9899:2011]	K.3.5.4.1, "The <code>gets_s</code> Function"
[Lai 2006]	
[NIST 2006]	SAMATE Reference Dataset Test Case ID 000-000-088
[Seacord 2013b]	Chapter 2, "Strings"
[xorl 2009]	FreeBSD-SA-09:11: NTPd Remote Stack Based Buffer Overflows

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/KAE>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
allow_extra_args	Whether to allow additional arguments that are not used by the format string.	True
allow_gnu_extensions	Whether to allow the GNU extensions to format specifications.	True
allow_unknown_specs	Whether to allow unknown format specifications. It may be necessary to set this option when using implementation-specific extensions. Arguments are not checked when the format string contains unknown format specifications.	True
blacklist	Dictionary of header globbing to (list of) function name globbing(s) of forbidden functions.	dict(...)
functions	A dictionary mapping the names of the functions to check, to a triple {function_kind, fmt_param_index, arg_start_index} where function_kind is either 'printf' or 'scanf', fmt_param_index is the index of the format-string parameter, and arg_start_index is the index of the first variadic argument.	dict(...)
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
arg_type_mismatch	{} expects argument of type '{}', but argument {} has type '{}' (disabled)
buffer_too_small	{} may write up to {} characters to buffer of size {}.
forbidden_libfunc_call	Call to forbidden function.
invalid_conversion	Invalid or non-standard conversion specification (disabled)
matching_arg_expected	{} expects a matching '{}' argument (disabled)
maybe_too_small	Target buffer may be too small. Use snprintf() instead.
precision_for_conversion	Precision must not be used with %{} conversion specifier (disabled)
too_many_args	Too many arguments for format. (disabled)
too_small	Target buffer has {} characters, but sprintf() may write up to {} characters (including null terminator).
unknown_buffer_size	Potential buffer overflow: {} used with buffer of unknown size.
unlimited_read	Potential buffer overflow: {} has no limit on amount of characters read.
unsupported_assignment_suppression	%n does not support assignment suppression (disabled)
unsupported_field_width	%n does not support field width (disabled)
unsupported_flags	%n does not support flags (disabled)
unsupported_flags_modifiers	Cannot use any flags or modifiers with '%%' (disabled)
unsupported_hash	%{} does not support the '#' flag (disabled)
unsupported_i_flag	%{} does not support the 'l' flag (disabled)
unsupported_length_modifier	%{} does not support the '{}' length modifier (disabled)
unsupported_tick	%{} does not support the "" flag (disabled)
unsupported_zero	%{} does not support the '0' flag (disabled)

CertC-STR32

Do not pass a non-null-terminated character sequence to a library function that expects a string.

Input: IR

Source languages: C, C++

Details

Many library functions accept a string or wide string argument with the constraint that the string they receive is properly null-terminated. Passing a character sequence or wide character sequence that is not null-terminated to such a function can result in accessing memory that is outside the bounds of the object. Do not pass a character sequence or wide character sequence that is not null-terminated to a library function that expects a string or wide string argument.

Noncompliant Code Example

This code example is noncompliant because the character sequence `c_str` will not be null-terminated when passed as an argument to `printf()`. [See [STR11-C. Do not specify the bound of a character array initialized with a string literal](#) on how to properly initialize character arrays.]

```
#include <stdio.h>

void func(void) {
    char c_str[3] = "abc";
    printf("%s\n", c_str);
}
```

Compliant Solution

This compliant solution does not specify the bound of the character array in the array declaration. If the array bound is omitted, the compiler allocates sufficient storage to store the entire string literal, including the terminating null character.

```
#include <stdio.h>

void func(void) {
    char c_str[] = "abc";
    printf("%s\n", c_str);
}
```

Noncompliant Code Example

This code example is noncompliant because the wide character sequence `cur_msg` will not be null-terminated when passed to `wcslen()`. This will occur if `lessen_memory_usage()` is invoked while `cur_msg_size` still has its initial value of 1024.

```
#include <stdlib.h>
#include <wchar.h>

wchar_t *cur_msg = NULL;
size_t cur_msg_size = 1024;
size_t cur_msg_len = 0;

void lessen_memory_usage(void) {
    wchar_t *temp;
    size_t temp_size;

    /* ... */

    if (cur_msg != NULL) {
        temp_size = cur_msg_size / 2 + 1;
        temp = realloc(cur_msg, temp_size * sizeof(wchar_t));
        /* temp &ndash; cur_msg may no longer be null-terminated */
        if (temp == NULL) {
            /* Handle error */
        }

        cur_msg = temp;
        cur_msg_size = temp_size;
        cur_msg_len = wcslen(cur_msg);
    }
}
```

Compliant Solution

In this compliant solution, `cur_msg` will always be null-terminated when passed to `wcslen()`:

```
#include <stdlib.h>
#include <wchar.h>

wchar_t *cur_msg = NULL;
size_t cur_msg_size = 1024;
size_t cur_msg_len = 0;

void lessen_memory_usage(void) {
    wchar_t *temp;
    size_t temp_size;

    /* ... */

    if (cur_msg != NULL) {
        temp_size = cur_msg_size / 2 + 1;
        temp = realloc(cur_msg, temp_size * sizeof(wchar_t));
        /* temp and cur_msg may no longer be null-terminated */
        if (temp == NULL) {
            /* Handle error */
        }

        cur_msg = temp;
        /* Properly null-terminate cur_msg */
        cur_msg[temp_size - 1] = L'\0';
        cur_msg_size = temp_size;
        cur_msg_len = wcslen(cur_msg);
    }
}
```

Noncompliant Code Example (`strncpy()`)

Although the `strncpy()` function takes a string as input, it does not guarantee that the resulting value is still null-terminated. In the following noncompliant code example, if no null character is contained in the first `n` characters of the `source` array, the result will not be null-terminated. Passing a non-null-terminated character sequence to `strlen()` is undefined behavior.

```
#include <string.h>

enum { STR_SIZE = 32 };

size_t func(const char *source) {
    char c_str[STR_SIZE];
    size_t ret = 0;

    if (source) {
        c_str[sizeof(c_str) - 1] = '\0';
        strncpy(c_str, source, sizeof(c_str));
        ret = strlen(c_str);
    } else {
        /* Handle null pointer */
    }
    return ret;
}
```

Compliant Solution (Truncation)

This compliant solution is correct if the programmer's intent is to truncate the string:

```
#include <string.h>

enum { STR_SIZE = 32 };

size_t func(const char *source) {
    char c_str[STR_SIZE];
    size_t ret = 0;

    if (source) {
        strncpy(c_str, source, sizeof(c_str) - 1);
        c_str[sizeof(c_str) - 1] = '\0';
        ret = strlen(c_str);
    } else {
        /* Handle null pointer */
    }
    return ret;
}
```

Compliant Solution (Truncation, `strncpy_s()`)

The C Standard, Annex K `strncpy_s()` function can also be used to copy with truncation. The `strncpy_s()` function copies up to `n` characters from the source array to a destination array. If no null character was copied from the source array, then the `n`th position in the destination array is set to a null character, guaranteeing that the resulting string is null-terminated.

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>

enum { STR_SIZE = 32 };

size_t func(const char *source) {
    char a[STR_SIZE];
    size_t ret = 0;

    if (source) {
        errno_t err = strncpy_s(
            a, sizeof(a), source, strlen(source)
        );
        if (err != 0) {
            /* Handle error */
        } else {
            ret = strnlen_s(a, sizeof(a));
        }
    } else {
        /* Handle null pointer */
    }
    return ret;
}
```

Compliant Solution (Copy without Truncation)

If the programmer's intent is to copy without truncation, this compliant solution copies the data and guarantees that the resulting array is null-terminated. If the string cannot be copied, it is handled as an error condition.

```
#include <string.h>

enum { STR_SIZE = 32 };

size_t func(const char *source) {
    char c_str[STR_SIZE];
    size_t ret = 0;

    if (source) {
        if (strlen(source) < sizeof(c_str)) {
            strcpy(c_str, source);
            ret = strlen(c_str);
        } else {
            /* Handle string-too-large */
        }
    } else {
        /* Handle null pointer */
    }
    return ret;
}
```

Risk Assessment

Failure to properly null-terminate a character sequence that is passed to a library function that expects a string can result in buffer overflows and the execution of arbitrary code with the permissions of the vulnerable process. Null-termination errors can also result in unintended information disclosure.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR32-C	High	Probable	Medium	P12	L1

Related Guidelines

ISO/IEC TR 24772:2013	String Termination [CMJ]
ISO/IEC TS 17961:2013	Passing a non-null-terminated character sequence to a library function that expects a string [strmod]
MITRE CWE	CWE-119 , Improper Restriction of Operations within the Bounds of a Memory Buffer CWE-123 , Write-what-where Condition CWE-125 , Out-of-bounds Read CWE-170 , Improper Null Termination

Bibliography

[Seacord 2013]	Chapter 2, "Strings"
[Viega 2005]	Section 5.2.14, "Miscalculated <code>NULL</code> Termination"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/KgE>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
functions_under_test	Functions which should not use non-null-terminated character sequences.	<code>['strcpy', 'strncpy', 'strcat', 'strncat', 'strcmp', 'strncmp', 'strcoll', 'strxfrm', 'strchr', 'strcspn', 'strpbrk', 'strchr', 'strrchr', 'strspn', 'strtok', 'strlen', 'strftime', 'fprintf', 'printf', 'fscanf', 'scanf', 'snprintf', 'sprintf', 'sscanf', 'vfprintf', 'vfscanf', 'vprintf', 'vscanf', 'vsnprintf', 'vsprintf', 'vsscanf', 'fwprintf', 'fwscanf', 'swprintf', 'swscanf', 'vswprintf', 'vswscanf', 'vwprintf', 'wprintf', 'wscanf', 'fputws', 'wcstod', 'wcstold', 'wcstol', 'wcstoll', 'wcstoul', 'wcscpy', 'wcsncpy', 'wcsncat', 'wcscmp', 'wcscoll', 'wcscncmp', 'wcsxfrm', 'wcschr', 'wcscspn', 'wcspbrk', 'wcsrchr', 'wcspchr', 'wcspn', 'wcstr', 'wcstok', 'wcslen', 'wcsftime', 'mbrtoc16', 'c16rtomb', 'mbrtoc32', 'c32rtomb']</code>
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
non-null-termination	Do not pass a non-null-terminated character sequence to a library function that expects a string.

CertC-STR34

Cast characters to unsigned char before converting to larger integer sizes.

Input: IR

Source languages: C, C++

Details

Signed character data must be converted to `unsigned char` before being assigned or converted to a larger signed type. This rule applies to both `signed char` and `(plain) char` characters on implementations where `char` is defined to have the same range, representation, and behaviors as `signed char`.

However, this rule is applicable only in cases where the character data may contain values that can be interpreted as negative numbers. For example, if the `char` type is represented by a two's complement 8-bit value, any character value greater than +127 is interpreted as a negative value.

This rule is a generalization of [STR37-C. Arguments to character-handling functions must be representable as an `unsigned char`.](#)

Noncompliant Code Example

This noncompliant code example is taken from a [vulnerability](#) in bash versions 1.14.6 and earlier that led to the release of CERT Advisory [CA-1996-22](#). This vulnerability resulted from the sign extension of character data referenced by the `c_str` pointer in the `yy_string_get()` function in the `parse.y` module of the bash source code:

```
static int yy_string_get(void) {
    register char *c_str;
    register int c;

    c_str = bash_input.location.string;
    c = EOF;

    /* If the string doesn't exist or is empty, EOF found */
    if (c_str && *c_str) {
        c = *c_str++;
        bash_input.location.string = c_str;
    }
    return (c);
}
```

The `c_str` variable is used to traverse the character string containing the command line to be parsed. As characters are retrieved from this pointer, they are stored in a variable of type `int`. For implementations in which the `char` type is defined to have the same range, representation, and behavior as `signed char`, this value is sign-extended when assigned to the `int` variable. For character code 255 decimal (-1 in two's complement form), this sign extension results in the value -1 being assigned to the integer, which is indistinguishable from `EOF`.

Noncompliant Code Example

This problem can be repaired by explicitly declaring the `c_str` variable as `unsigned char`:

```
static int yy_string_get(void) {
    register unsigned char *c_str;
    register int c;

    c_str = bash_input.location.string;
    c = EOF;

    /* If the string doesn't exist or is empty, EOF found */
    if (c_str && *c_str) {
        c = *c_str++;
        bash_input.location.string = c_str;
    }
    return (c);
}
```

This example, however, violates [STR04-C. Use plain `char` for characters in the basic character set.](#)

Compliant Solution

In this compliant solution, the result of the expression `*c_str++` is cast to `unsigned char` before assignment to the `int` variable `c`:

```
static int yy_string_get(void) {
    register char *c_str;
    register int c;

    c_str = bash_input.location.string;
    c = EOF;

    /* If the string doesn't exist or is empty, EOF found */
    if (c_str && *c_str) {
        /* Cast to unsigned type */
        c = (unsigned char)*c_str++;

        bash_input.location.string = c_str;
    }
    return (c);
}
```

Noncompliant Code Example

In this noncompliant code example, the cast of `*s` to `unsigned int` can result in a value in excess of `UCHAR_MAX` because of integer promotions, a violation of [ARR30-C. Do not form or use out-of-bounds pointers or array subscripts.](#):

```
#include <limits.h>
#include <stddef.h>

static const char table[UCHAR_MAX + 1] = { 'a' /* ... */ };

ptrdiff_t first_not_in_table(const char *c_str) {
    for (const char *s = c_str; *s; ++s) {
        if (table[(unsigned int)*s] != *s) {
            return s - c_str;
        }
    }
    return -1;
}
```

Compliant Solution

This compliant solution casts the value of type `char` to `unsigned char` before the implicit promotion to a larger type:

```
#include <limits.h>
#include <stddef.h>

static const char table[UCHAR_MAX + 1] = { 'a' /* ... */ };

ptrdiff_t first_not_in_table(const char *c_str) {
    for (const char *s = c_str; *s; ++s) {
        if ((table[(unsigned char)*s] != *s) {
            return s - c_str;
        }
    }
    return -1;
}
```

Risk Assessment

Conversion of character data resulting in a value in excess of `UCHAR_MAX` is an often-missed error that can result in a disturbingly broad range of potentially severe [vulnerabilities](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR34-C	Medium	Probable	Medium	P8	L2

Related Guidelines

CERT C Secure Coding Standard	STR37-C. Arguments to character-handling functions must be representable as an unsigned char STR04-C. Use plain char for characters in the basic character set ARR30-C. Do not form or use out-of-bounds pointers or array subscripts
ISO/IEC TS 17961:2013	Conversion of signed characters to wider integer types before a check for EOF [signconv]
MISRA-C:2012	Rule 10.1 (required) Rule 10.2 (required) Rule 10.3 (required) Rule 10.4 (required)
MITRE CWE	CWE-704 , Incorrect Type Conversion or Cast

Bibliography

[xorL 2009]	CVE-2009-0887: Linux-PAM Signedness Issue
-----------------------------	---

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/QgBi>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
only_arguments_of	Can be used to provide a set of function/macro names; only arguments to them will be considered then	set([])
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_from_char_to_larger_type	Cast characters to unsigned char before converting to larger integer sizes

CertC-STR37

Arguments to character-handling functions must be representable as an unsigned char.

Input: IR

Source languages: C, C++

Details

According to the C Standard, 7.4 [[ISO/IEC 9899:2011](#)],

The header `<ctype.h>` declares several functions useful for classifying and mapping characters. In all cases the argument is an `int`, the value of which shall be representable as an `unsigned char` or shall equal the value of the macro `EOF`. If the argument has any other value, the behavior is [undefined](#).

See also [undefined behavior 113](#).

This rule is applicable only to code that runs on platforms where the `char` data type is defined to have the same range, representation, and behavior as `signed char`.

Following are the character classification functions that this rule addresses:

isalnum	isalpha	isascii ^{XSI}	isblank
iscntrl	isdigit	isgraph	islower
isprint	ispunct	isspace	isupper
isxdigit	toascii ^{XSI}	toupper	tolower

XSI denotes an X/Open System Interfaces Extension to ISO/IEC 9945-POSIX. These functions are not defined by the C Standard.

This rule is a specific instance of [STR34-C. Cast characters to unsigned char before converting to larger integer sizes](#).

Noncompliant Code Example

On implementations where plain `char` is signed, this code example is noncompliant because the parameter to `isspace`, `*t`, is defined as a `const char *`, and this value might not be representable as an `unsigned char`:

```
#include <ctype.h>
#include <string.h>

size_t count_preceding_whitespace(const char *s) {
    const char *t = s;
    size_t length = strlen(s) + 1;
    while (isspace(*t) && (t - s < length)) {
        ++t;
    }
    return t - s;
}
```

The argument to `isspace` must be `EOF` or representable as an `unsigned char`; otherwise, the result is undefined.

Compliant Solution

This compliant solution casts the character to `unsigned char` before passing it as an argument to the `isspace` function:

```
#include <ctype.h>
#include <string.h>

size_t count_preceding_whitespace(const char *s) {
    const char *t = s;
    size_t length = strlen(s) + 1;
    while (isspace((unsigned char)*t) && (t - s < length)) {
        ++t;
    }
    return t - s;
}
```

Risk Assessment

Passing values to character handling functions that cannot be represented as an `unsigned char` to character handling functions is [undefined behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR37-C	Low	Unlikely	Low	P3	L3

CERT C Secure Coding Standard	STR34-C. Cast characters to unsigned char before converting to larger integer sizes
ISO/IEC TS 17961	Passing arguments to character-handling functions that are not representable as unsigned char [chrsgnext]
MITRE CWE	CWE-704 , Incorrect Type Conversion or Cast CWE-686 , Function Call with Incorrect Argument Type

Bibliography

[ISO/IEC 9899:2011]	7.4, "Character Handling <ctype.h>"
[Kettlewell 2002]	Section 1.1, "<ctype.h> and Characters Types"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/fAs>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
only_arguments_of	Can be used to provide a set of function/macro names; only arguments to them will be considered then	set(['isupper', 'toupper', 'isalnum', 'toascii', 'isxdigit', 'isgraph', 'isspace', 'iscntrl', 'islower', 'isprint', 'isalpha', 'isascii', 'isdigit', 'ispunct', 'isblank', 'tolower'])
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_from_char_to_larger_type	Arguments to character-handling functions must be representable as an unsigned char

CertC-STR38

Do not confuse narrow and wide character strings and functions.

Input: IR

Source languages: C, C++

Details

Passing narrow string arguments to wide string functions or wide string arguments to narrow string functions can lead to [unexpected](#) and [undefined behavior](#). Scaling problems are likely because of the difference in size between wide and narrow characters. (See [ARR39-C. Do not add or subtract a scaled integer to a pointer](#).) Because wide strings are terminated by a null wide character and can contain null bytes, determining the length is also problematic.

Because `wchar_t` and `char` are distinct types, many compilers will produce a warning diagnostic if an inappropriate function is used. (See [MSC00-C. Compile cleanly at high warning levels](#).)

Noncompliant Code Example (Wide Strings with Narrow String Functions)

This noncompliant code example incorrectly uses the `strncpy()` function in an attempt to copy up to 10 wide characters. However, because wide characters can contain null bytes, the copy operation may end earlier than anticipated, resulting in the truncation of the wide string.

```
#include <stddef.h>
#include <string.h>
```

```

void func(void) {
    wchar_t wide_str1[] = L"0123456789";
    wchar_t wide_str2[] = L"0000000000";
    strncpy(wide_str2, wide_str1, 10);
}

```

Noncompliant Code Example (Narrow Strings with Wide String Functions)

This noncompliant code example incorrectly invokes the `wcsncpy()` function to copy up to 10 wide characters from `narrow_str1` to `narrow_str2`. Because `narrow_str2` is a narrow string, it has insufficient memory to store the result of the copy and the copy will result in a buffer overflow.

```

#include <wchar.h>

void func(void) {
    char narrow_str1[] = "01234567890123456789";
    char narrow_str2[] = "0000000000";
    wcsncpy(narrow_str2, narrow_str1, 10);
}

```

Compliant Solution

This compliant solution uses the proper-width functions. Using `wcsncpy()` for wide character strings and `strncpy()` for narrow character strings ensures that data is not truncated and buffer overflow does not occur.

```

#include <string.h>
#include <wchar.h>

void func(void) {
    wchar_t wide_str1[] = L"0123456789";
    wchar_t wide_str2[] = L"0000000000";
    /* Use of proper-width function */
    wcsncpy(wide_str2, wide_str1, 10);

    char narrow_str1[] = "0123456789";
    char narrow_str2[] = "0000000000";
    /* Use of proper-width function */
    strncpy(narrow_str2, narrow_str1, 10);
}

```

Noncompliant Code Example (`strlen()`)

In this noncompliant code example, the `strlen()` function is used to determine the size of a wide character string:

```

#include <stdlib.h>
#include <string.h>

void func(void) {
    wchar_t wide_str1[] = L"0123456789";
    wchar_t *wide_str2 = (wchar_t *)malloc(strlen(wide_str1) + 1);
    if (wide_str2 == NULL) {
        /* Handle error */
    }
    /* ... */
    free(wide_str2);
    wide_str2 = NULL;
}

```

The `strlen()` function determines the number of characters that precede the terminating null character. However, wide characters can contain null bytes, particularly when expressing characters from the ASCII character set, as in this example. As a result, the `strlen()` function will return the number of bytes preceding the first null byte in the wide string.

Compliant Solution

This compliant solution correctly calculates the number of bytes required to contain a copy of the wide string, including the terminating null wide character:

```

#include <stdlib.h>
#include <wchar.h>

void func(void) {
    wchar_t wide_str1[] = L"0123456789";
    wchar_t *wide_str2 = (wchar_t *)malloc(
        (wcslen(wide_str1) + 1) * sizeof(wchar_t));
    if (wide_str2 == NULL) {
        /* Handle error */
    }
    /* ... */

    free(wide_str2);
    wide_str2 = NULL;
}

```

Risk Assessment

Confusing narrow and wide character strings can result in buffer overflows, data truncation, and other defects.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR38-C	High	Likely	Low	P27	L1

Bibliography

[ISO/IEC 9899:2011]	7.24.2.4, "The <code>strncpy</code> Function" 7.29.4.2.2, "The <code>wcsncpy</code> Function"
Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [https://www.securecoding.cert.org/confluence/x/FADAAQ], Copyright (C) 1995-2016 Carnegie Mellon University. See <code>axivion_copyright_guide.pdf</code> for full details.	

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	<code>msg_for_narrow_and_wide_char(['klass', 'node'])</code>
priority	Priority based on the combination of severity, likelihood and remediation cost	27
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low
reported_messages	If provided, only messages of these types are reported.	167
reported_severities	List of severities to display.	{'error', 'warning', 'remark'}
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

CertC-MEM01

Store a new value in pointers immediately after `free()`.

Input: IR

Source languages: C, C++

Details

Dangling pointers can lead to exploitable double-free and access-freed-memory [vulnerabilities](#). A simple yet effective way to eliminate dangling pointers and avoid many memory-related vulnerabilities is to set pointers to `NULL` after they are freed or to set them to another valid object.

Noncompliant Code Example

In this noncompliant code example, the type of a message is used to determine how to process the message itself. It is assumed that `message_type` is an integer and `message` is a pointer to an array of characters that were allocated dynamically. If `message_type` equals `value_1`, the message is processed accordingly. A similar operation occurs when `message_type` equals `value_2`. However, if `message_type == value_1` evaluates to true and `message_type == value_2` also evaluates to true, then `message` is freed twice, resulting in a double-free vulnerability.

```
char *message;
int message_type;

/* Initialize message and message_type */

if (message_type == value_1) {
    /* Process message type 1 */
    free(message);
}

/* ... */
if (message_type == value_2) {
    /* Process message type 2 */
    free(message);
}
```

Compliant Solution

Calling `free()` on a null pointer results in no action being taken by `free()`. Setting `message` to `NULL` after it is freed eliminates the possibility that the `message` pointer can be used to free the same memory more than once.

```
char *message;
int message_type;

/* Initialize message and message_type */

if (message_type == value_1) {
    /* Process message type 1 */
    free(message);
}
```

```

    message = NULL;
}
/* ... */
if (message_type == value_2) {
/* Process message type 2 */
free(message);
message = NULL;
}

```

Exceptions

MEM01-C-EX1: If a nonstatic variable goes out of scope immediately following the `free()`, it is not necessary to clear its value because it is no longer accessible.

```

void foo(void) {
    char *str;
    /* ... */
    free(str);
    return;
}

```

Risk Assessment

Setting pointers to `NULL` or to another valid value after memory is freed is a simple and easily implemented solution for reducing dangling pointers. Dangling pointers can result in freeing memory multiple times or in writing to memory that has already been freed. Both of these problems can lead to an attacker executing arbitrary code with the permissions of the vulnerable process.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
MEM01-C	High	Unlikely	Low	P9	L2

Related Guidelines

SEI CERT C++ Coding Standard	MEM01-CPP. Store a valid value in pointers immediately after deallocation
ISO/IEC TR 24772:2013	Dangling References to Stack Frames [DCM] Dangling Reference to Heap [XYK] Off-by-one Error [XZH]
MITRE CWE	CWE-415 , Double free CWE-416 , Use after free

Bibliography

[Seacord 2013]	Chapter 4, "Dynamic Memory Management"
[Plakosh 2005]	

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/uAE>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	9
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
free_without_pointer_reset	Call to <code>free()</code> not immediately followed by an assignment to the freed pointer.

CertC-MEM02

Immediately cast the result of a memory allocation function call into a pointer to the allocated type.

Input: IR

Source languages: C, C++

Details

An object of type `void *` is a generic data pointer. It can point to any data object. For any incomplete or object type `T`, C permits implicit conversion from `T *` to `void *` or from `void *` to `T *`. C Standard memory allocation functions `aligned_alloc()`, `malloc()`, `calloc()`, and `realloc()` use `void *` to declare parameters and return types of functions designed to work for objects of different types.

For example, the C library declares `malloc()` as

```
void *malloc(size_t);
```

Calling `malloc(s)` allocates memory for an object whose size is `s` and returns either a null pointer or a pointer to the allocated memory. A program can implicitly convert the pointer that `malloc()` returns into a different pointer type.

Because objects returned by the C Standard memory allocation functions are implicitly converted into any object type, we recommend casting the results of these functions into a pointer of the allocated type because it increases the chances that the compiler will catch and diagnose a mismatch between the intended type of the object and the actual type of the object.

Noncompliant Code Example

The argument to `malloc()` can be *any* value of (`unsigned`) type `size_t`. If the program uses the allocated storage to represent an object (possibly an array) whose size is greater than the requested size, the behavior is [undefined](#). The implicit pointer conversion lets this slip by without complaint from the compiler.

Consider the following example:

```
#include <stdlib.h>

typedef struct gadget gadget;
struct gadget {
    int i;
    double d;
};

typedef struct widget widget;
struct widget {
    char c[10];
    int i;
    double d;
};

widget *p;

/* ... */

p = malloc(sizeof(gadget)); /* Imminent problem */
if (p != NULL) {
    p->i = 0;                /* Undefined behavior */
    p->d = 0.0;              /* Undefined behavior */
}
```

An [implementation](#) may add padding to a gadget or widget so that `sizeof(gadget)` equals `sizeof(widget)`, but this is highly unlikely. More likely, `sizeof(gadget)` is less than `sizeof(widget)`. In that case,

```
p = malloc(sizeof(gadget)); /* Imminent problem */
```

quietly assigns `p` to point to storage too small for a widget. The subsequent assignments to `p->i` and `p->d` will most likely produce memory overruns.

Casting the result of `malloc()` to the appropriate pointer type enables the compiler to catch subsequent inadvertent pointer conversions. When allocating individual objects, the "appropriate pointer type" is a pointer to the type argument in the `sizeof` expression passed to `malloc()`.

In this code example, `malloc()` allocates space for a `gadget`, and the cast immediately converts the returned pointer to a `gadget *`:

```
widget *p;

/* ... */

p = (gadget *)malloc(sizeof(gadget)); /* Invalid assignment */
```

This lets the compiler detect the invalid assignment because it attempts to convert a `gadget *` into a `widget *`.

Compliant Solution (Hand Coded)

This compliant solution repeats the same type in the `sizeof` expression and the pointer cast:

```
widget *p;

/* ... */

p = (widget *)malloc(sizeof(widget));
```

Compliant Solution (Macros)

Repeating the same type in the `sizeof` expression and the pointer cast is easy to do but still invites errors. Packaging the repetition in a macro, such as

```
#define MALLOC(type) ((type *)malloc(sizeof(type)))
```

further reduces the possibility of error.

```
widget *p;
```

```
/* ... */
p = MALLOC(widget); /* OK */
if (p != NULL) {
    p->i = 0;           /* OK */
    p->d = 0.0;         /* OK */
}
```

Here, the entire allocation expression [to the right of the assignment operator] allocates storage for a `widget` and returns a `widget *`. If `p` were not a `widget *`, the compiler would complain about the assignment.

When allocating an array with `N` elements of type `T`, the appropriate type in the cast expression is still `T *`, but the argument to `malloc()` should be of the form `N * sizeof(T)`. Again, packaging this form as a macro, such as

```
#define MALLOC_ARRAY(number, type) \
    ((type *)malloc((number) * sizeof(type)))
```

reduces the chance of error in an allocation expression.

```
enum { N = 16 };
widget *p;

/* ... */

p = MALLOC_ARRAY(N, widget); /* OK */
```

A small collection of macros can provide secure implementations for common uses for the standard memory allocation functions. The omission of a `REALLOC()` macro is intentional [see [EXP39-C. Do not access a variable through a pointer of an incompatible type](#)].

```
/* Allocates a single object using malloc() */
#define MALLOC(type) ((type *)malloc(sizeof(type)))

/* Allocates an array of objects using malloc() */
#define MALLOC_ARRAY(number, type) \
    ((type *)malloc((number) * sizeof(type)))

/*
 * Allocates a single object with a flexible
 * array member using malloc().
 */
#define MALLOC_FLEX(stype, number, etype) \
    ((stype *)malloc(sizeof(stype) \
        + (number) * sizeof(etype)))

/* Allocates an array of objects using calloc() */
#define CALLOC(number, type) \
    ((type *)calloc(number, sizeof(type)))

/* Reallocates an array of objects using realloc() */
#define REALLOC_ARRAY(pointer, number, type) \
    ((type *)realloc(pointer, (number) * sizeof(type)))

/*
 * Reallocates a single object with a flexible
 * array member using realloc().
 */
#define REALLOC_FLEX(pointer, stype, number, etype) \
    ((stype *)realloc(pointer, sizeof(stype) \
        + (number) * sizeof(etype)))
```

The following is an example:

```
enum month { Jan, Feb, /* ... */ };
typedef enum month month;

typedef struct date date;
struct date {
    unsigned char dd;
    month mm;
    unsigned yy;
};

typedef struct string string;
struct string {
    size_t length;
    char text[];
};

date *d, *week, *fortnight;
string *name;

d = MALLOC(date);
week = MALLOC_ARRAY(7, date);
name = MALLOC_FLEX(string, 16, char);
fortnight = CALLOC(14, date);
```

If one or more of the operands to the multiplication operations used in many of these macro definitions can be influenced by untrusted data, these operands should be checked for overflow before the macro is invoked [see [INT32-C. Ensure that operations on signed integers do not result in overflow](#)].

The use of type-generic function-like macros is an allowed exception (PRE00-C-EX4) to [PRE00-C. Prefer inline or static functions to function-like macros](#).

Exceptions

MEM02-C-EX1: Do not immediately cast the results of `malloc()` for code that will be compiled using a C90-conforming compiler because it is possible for the cast to hide a more critical defect [see [DCL31-C. Declare identifiers before using them](#) for a code example that uses `malloc()` without first declaring it].

Failing to cast the result of a memory allocation function call into a pointer to the allocated type can result in inadvertent pointer conversions. Code that follows this recommendation will compile and execute equally well in C++.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
MEM02-C	Low	Unlikely	Low	P3	L3

Related Guidelines

SEI CERT C++ Coding Standard	MEM02-CPP. Immediately cast the result of a memory allocation function call into a pointer to the allocated type
--	--

Bibliography

[Summit 2005]	Question 7.7
	Question 7.7b

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/gwg>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
malloc_with_wrong_cast	Wrong cast on result of allocator, should cast to {} instead
malloc_without_cast	Missing cast to allocated type.

CertC-MEM30

Do not access freed memory.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Evaluating a pointer - including dereferencing the pointer, using it as an operand of an arithmetic operation, type casting it, and using it as the right-hand side of an assignment - into memory that has been deallocated by a memory management function is [undefined behavior](#). Pointers to memory that has been deallocated are called *dangling pointers*. Accessing a dangling pointer can result in exploitable [vulnerabilities](#).

According to the C Standard, using the value of a pointer that refers to space deallocated by a call to the `free()` or `realloc()` function is undefined behavior. (See [undefined behavior 177](#).)

Reading a pointer to deallocated memory is undefined behavior because the pointer value is [indeterminate](#) and might be a [trap representation](#). Fetching a trap representation might perform a hardware trap (but is not required to).

It is at the memory manager's discretion when to reallocate or recycle the freed memory. When memory is freed, all pointers into it become invalid, and its contents might either be returned to the operating system, making the freed space inaccessible, or remain intact and accessible. As a result, the data at the freed location can appear to be valid but change unexpectedly. Consequently, memory must not be written to or read from once it is freed.

Noncompliant Code Example

This example from Brian Kernighan and Dennis Ritchie [[Kernighan 1988](#)] shows both the incorrect and correct techniques for freeing the memory associated with a linked list. In their (intentionally) incorrect example, `p` is freed before `p->next` is executed, so that `p->next` reads memory that has already been freed.

```
#include <stdlib.h>
struct node {
```

```

int value;
struct node *next;
};

void free_list(struct node *head) {
    for (struct node *p = head; p != NULL; p = p->next) {
        free(p);
    }
}

```

Compliant Solution

Kernighan and Ritchie correct this error by storing a reference to `p->next` in `q` before freeing `p`:

```

#include <stdlib.h>

struct node {
    int value;
    struct node *next;
};

void free_list(struct node *head) {
    struct node *q;
    for (struct node *p = head; p != NULL; p = q) {
        q = p->next;
        free(p);
    }
}

```

Noncompliant Code Example

In this noncompliant code example, `buf` is written to after it has been freed. Write-after-free vulnerabilities can be [exploited](#) to run arbitrary code with the permissions of the vulnerable process. Typically, allocations and frees are far removed, making it difficult to recognize and diagnose these problems.

```

#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char *return_val = 0;
    const size_t bufsize = strlen(argv[0]) + 1;
    char *buf = (char *)malloc(bufsize);
    if (!buf) {
        return EXIT_FAILURE;
    }
    /* ... */
    free(buf);
    /* ... */
    strcpy(buf, argv[0]);
    /* ... */
    return EXIT_SUCCESS;
}

```

Compliant Solution

In this compliant solution, the memory is freed after its final use:

```

#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char *return_val = 0;
    const size_t bufsize = strlen(argv[0]) + 1;
    char *buf = (char *)malloc(bufsize);
    if (!buf) {
        return EXIT_FAILURE;
    }
    /* ... */
    strcpy(buf, argv[0]);
    /* ... */
    free(buf);
    return EXIT_SUCCESS;
}

```

Noncompliant Code Example

In this noncompliant example, `realloc()` may free `c_str1` when it returns a null pointer, resulting in `c_str1` being freed twice. The C Standards Committee's proposed response to [Defect Report #400](#) makes it implementation-defined whether or not the old object is deallocated when `size` is zero and memory for the new object is not allocated. The current implementation of `realloc()` in the GNU C Library and Microsoft Visual Studio's Runtime Library will free `c_str1` and return a null pointer for zero byte allocations. Freeing a pointer twice can result in a potentially exploitable vulnerability commonly referred to as a *double-free vulnerability* [[Seacord 2013b](#)].

```

#include <stdlib.h>

void f(char *c_str1, size_t size) {
    char *c_str2 = (char *)realloc(c_str1, size);
    if (c_str2 == NULL) {
        free(c_str1);
    }
}

```

Compliant Solution

This compliant solution does not pass a size argument of zero to the `realloc()` function, eliminating the possibility of `c_str1` being freed twice:

```
#include <stdlib.h>

void f(char *c_str1, size_t size) {
    if (size != 0) {
        char *c_str2 = (char *)realloc(c_str1, size);
        if (c_str2 == NULL) {
            free(c_str1);
        }
    } else {
        free(c_str1);
    }
}
```

If the intent of calling `f()` is to reduce the size of the object, then doing nothing when the size is zero would be unexpected; instead, this compliant solution frees the object.

Noncompliant Code Example

In this noncompliant example ([CVE-2009-1364](#)) from `libwmf` version 0.2.8.4, the return value of `gdRealloc` (a simple wrapper around `realloc()` that reallocates space pointed to by `im->clip->list`) is set to `more`. However, the value of `im->clip->list` is used directly afterwards in the code, and the C Standard specifies that if `realloc()` moves the area pointed to, then the original block is freed. An attacker can then execute arbitrary code by forcing a reallocation (with a sufficient `im->clip->count`) and accessing freed memory ([xorl 2009](#)).

```
void gdClipSetAdd(gdImagePtr im, gdClipRectanglePtr rect) {
    gdClipRectanglePtr more;
    if (im->clip == 0) {
        /* ... */
    }
    if (im->clip->count == im->clip->max) {
        more = gdRealloc (im->clip->list,(im->clip->max + 8) *
                         sizeof (gdClipRectangle));
        /*
         * If the realloc fails, then we have not lost the
         * im->clip->list value.
         */
        if (more == 0) return;
        im->clip->max += 8;
    }
    im->clip->list[im->clip->count] = *rect;
    im->clip->count++;
}
```

Compliant Solution

This compliant solution simply reassigns `im->clip->list` to the value of `more` after the call to `realloc()`:

```
void gdClipSetAdd(gdImagePtr im, gdClipRectanglePtr rect) {
    gdClipRectanglePtr more;
    if (im->clip == 0) {
        /* ... */
    }
    if (im->clip->count == im->clip->max) {
        more = gdRealloc (im->clip->list,(im->clip->max + 8) *
                         sizeof (gdClipRectangle));
        if (more == 0) return;
        im->clip->max += 8;
        im->clip->list = more;
    }
    im->clip->list[im->clip->count] = *rect;
    im->clip->count++;
}
```

Risk Assessment

Reading memory that has already been freed can lead to abnormal program termination and denial-of-service attacks. Writing memory that has already been freed can additionally lead to the execution of arbitrary code with the permissions of the vulnerable process.

Freeing memory multiple times has similar consequences to accessing memory after it is freed. Reading a pointer to deallocated memory is [undefined behavior](#) because the pointer value is [indeterminate](#) and might be a [trap representation](#). When reading from or writing to freed memory does not cause a trap, it may corrupt the underlying data structures that manage the heap in a manner that can be exploited to execute arbitrary code. Alternatively, writing to memory after it has been freed might modify memory that has been reallocated.

Programmers should be wary when freeing memory in a loop or conditional statement; if coded incorrectly, these constructs can lead to double-free vulnerabilities. It is also a common error to misuse the `realloc()` function in a manner that results in double-free vulnerabilities. (See [MEM04-C. Beware of zero-length allocations](#).)

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM30-C	High	Likely	Medium	P18	L1

Related Guidelines

CERT C Secure Coding Standard	MEM01-C. Store a new value in pointers immediately after free()
SEI CERT C++ Coding Standard	MEM50-CPP. Do not access freed memory
ISO/IEC TR 24772:2013	Dangling References to Stack Frames [DCM] Dangling Reference to Heap [XYK]
ISO/IEC TS 17961	Accessing freed memory [accfree] Freeing memory multiple times [dblfree]
MISRA C:2012	Rule 18.6 (required)
MITRE CWE	CWE-415 , Double Free CWE-416 , Use After Free

Bibliography

[ISO/IEC 9899:2011]	7.22.3, "Memory Management Functions"
[Kernighan 1988]	Section 7.8.5, "Storage Management"
[OWASP Freed Memory]	
[MIT 2005]	
[Seacord 2013b]	Chapter 4, "Dynamic Memory Management"
[Viega 2005]	Section 5.2.19, "Using Freed Memory"
[VU#623332]	
[xorl 2009]	CVE-2009-1364: LibWMF Pointer Use after free()

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki <https://www.securecoding.cert.org/confluence/x/vAE>, Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
resources	Configuration of memory types and allocate/release functions for them in addition to what <code>iranalysis.config</code> provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)

Possible Messages

Name	Message
double_free	Dynamic memory released here was already released earlier
possible_double_free	Dynamic memory released here possibly already released earlier
possible_use_after_free	Dynamic memory possibly used after it was previously released
use_after_free	Dynamic memory used after it was previously released

CertC-MEM31

Free dynamically allocated memory when no longer needed.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Before the lifetime of the last pointer that stores the return value of a call to a standard memory allocation function has ended, it must be matched by a call to `free()` with that pointer value.

Noncompliant Code Example

In this noncompliant example, the object allocated by the call to `malloc()` is not freed before the end of the lifetime of the last pointer `text_buffer` referring to the object:

```
#include <stdlib.h>

enum { BUFFER_SIZE = 32 };

int f(void) {
    char *text_buffer = (char *)malloc(BUFFER_SIZE);
    if (text_buffer == NULL) {
        return -1;
    }
    return 0;
}
```

Compliant Solution

In this compliant solution, the pointer is deallocated with a call to `free()`:

```
#include <stdlib.h>

enum { BUFFER_SIZE = 32 };

int f(void) {
    char *text_buffer = (char *)malloc(BUFFER_SIZE);
    if (text_buffer == NULL) {
        return -1;
    }

    free(text_buffer);
    return 0;
}
```

Exceptions

MEM31-C-EX1: Allocated memory does not need to be freed if it is assigned to a pointer with static storage duration whose lifetime is the entire execution of a program. The following code example illustrates a pointer that stores the return value from `malloc()` in a `static` variable:

```
#include <stdlib.h>

enum { BUFFER_SIZE = 32 };

int f(void) {
    static char *text_buffer = NULL;
    if (text_buffer == NULL) {
        text_buffer = (char *)malloc(BUFFER_SIZE);
        if (text_buffer == NULL) {
            return -1;
        }
    }
    return 0;
}
```

Risk Assessment

Failing to free memory can result in the exhaustion of system memory resources, which can lead to a [denial-of-service attack](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM31-C	Medium	Probable	Medium	P8	L2

Related Guidelines

ISO/IEC TR 24772:2013	Memory Leak [XYL]
ISO/IEC TS 17961	Failing to close files or free dynamic memory when they are no longer needed [fclose]
MITRE CWE	CWE-401 , Improper Release of Memory Before Removing Last Reference ("Memory Leak")

Bibliography

[ISO/IEC 9899:2011]	Subclause 7.22.3, "Memory Management Functions"
-------------------------------------	---

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/vQE>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)

Possible Messages

Name	Message
memory_leak	Call allocates leaking memory
possible_memory_leak	Call allocates possibly leaking memory

CertC-MEM33

Allocate and copy structures containing a flexible array member dynamically.

Input: IR

Source languages: C, C++

Details

The C Standard, 6.7.2.1, paragraph 18 [[ISO/IEC 9899:2011](#)], says

As a special case, the last element of a structure with more than one named member may have an incomplete array type; this is called a *flexible array member*. In most situations, the flexible array member is ignored. In particular, the size of the structure is as if the flexible array member were omitted except that it may have more trailing padding than the omission would imply.

The following is an example of a structure that contains a flexible array member:

```
struct flex_array_struct {
    int num;
    int data[];
};
```

This definition means that when computing the size of such a structure, only the first member, `num`, is considered. Unless the appropriate size of the flexible array member has been explicitly added when allocating storage for an object of the `struct`, the result of accessing the member `data` of a variable of nonpointer type `struct flex_array_struct` is [undefined](#). [DCL38-C. Use the correct syntax when declaring a flexible array member](#) describes the correct way to declare a `struct` with a flexible array member.

To avoid the potential for undefined behavior, structures that contain a flexible array member should always be allocated dynamically. Flexible array structures must

- Have dynamic storage duration (be allocated via `malloc()` or another dynamic allocation function)
- Be dynamically copied using `memcpy()` or a similar function and not by assignment
- When used as an argument to a function, be passed by pointer and not copied by value

Noncompliant Code Example (Storage Duration)

This noncompliant code example uses automatic storage for a structure containing a flexible array member:

```
#include <stddef.h>

struct flex_array_struct {
    size_t num;
    int data[];
};

void func(void) {
    struct flex_array_struct flex_struct;
    size_t array_size = 4;

    /* Initialize structure */
    flex_struct.num = array_size;

    for (size_t i = 0; i < array_size; ++i) {
        flex_struct.data[i] = 0;
    }
}
```

Because the memory for `flex_struct` is reserved on the stack, no space is reserved for the `data` member. Accessing the `data` member is [undefined behavior](#).

Compliant Solution {Storage Duration}

This compliant solution dynamically allocates storage for `flex_array_struct`:

```
#include <stdlib.h>

struct flex_array_struct {
    size_t num;
    int data[];
};

void func(void) {
    struct flex_array_struct *flex_struct;
    size_t array_size = 4;

    /* Dynamically allocate memory for the struct */
    flex_struct = (struct flex_array_struct *)malloc(
        sizeof(struct flex_array_struct)
        + sizeof(int) * array_size);
    if (flex_struct == NULL) {
        /* Handle error */
    }

    /* Initialize structure */
    flex_struct->num = array_size;

    for (size_t i = 0; i < array_size; ++i) {
        flex_struct->data[i] = 0;
    }
}
```

Noncompliant Code Example {Copying}

This noncompliant code example attempts to copy an instance of a structure containing a flexible array member (`struct flex_array_struct`) by assignment:

```
#include <stddef.h>

struct flex_array_struct {
    size_t num;
    int data[];
};

void func(struct flex_array_struct *struct_a,
          struct flex_array_struct *struct_b) {
    *struct_b = *struct_a;
}
```

When the structure is copied, the size of the flexible array member is not considered, and only the first member of the structure, `num`, is copied, leaving the array contents untouched.

Compliant Solution {Copying}

This compliant solution uses `memcpy()` to properly copy the content of `struct_a` into `struct_b`:

```
#include <string.h>

struct flex_array_struct {
    size_t num;
    int data[];
};

void func(struct flex_array_struct *struct_a,
          struct flex_array_struct *struct_b) {
    if (struct_a->num > struct_b->num) {
        /* Insufficient space; handle error */
        return;
    }
    memcpy(struct_b, struct_a,
           sizeof(struct flex_array_struct) + (sizeof(int)
               * struct_a->num));
}
```

Noncompliant Code Example {Function Arguments}

In this noncompliant code example, the flexible array structure is passed by value to a function that prints the array elements:

```
#include <stdio.h>
#include <stdlib.h>

struct flex_array_struct {
    size_t num;
    int data[];
};

void print_array(struct flex_array_struct struct_p) {
    puts("Array is: ");
    for (size_t i = 0; i < struct_p.num; ++i) {
        printf("%d ", struct_p.data[i]);
    }
    putchar('\n');
}
```

```

void func(void) {
    struct flex_array_struct *struct_p;
    size_t array_size = 4;

    /* Space is allocated for the struct */
    struct_p = (struct flex_array_struct *)malloc(
        sizeof(struct flex_array_struct)
        + sizeof(int) * array_size);
    if (struct_p == NULL) {
        /* Handle error */
    }
    struct_p->num = array_size;

    for (size_t i = 0; i < array_size; ++i) {
        struct_p->data[i] = i;
    }
    print_array(*struct_p);
}

```

Because the argument is passed by value, the size of the flexible array member is not considered when the structure is copied, and only the first member of the structure, `num`, is copied.

Compliant Solution {Function Arguments}

In this compliant solution, the structure is passed by reference and not by value:

```

#include <stdio.h>
#include <stdlib.h>

struct flex_array_struct {
    size_t num;
    int data[];
};

void print_array(struct flex_array_struct *struct_p) {
    puts("Array is: ");
    for (size_t i = 0; i < struct_p->num; ++i) {
        printf("%d ", struct_p->data[i]);
    }
    putchar('\n');
}

void func(void) {
    struct flex_array_struct *struct_p;
    size_t array_size = 4;

    /* Space is allocated for the struct and initialized... */

    print_array(struct_p);
}

```

Risk Assessment

Failure to use structures with flexible array members correctly can result in [undefined behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM33-C	Low	Unlikely	Low	P3	L3

Related Guidelines

CERT C Secure Coding Standard	DCL38-C. Use the correct syntax when declaring a flexible array member
---	--

Bibliography

[ISO/IEC 9899:2011]	Subclause 6.7.2.1, "Structure and Union Specifiers"
[JTC1/SC22/WG14 N791]	Solving the Struct Hack Problem

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/6AAU>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
alloc	Allocate structures containing a flexible array member dynamically.
copy	Assignment does not copy the flexible array member. Use memcpy() instead.

CertC-MEM34

Only free memory allocated dynamically.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

The C Standard, Annex J [[ISO/IEC 9899:2011](#)], states that the behavior of a program is undefined when

The pointer argument to the `free` or `realloc` function does not match a pointer earlier returned by a memory management function, or the space has been deallocated by a call to `free` or `realloc`.

See also [undefined behavior 179](#).

Freeing memory that is not allocated dynamically can result in heap corruption and other serious errors. Do not call `free()` on a pointer other than one returned by a standard memory allocation function, such as `malloc()`, `calloc()`, `realloc()`, or `aligned_alloc()`.

A similar situation arises when `realloc()` is supplied a pointer to non-dynamically allocated memory. The `realloc()` function is used to resize a block of dynamic memory. If `realloc()` is supplied a pointer to memory not allocated by a standard memory allocation function, the behavior is undefined. One consequence is that the program may [terminate abnormally](#).

This rule does not apply to null pointers. The C Standard guarantees that if `free()` is passed a null pointer, no action occurs.

Noncompliant Code Example

This noncompliant code example sets `c_str` to reference either dynamically allocated memory or a statically allocated string literal depending on the value of `argc`. In either case, `c_str` is passed as an argument to `free()`. If anything other than dynamically allocated memory is referenced by `c_str`, the call to `free(c_str)` is erroneous.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum { MAX_ALLOCATION = 1000 };

int main(int argc, const char *argv[]) {
    char *c_str = NULL;
    size_t len;

    if (argc == 2) {
        len = strlen(argv[1]) + 1;
        if (len > MAX_ALLOCATION) {
            /* Handle error */
        }
        c_str = (char *)malloc(len);
        if (c_str == NULL) {
            /* Handle error */
        }
        strcpy(c_str, argv[1]);
    } else {
        c_str = "usage: $>a.exe [string]";
        printf("%s\n", c_str);
    }
    free(c_str);
    return 0;
}
```

Compliant Solution

This compliant solution eliminates the possibility of `c_str` referencing memory that is not allocated dynamically when passed to `free()`:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum { MAX_ALLOCATION = 1000 };

int main(int argc, const char *argv[]) {
    char *c_str = NULL;
    size_t len;

    if (argc == 2) {
        len = strlen(argv[1]) + 1;
        if (len > MAX_ALLOCATION) {
            /* Handle error */
        }
        c_str = (char *)malloc(len);
        if (c_str == NULL) {
            /* Handle error */
        }
        strcpy(c_str, argv[1]);
    } else {
        printf("%s\n", "usage: $>a.exe [string]");
        return EXIT_FAILURE;
    }
    free(c_str);
    return 0;
}
```

Noncompliant Code Example (`realloc()`)

In this noncompliant example, the pointer parameter to `realloc()`, `buf`, does not refer to dynamically allocated memory:

```
#include <stdlib.h>

enum { BUFSIZE = 256 };

void f(void) {
    char buf[BUFSIZE];
    char *p = (char *)realloc(buf, 2 * BUFSIZE);
    if (p == NULL) {
        /* Handle error */
    }
}
```

Compliant Solution (`realloc()`)

In this compliant solution, `buf` refers to dynamically allocated memory:

```
#include <stdlib.h>

enum { BUFSIZE = 256 };

void f(void) {
    char *buf = (char *)malloc(BUFSIZE * sizeof(char));
    char *p = (char *)realloc(buf, 2 * BUFSIZE);
    if (p == NULL) {
        /* Handle error */
    }
}
```

Note that `realloc()` will behave properly even if `malloc()` failed, because when given a null pointer, `realloc()` behaves like a call to `malloc()`.

Risk Assessment

The consequences of this error depend on the [implementation](#), but they range from nothing to arbitrary code execution if that memory is reused by `malloc()`.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM34-C	High	Likely	Medium	P18	L1

Related Guidelines

CERT C Secure Coding Standard	MEM31-C. Free dynamically allocated memory when no longer needed
SEI CERT C++ Coding Standard	MEM51-CPP. Properly deallocate dynamically allocated resources
ISO/IEC TS 17961	Reallocating or freeing memory that was not dynamically allocated [xfree]
MITRE CWE	CWE-590, Free of Memory Not on the Heap

Bibliography

[ISO/IEC 9899:2011]	Subclause J.2, "Undefined Behavior"
[Seacord 2013b]	Chapter 4, "Dynamic Memory Management"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)

Possible Messages

Name	Message
possible_stack_free	{} possibly released by call to {} is a stack object
stack_free	{} released by call to {} is a stack object

CertC-MEM35

Allocate sufficient memory for an object.

Input: IR

Source languages: C, C++

Details

The types of integer expressions used as size arguments to `malloc()`, `calloc()`, `realloc()`, or `aligned_alloc()` must have sufficient range to represent the size of the objects to be stored. If size arguments are incorrect or can be manipulated by an attacker, then a buffer overflow may occur. Incorrect size arguments, inadequate range checking, integer overflow, or truncation can result in the allocation of an inadequately sized buffer.

Typically, the amount of memory to allocate will be the size of the type of object to allocate. When allocating space for an array, the size of the object will be multiplied by the bounds of the array. When allocating space for a structure containing a flexible array member, the size of the array member must be added to the size of the structure. (See [MEM33-C. Allocate and copy structures containing a flexible array member dynamically](#).) Use the correct type of the object when computing the size of memory to allocate.

[STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator](#) is a specific instance of this rule.

Noncompliant Code Example (Pointer)

In this noncompliant code example, inadequate space is allocated for a `struct tm` object because the size of the pointer is being used to determine the size of the pointed-to object:

```
#include <stdlib.h>
#include <time.h>

struct tm *make_tm(int year, int mon, int day, int hour,
                   int min, int sec) {
    struct tm *tmb;
    tmb = (struct tm *)malloc(sizeof(tmb));
    if (tmb == NULL) {
        return NULL;
    }
    *tmb = (struct tm) {
        .tm_sec = sec, .tm_min = min, .tm_hour = hour,
        .tm_mday = day, .tm_mon = mon, .tm_year = year
    };
    return tmb;
}
```

Compliant Solution (Pointer)

In this compliant solution, the correct amount of memory is allocated for the `struct tm` object. When allocating space for a single object, passing the (dereferenced) pointer type to the `sizeof` operator is a simple way to allocate sufficient memory. Because the `sizeof` operator does not evaluate its operand, dereferencing an uninitialized or null pointer in this context is well-defined behavior.

```
#include <stdlib.h>
#include <time.h>

struct tm *make_tm(int year, int mon, int day, int hour,
                   int min, int sec) {
    struct tm *tmb;
    tmb = (struct tm *)malloc(sizeof(*tmb));
```

```

if (tmb == NULL) {
    return NULL;
}
*tmb = (struct tm) {
    .tm_sec = sec, .tm_min = min, .tm_hour = hour,
    .tm_mday = day, .tm_mon = mon, .tm_year = year
};
return tmb;
}

```

Noncompliant Code Example (Integer)

In this noncompliant code example, an array of `long` is allocated and assigned to `p`. The code attempts to check for unsigned integer overflow in compliance with [INT30-C. Ensure that unsigned integer operations do not wrap](#) and also ensures that `len` is not equal to zero. (See [MEM04-C. Beware of zero-length allocations](#).) However, because `sizeof(int)` is used to compute the size, and not `sizeof(long)`, an insufficient amount of memory can be allocated on implementations where `sizeof(long)` is larger than `sizeof(int)`, and filling the array can cause a heap buffer overflow.

```

#include <stdint.h>
#include <stdlib.h>

void function(size_t len) {
    long *p;
    if (len == 0 || len > SIZE_MAX / sizeof(long)) {
        /* Handle overflow */
    }
    p = (long *)malloc(len * sizeof(int));
    if (p == NULL) {
        /* Handle error */
    }
    free(p);
}

```

Compliant Solution (Integer)

This compliant solution uses `sizeof(long)` to correctly size the memory allocation:

```

#include <stdint.h>
#include <stdlib.h>

void function(size_t len) {
    long *p;
    if (len == 0 || len > SIZE_MAX / sizeof(long)) {
        /* Handle overflow */
    }
    p = (long *)malloc(len * sizeof(long));
    if (p == NULL) {
        /* Handle error */
    }
    free(p);
}

```

Compliant Solution (Integer)

Alternatively, `sizeof(*p)` can be used to properly size the allocation:

```

#include <stdint.h>
#include <stdlib.h>

void function(size_t len) {
    long *p;
    if (len == 0 || len > SIZE_MAX / sizeof(*p)) {
        /* Handle overflow */
    }
    p = (long *)malloc(len * sizeof(*p));
    if (p == NULL) {
        /* Handle error */
    }
    free(p);
}

```

Risk Assessment

Providing invalid size arguments to memory allocation functions can lead to buffer overflows and the execution of arbitrary code with the permissions of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM35-C	High	Probable	High	P6	L2

Related Guidelines

CERT C Secure Coding Standard	ARR01-C. Do not apply the sizeof operator to a pointer when taking the size of an array INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data INT32-C. Ensure that operations on signed integers do not result in overflow INT18-C. Evaluate integer expressions in a larger size before comparing or assigning to that size MEM04-C. Beware of zero-length allocations
ISO/IEC TR 24772:2013	Buffer Boundary Violation [Buffer Overflow] [HCB]
ISO/IEC TS 17961:2013	Taking the size of a pointer to determine the size of the pointed-to type [sizeofptr]
MITRE CWE	CWE-131 , Incorrect Calculation of Buffer Size CWE-190 , Integer Overflow or Wraparound CWE-467 , Use of <code>sizeof()</code> on a Pointer Type

Bibliography

[Coverity 2007]	
[Drepper 2006]	Section 2.1.1, "Respecting Memory Bounds"
[Seacord 2013]	Chapter 4, "Dynamic Memory Management" Chapter 5, "Integer Security"
[Viega 2005]	Section 5.6.8, "Use of <code>sizeof()</code> on a Pointer Type"
[xorl 2009]	CVE-2009-0587: Evolution Data Server Base64 Integer Overflows

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/2wE>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
wrong_malloc_size	Wrong allocation size: <code>sizeof</code> on {} used, but {} expected

CertC-MEM36

Do not modify the alignment of objects by calling `realloc()`.

Input: IR

Source languages: C, C++

Note: Rule requires `CONFIG.Run_IRAnalysis_Checks = True`.

Details

Do not invoke `realloc()` to modify the size of allocated objects that have stricter alignment requirements than those guaranteed by `malloc()`. Storage allocated by a call to the standard `aligned_alloc()` function, for example, can have stricter than normal alignment requirements. The C standard requires only that a pointer returned by `realloc()` be suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement.

Noncompliant Code Example

This noncompliant code example returns a pointer to allocated memory that has been aligned to a 4096-byte boundary. If the `resize` argument to the `realloc()` function is larger than the object referenced by `ptr`, then `realloc()` will allocate new memory that is suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement but may not preserve the stricter alignment of the original object.

```
#include <stdlib.h>

void func(void) {
    size_t resize = 1024;
    size_t alignment = 1 << 12;
    int *ptr;
    int *ptr1;

    if (NULL == (ptr = (int *)aligned_alloc(alignment, sizeof(int)))) {
        /* Handle error */
    }

    if (NULL == (ptr1 = (int *)realloc(ptr, resize))) {
        /* Handle error */
    }
}
```

Implementation Details

When compiled with GCC 4.1.2 and run on the x86_64 Red Hat Linux platform, the following code produces the following output:

CODE

```
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    size_t size = 16;
    size_t resize = 1024;
    size_t align = 1 << 12;
    int *ptr;
    int *ptr1;

    if (posix_memalign((void **)&ptr, align, size) != 0) {
        exit(EXIT_FAILURE);
    }

    printf("memory aligned to %zu bytes\n", align);
    printf("ptr = %p\n\n", ptr);

    if ((ptr1 = (int *) realloc((int *)ptr, resize)) == NULL) {
        exit(EXIT_FAILURE);
    }

    puts("After realloc():\n");
    printf("ptr1 = %p\n", ptr1);

    free(ptr1);
    return 0;
}
```

OUTPUT

```
memory aligned to 4096 bytes
ptr = 0x1621b000

After realloc():
ptr1 = 0x1621a010
```

`ptr1` is no longer aligned to 4096 bytes.

Compliant Solution

This compliant solution allocates `resize` bytes of new memory with the same alignment as the old memory, copies the original memory content, and then frees the old memory. This solution has [implementation-defined behavior](#) because it depends on whether extended alignments in excess of `_Alignof(max_align_t)` are supported and the contexts in which they are supported. If not supported, the behavior of this compliant solution is undefined.

```
#include <stdlib.h>
#include <string.h>

void func(void) {
    size_t resize = 1024;
    size_t alignment = 1 << 12;
    int *ptr;
    int *ptr1;

    if (NULL == (ptr = (int *)aligned_alloc(alignment,
                                             sizeof(int)))) {
        /* Handle error */
    }

    if (NULL == (ptr1 = (int *)aligned_alloc(alignment,
                                              resize))) {
        /* Handle error */
    }

    if (NULL == (memcpy(ptr1, ptr, sizeof(int)))) {
        /* Handle error */
    }

    free(ptr);
}
```

Compliant Solution (Windows)

Windows defines the `_aligned_malloc()` function to allocate memory on a specified alignment boundary. The `_aligned_realloc()` [MSDN] can be used to change the size of this memory. This compliant solution demonstrates one such usage:

```
#include <malloc.h>

void func(void) {
    size_t alignment = 1 << 12;
    int *ptr;
    int *ptr1;

    /* Original allocation */
    if (NULL == (ptr = (int *)_aligned_malloc(sizeof(int),
                                                alignment))) {
        /* Handle error */
    }

    /* Reallocation */
    if (NULL == (ptr1 = (int *)_aligned_realloc(ptr, 1024,
                                                alignment))) {
        _aligned_free(ptr);
        /* Handle error */
    }

    _aligned_free(ptr1);
}
```

The `size` and `alignment` arguments for `_aligned_malloc()` are provided in reverse order of the C Standard `aligned_alloc()` function.

Risk Assessment

Improper alignment can lead to arbitrary memory locations being accessed and written to.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
MEM36-C	Low	Probable	High	P2	L3

Bibliography

[ISO/IEC 9899:2011]	7.22.3.1, "The <code>aligned_alloc</code> Function"
[MSDN]	<code>aligned_malloc()</code>

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/4YEzAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
<code>aligned_allocators</code>	Functions that allocate aligned memory.	<code>('aligned_alloc', '_aligned_malloc', 'posix_memalign')</code>
<code>level</code>	Grouping of priorities into different levels	3
<code>likelihood</code>	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
<code>priority</code>	Priority based on the combination of severity, likelihood and remediation cost	2
<code>realloc_functions</code>	Functions that reallocate without allowing for aligned memory. The first argument must be the pointer to reallocate.	<code>('realloc',)</code>
<code>recommendation</code>	Whether this check is classified as a recommendation or rule	False
<code>remediation_cost</code>	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
<code>realloc_aligned</code>	Do not modify the alignment of objects by calling <code>realloc()</code> .

CertC-ENV30

Do not modify the object referenced by the return value of certain functions.

Input: IR

Source languages: C, C++

Details

Some functions return a pointer to an object that cannot be modified without causing [undefined behavior](#). These functions include `getenv()`, `setlocale()`, `localeconv()`, `asctime()`, and `strerror()`. In such cases, the function call results must be treated as being `const`-qualified.

The C Standard, 7.22.4.6, paragraph 4 [[ISO/IEC 9899:2011](#)], defines `getenv()` as follows:

The `getenv` function returns a pointer to a string associated with the matched list member. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `getenv` function. If the specified name cannot be found, a null pointer is returned.

If the string returned by `getenv()` must be altered, a local copy should be created. Altering the string returned by `getenv()` is [undefined behavior](#). (See [undefined behavior 184](#).)

Similarly, subclause 7.11.1.1, paragraph 8 [[ISO/IEC 9899:2011](#)], defines `setlocale()` as follows:

The pointer to string returned by the `setlocale` function is such that a subsequent call with that string value and its associated category will restore that part of the program's locale. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `setlocale` function.

And subclause 7.11.2.1, paragraph 8 [[ISO/IEC 9899:2011](#)], defines `localeconv()` as follows:

The `localeconv` function returns a pointer to the filled-in object. The structure pointed to by the return value shall not be modified by the program, but may be overwritten by a subsequent call to the `localeconv` function. In addition, calls to the `setlocale` function with categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC` may overwrite the contents of the structure.

Altering the string returned by `setlocale()` or the structure returned by `localeconv()` are [undefined behaviors](#). (See [undefined behaviors 120](#) and [121](#).) Furthermore, the C Standard imposes no requirements on the contents of the string by `setlocale()`. Consequently, no assumptions can be made as to the string's internal contents or structure.

Finally, subclause 7.24.6.2, paragraph 4 [[ISO/IEC 9899:2011](#)], states

The `strerror` function returns a pointer to the string, the contents of which are locale-specific. The array pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `strerror` function.

Altering the string returned by `strerror()` is [undefined behavior](#). (See [undefined behavior 184](#).)

Noncompliant Code Example (`getenv()`)

This noncompliant code example modifies the string returned by `getenv()` by replacing all double quotation marks ("") with underscores (_):

```
#include <stdlib.h>

void trstr(char *c_str, char orig, char rep) {
    while (*c_str != '\0') {
        if (*c_str == orig) {
            *c_str = rep;
        }
        ++c_str;
    }
}

void func(void) {
    char *env = getenv("TEST_ENV");
    if (env == NULL) {
        /* Handle error */
    }
    trstr(env, "'", '_');
}
```

Compliant Solution (`getenv()`) (Environment Not Modified)

If the programmer does not intend to modify the environment, this compliant solution demonstrates how to modify a copy of the return value:

```
#include <stdlib.h>
#include <string.h>

void trstr(char *c_str, char orig, char rep) {
    while (*c_str != '\0') {
        if (*c_str == orig) {
            *c_str = rep;
        }
        ++c_str;
    }
}

void func(void) {
    const char *env;
    char *copy_of_env;

    env = getenv("TEST_ENV");
    if (env == NULL) {
        /* Handle error */
    }

    copy_of_env = (char *)malloc(strlen(env) + 1);
    if (copy_of_env == NULL) {
```

```

    /* Handle error */
}

strcpy(copy_of_env, env);
trstr(copy_of_env, "'", '_');
/* ... */
free(copy_of_env);
}

```

Compliant Solution {getenv()} {Modifying the Environment in POSIX}

If the programmer's intent is to modify the environment, this compliant solution, which saves the altered string back into the environment by using the POSIX `setenv()` and `strdup()` functions, can be used:

```

#include <stdlib.h>
#include <string.h>

void trstr(char *c_str, char orig, char rep) {
    while (*c_str != '\0') {
        if (*c_str == orig) {
            *c_str = rep;
        }
        ++c_str;
    }
}

void func(void) {
    const char *env;
    char *copy_of_env;

    env = getenv("TEST_ENV");
    if (env == NULL) {
        /* Handle error */
    }

    copy_of_env = strdup(env);
    if (copy_of_env == NULL) {
        /* Handle error */
    }

    trstr(copy_of_env, "'", '_');

    if (setenv("TEST_ENV", copy_of_env, 1) != 0) {
        /* Handle error */
    }
    /* ... */
    free(copy_of_env);
}

```

Noncompliant Code Example {localeconv()}

In this noncompliant example, the object returned by `localeconv()` is directly modified:

```

#include <locale.h>

void f2(void) {
    struct lconv *conv = localeconv();

    if ('\0' == conv->decimal_point[0]) {
        conv->decimal_point = ".";
    }
}

```

Compliant Solution {localeconv()} {Copy}

This compliant solution modifies a copy of the object returned by `localeconv()`:

```

#include <locale.h>
#include <stdlib.h>
#include <string.h>

void f2(void) {
    const struct lconv *conv = localeconv();
    if (conv == NULL) {
        /* Handle error */
    }

    struct lconv *copy_of_conv = (struct lconv *)malloc(
        sizeof(struct lconv));
    if (copy_of_conv == NULL) {
        /* Handle error */
    }

    memcpy(copy_of_conv, conv, sizeof(struct lconv));

    if ('\0' == copy_of_conv->decimal_point[0]) {
        copy_of_conv->decimal_point = ".";
    }
    /* ... */
    free(copy_of_conv);
}

```

Modifying the object pointed to by the return value of `getenv()`, `setlocale()`, `localeconv()`, `asctime()`, or `strerror()` is [undefined behavior](#). Even if the modification succeeds, the modified object can be overwritten by a subsequent call to the same function.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ENV30-C	Low	Probable	Medium	P4	L3

Related Guidelines

ISO/IEC TS 17961:2013	Modifying the string returned by <code>getenv</code> , <code>localeconv</code> , <code>setlocale</code> , and <code>strerror</code> [libmod]
---------------------------------------	--

Bibliography

[IEEE Std 1003.1:2013]	XSH, System Interfaces, <code>getenv</code> XSH, System Interfaces, <code>setlocale</code> XSH, System Interfaces, <code>localeconv</code>
[ISO/IEC 9899:2011]	7.11.1.1, "The <code>setlocale</code> Function" 7.11.2.1, "The <code>localeconv</code> Function" 7.22.4.6, "The <code>getenv</code> Function" 7.24.6.2, "The <code>strerror</code> Function"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/XgA1>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
funcs		{'getenv', 'setlocale', 'localeconv', 'asctime', 'strerror'}
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
nonconst_system_pointer_retrievals	Return value should be assigned to a pointer to const-qualified type.
string_of_system_pointer_modified	Return value of call to system function should be considered const, including strings referenced by it

CertC-ENV32

All exit handlers must return normally.

Input: IR

Source languages: C, C++

Details

The C Standard provides three functions that cause an application to terminate normally: `_Exit()`, `exit()`, and `quick_exit()`. These are collectively called *exit functions*. When the `exit()` function is called, or control transfers out of the `main()` entry point function, functions registered with `atexit()` are called (but not `at_quick_exit()`). When the `quick_exit()` function is called, functions registered with `at_quick_exit()` (but not `atexit()`) are called. These functions are collectively called *exit handlers*. When the `_Exit()` function is called, no exit handlers or signal handlers are called.

Exit handlers must terminate by returning. It is important and potentially safety-critical for all exit handlers to be allowed to perform their cleanup actions. This is particularly true because the application programmer does not always know about handlers that may have been installed by support libraries. Two specific issues include nested calls to an exit function and terminating a call to an exit handler by invoking `longjmp`.

A nested call to an exit function is [undefined behavior](#). (See [undefined behavior 182](#).) This behavior can occur only when an exit function is invoked from an exit handler or when an exit function is called from within a signal handler. (See [SIG30-C. Call only asynchronous-safe functions within signal handlers](#).)

If a call to the `longjmp()` function is made that would terminate the call to a function registered with `atexit()`, the behavior is [undefined](#).

Noncompliant Code Example

In this noncompliant code example, the `exit1()` and `exit2()` functions are registered by `atexit()` to perform required cleanup upon program termination. However, if `some_condition` evaluates to true, `exit()` is called a second time, resulting in [undefined behavior](#).

```
#include <stdlib.h>

void exit1(void) {
    /* ... Cleanup code ... */
    return;
}

void exit2(void) {
    extern int some_condition;
    if (some_condition) {
        /* ... More cleanup code ... */
        exit(0);
    }
    return;
}

int main(void) {
    if (atexit(exit1) != 0) {
        /* Handle error */
    }
    if (atexit(exit2) != 0) {
        /* Handle error */
    }
    /* ... Program code ... */
    return 0;
}
```

Functions registered by the `atexit()` function are called in the reverse order from which they were registered. Consequently, if `exit2()` exits in any way other than by returning, `exit1()` will not be executed. The same may also be true for `atexit()` handlers installed by support libraries.

Compliant Solution

A function that is registered as an exit handler by `atexit()` must exit by returning, as in this compliant solution:

```
#include <stdlib.h>

void exit1(void) {
    /* ... Cleanup code ... */
    return;
}

void exit2(void) {
    extern int some_condition;
    if (some_condition) {
        /* ... More cleanup code ... */
    }
    return;
}

int main(void) {
    if (atexit(exit1) != 0) {
        /* Handle error */
    }
    if (atexit(exit2) != 0) {
        /* Handle error */
    }
    /* ... Program code ... */
    return 0;
}
```

Noncompliant Code Example

In this noncompliant code example, `exit1()` is registered by `atexit()` so that upon program termination, `exit1()` is called. The `exit1()` function jumps back to `main()` to return, with undefined results.

```
#include <stdlib.h>
#include <setjmp.h>

jmp_buf env;
int val;

void exit1(void) {
    longjmp(env, 1);
}

int main(void) {
    if (atexit(exit1) != 0) {
        /* Handle error */
    }
    if (setjmp(env) == 0) {
        exit(0);
    } else {
        return 0;
    }
}
```

Compliant Solution

This compliant solution does not call `longjmp()` but instead returns from the exit handler normally:

```
#include <stdlib.h>

void exit1(void) {
    return;
}

int main(void) {
    if (atexit(exit1) != 0) {
        /* Handle error */
    }
    return 0;
}
```

Risk Assessment

Terminating a call to an exit handler in any way other than by returning is [undefined behavior](#) and may result in [abnormal program termination](#) or other unpredictable behavior. It may also prevent other registered handlers from being invoked.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ENV32-C	Medium	Likely	Medium	P12	L1

Related Guidelines

CERT C Secure Coding Standard	SIG30-C. Call only asynchronous-safe functions within signal handlers
ISO/IEC TR 24772:2013	Structured Programming [EWD] Termination Strategy [REU]
MITRE CWE	CWE-705 , Incorrect Control Flow Scoping

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/voAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
exit_functions	Functions which should not be used in exit handlers.	['exit', 'quick_exit', 'longjmp']
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
non_normal_return_exit_handler	All exit handlers must return normally.

CertC-ENV33

Do not call `system()`.

Input: IR

Source languages: C, C++

Details

The C Standard `system()` function executes a specified command by invoking an [implementation-defined](#) command processor, such as a UNIX shell or `CMD.EXE` in Microsoft Windows. The POSIX `popen()` and Windows `_popen()` functions also invoke a command processor but create a pipe between the calling program and the executed command, returning a pointer to a stream that can be used to either read from or write to the pipe [[IEEE Std 1003.1:2013](#)].

Use of the `system()` function can result in exploitable [vulnerabilities](#), in the worst case allowing execution of arbitrary system commands. Situations in which calls to `system()` have high risk include the following:

- When passing an unsanitized or improperly sanitized command string originating from a tainted source
- If a command is specified without a path name and the command processor path name resolution mechanism is accessible to an attacker

- If a relative path to an executable is specified and control over the current working directory is accessible to an attacker
- If the specified executable program can be spoofed by an attacker

Do not invoke a command processor via `system()` or equivalent functions to execute a command.

Noncompliant Code Example

In this noncompliant code example, the `system()` function is used to execute `any_cmd` in the host environment. Invocation of a command processor is not required.

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

enum { BUFFERSIZE = 512 };

void func(const char *input) {
    char cmdbuf[BUFFERSIZE];
    int len_wanted = snprintf(cmdbuf, BUFFERSIZE,
        "any_cmd '%s'", input);
    if (len_wanted >= BUFFERSIZE) {
        /* Handle error */
    } else if (len_wanted < 0) {
        /* Handle error */
    } else if (system(cmdbuf) == -1) {
        /* Handle error */
    }
}
```

If this code is compiled and run with elevated privileges on a Linux system, for example, an attacker can create an account by entering the following string:

```
'happy'; useradd 'attacker'
```

The shell would interpret this string as two separate commands:

```
any_cmd 'happy';
useradd 'attacker'
```

and create a new user account that the attacker can use to access the compromised system.

This noncompliant code example also violates [STR02-C. Sanitize data passed to complex subsystems](#).

Compliant Solution (POSIX)

In this compliant solution, the call to `system()` is replaced with a call to `execve()`. The `exec` family of functions does not use a full shell interpreter, so it is not vulnerable to command-injection attacks, such as the one illustrated in the noncompliant code example.

The `exec1()`, `execle()`, `execv()`, and `execve()` functions duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a forward slash character (/). As a result, they should be used without a forward slash character (/) only if the `PATH` environment variable is set to a safe value, as described in [ENV03-C. Sanitize the environment when invoking external programs](#).

The `exec1()`, `execle()`, `execv()`, and `execve()` functions do not perform path name substitution.

Additionally, precautions should be taken to ensure the external executable cannot be modified by an untrusted user, for example, by ensuring the executable is not writable by the user.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>

void func(char *input) {
    pid_t pid;
    int status;
    pid_t ret;
    char *const args[3] = {"any_exe", input, NULL};
    char **env;
    extern char **environ;

    /* ... Sanitize arguments ... */

    pid = fork();
    if (pid == -1) {
        /* Handle error */
    } else if (pid != 0) {
        while ((ret = waitpid(pid, &status, 0)) == -1) {
            if (errno != EINTR) {
                /* Handle error */
                break;
            }
        }
        if ((ret != -1) &&
            (!WIFEXITED(status) || !WEXITSTATUS(status)) ) {
            /* Report unexpected child status */
        }
    } else {
        /* ... Initialize env as a sanitized copy of environ ... */
        if (execve("/usr/bin/any_cmd", args, env) == -1) {
            /* Handle error */
            _Exit(127);
        }
    }
}
```

```
    }
}
```

This compliant solution is significantly different from the preceding noncompliant code example. First, `input` is incorporated into the `args` array and passed as an argument to `execve()`, eliminating concerns about buffer overflow or string truncation while forming the command string. Second, this compliant solution forks a new process before executing `"/usr/bin/any_cmd"` in the child process. Although this method is more complicated than calling `system()`, the added security is worth the additional effort.

The exit status of 127 is the value set by the shell when a command is not found, and POSIX recommends that applications should do the same. XCU, Section 2.8.2, of *Standard for Information Technology-Portable Operating System Interface (POSIX®), Base Specifications, Issue 7*[[IEEE Std 1003.1:2013](#)], says

If a command is not found, the exit status shall be 127. If the command name is found, but it is not an executable utility, the exit status shall be 126. Applications that invoke utilities without using the shell should use these exit status values to report similar errors.

Compliant Solution (Windows)

This compliant solution uses the Microsoft Windows [CreateProcess\(\)](#) API:

```
#include <Windows.h>

void func(TCHAR *input) {
    STARTUPINFO si = { 0 };
    PROCESS_INFORMATION pi;
    si.cb = sizeof(si);
    if (!CreateProcess(TEXT("any_cmd.exe"), input, NULL, NULL, FALSE,
                      0, 0, 0, &si, &pi)) {
        /* Handle error */
    }
    CloseHandle(pi.hThread);
    CloseHandle(pi.hProcess);
}
```

This compliant solution relies on the `input` parameter being `non-const`. If it were `const`, the solution would need to create a copy of the parameter because the `CreateProcess()` function can modify the command-line arguments to be passed into the newly created process.

This solution creates the process such that the child process does not inherit any handles from the parent process, in compliance with [WIN03-C. Understand HANDLE inheritance](#).

Noncompliant Code Example (POSIX)

This noncompliant code invokes the C `system()` function to remove the `.config` file in the user's home directory.

```
#include <stdlib.h>

void func(void) {
    system("rm ~/.config");
}
```

If the vulnerable program has elevated privileges, an attacker can manipulate the value of the `HOME` environment variable such that this program can remove any file named `.config` anywhere on the system.

Compliant Solution (POSIX)

An alternative to invoking the `system()` call to execute an external program to perform a required operation is to implement the functionality directly in the program using existing library calls. This compliant solution calls the POSIX [unlink\(\)](#) function to remove a file without invoking the `system()` function [[IEEE Std 1003.1:2013](#)]

```
#include <pwd.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
void func(void) {
    const char *file_format = "%s/.config";
    size_t len;
    char *pathname;
    struct passwd *pwd;

    /* Get /etc/passwd entry for current user */
    pwd = getpwuid(getuid());
    if (pwd == NULL) {
        /* Handle error */
    }

    /* Build full path name home dir from pw entry */

    len = strlen(pwd->pw_dir) + strlen(file_format) + 1;
    pathname = (char *)malloc(len);
    if (NULL == pathname) {
        /* Handle error */
    }
    int r = snprintf(pathname, len, file_format, pwd->pw_dir);
    if (r < 0 || r >= len) {
        /* Handle error */
    }
    if (unlink(pathname) != 0) {
        /* Handle error */
    }
}
```

```

    free(pathname);
}

```

The `unlink()` function is not susceptible to a symlink attack where the final component of `pathname` (the file name) is a symbolic link because `unlink()` will remove the symbolic link and not affect any file or directory named by the contents of the symbolic link. [See [F1001-C. Be careful using functions that use file names for identification.](#)] While this reduces the susceptibility of the `unlink()` function to symlink attacks, it does not eliminate it. The `unlink()` function is still susceptible if one of the directory names included in the `pathname` is a symbolic link. This could cause the `unlink()` function to delete a similarly named file in a different directory.

Compliant Solution [Windows]

This compliant solution uses the Microsoft Windows `SHGetKnownFolderPath()` API to get the current user's My Documents folder, which is then combined with the file name to create the path to the file to be deleted. The file is then removed using the `DeleteFile()` API.

```

#include <Windows.h>
#include <ShlObj.h>
#include <Shlwapi.h>

#if defined(_MSC_VER)
#pragma comment(lib, "Shlwapi")
#endif

void func(void) {
    HRESULT hr;
    LPWSTR path = 0;
    WCHAR full_path[MAX_PATH];

    hr = SHGetKnownFolderPath(&FOLDERID_Documents, 0, NULL, &path);
    if (FAILED(hr)) {
        /* Handle error */
    }
    if (!PathCombineW(full_path, path, L".config")) {
        /* Handle error */
    }
    CoTaskMemFree(path);
    if (!DeleteFileW(full_path)) {
        /* Handle error */
    }
}

```

Exceptions

ENV33-C-EX1: It is permissible to call `system()` with a null pointer argument to determine the presence of a command processor for the system.

Risk Assessments

If the command string passed to `system()`, `popen()`, or other function that invokes a command processor is not fully [sanitized](#), the risk of [exploitation](#) is high. In the worst case scenario, an attacker can execute arbitrary system commands on the compromised machine with the privileges of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ENV33-C	High	Probable	Medium	P12	L1

Related Guidelines

CERT C Secure Coding Standard	ENV03-C. Sanitize the environment when invoking external programs.
SEI CERT C++ Coding Standard	ENV02-CPP. Do not call system() if you do not need a command processor
CERT Oracle Secure Coding Standard for Java	IDS07-J. Sanitize untrusted data passed to the Runtime.exec() method
ISO/IEC TR 24772:2013	Unquoted Search Path or Element [XZQ]
ISO/IEC TS 17961:2013	Calling system [syscall]
MITRE CWE	CWE-78 , Improper Neutralization of Special Elements Used in an OS Command (aka "OS Command Injection") CWE-88 , Argument Injection or Modification

Bibliography

[IEEE Std 1003.1:2013]	XSH, System Interfaces, exec XSH, System Interfaces, popen XSH, System Interfaces, unlink
[Wheeler 2004]	

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/1AgJ>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
blacklist	Dictionary of header globbing to (list of) function name globbing(s) of forbidden functions.	dict(...)
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
forbidden_libfunc_call	Do not call system().

CertC-FI030

Exclude user input from format strings.

Input: IR

Source languages: C, C++

Details

Never call a formatted I/O function with a format string containing a [tainted value](#). An attacker who can fully or partially control the contents of a format string can crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location. Consequently, the attacker can execute arbitrary code with the permissions of the vulnerable process [[Seacord 2013b](#)]. Formatted output functions are particularly dangerous because many programmers are unaware of their capabilities. For example, formatted output functions can be used to write an integer value to a specified address using the %n conversion specifier.

Noncompliant Code Example

The `incorrect_password()` function in this noncompliant code example is called during identification and authentication to display an error message if the specified user is not found or the password is incorrect. The function accepts the name of the user as a string referenced by `user`. This is an exemplar of [untrusted data](#) that originates from an unauthenticated user. The function constructs an error message that is then output to `stderr` using the C Standard `fprintf()` function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void incorrect_password(const char *user) {
    int ret;
    /* User names are restricted to 256 or fewer characters */
    static const char msg_format[] = "%s cannot be authenticated.\n";
    size_t len = strlen(user) + sizeof(msg_format);
    char *msg = (char *)malloc(len);
    if (msg == NULL) {
        /* Handle error */
    }
    ret = snprintf(msg, len, msg_format, user);
    if (ret < 0) {
        /* Handle error */
    } else if (ret >= len) {
        /* Handle truncated output */
    }
    fprintf(stderr, msg);
    free(msg);
}
```

The `incorrect_password()` function calculates the size of the message, allocates dynamic storage, and then constructs the message in the allocated memory using the `snprintf()` function. The addition operations are not checked for integer overflow because the string referenced by `user` is known to have a length of 256 or less. Because the `%s` characters are replaced by the string referenced by `user` in the call to `snprintf()`, the resulting string needs 1 byte less than is allocated. The `snprintf()` function is commonly used for messages that are displayed in multiple locations or messages that are difficult to build. However, the resulting code contains a format-string [vulnerability](#) because the `msg` includes untrusted user input and is passed as the format-string argument in the call to `fprintf()`.

Compliant Solution (`fputs()`)

This compliant solution fixes the problem by replacing the `fprintf()` call with a call to `fputs()`, which outputs `msg` directly to `stderr` without evaluating its contents:

```
#include <stdio.h>
#include <stdlib.h>
```

```

#include <string.h>

void incorrect_password(const char *user) {
    int ret;
    /* User names are restricted to 256 or fewer characters */
    static const char msg_format[] = "%s cannot be authenticated.\n";
    size_t len = strlen(user) + sizeof(msg_format);
    char *msg = (char *)malloc(len);
    if (msg == NULL) {
        /* Handle error */
    }
    ret = snprintf(msg, len, msg_format, user);
    if (ret < 0) {
        /* Handle error */
    } else if (ret >= len) {
        /* Handle truncated output */
    }
    fputs(msg, stderr);
    free(msg);
}

```

Compliant Solution (`fprintf()`)

This compliant solution passes the untrusted user input as one of the variadic arguments to `fprintf()` and not as part of the format string, eliminating the possibility of a format-string vulnerability:

```

#include <stdio.h>

void incorrect_password(const char *user) {
    static const char msg_format[] = "%s cannot be authenticated.\n";
    fprintf(stderr, msg_format, user);
}

```

Noncompliant Code Example (POSIX)

This noncompliant code example is similar to the first noncompliant code example but uses the POSIX function `syslog()` [[IEEE Std 1003.1:2013](#)] instead of the `fprintf()` function. The `syslog()` function is also susceptible to format-string vulnerabilities.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <syslog.h>

void incorrect_password(const char *user) {
    int ret;
    /* User names are restricted to 256 or fewer characters */
    static const char msg_format[] = "%s cannot be authenticated.\n";
    size_t len = strlen(user) + sizeof(msg_format);
    char *msg = (char *)malloc(len);
    if (msg != NULL) {
        /* Handle error */
    }
    ret = snprintf(msg, len, msg_format, user);
    if (ret < 0) {
        /* Handle error */
    } else if (ret >= len) {
        /* Handle truncated output */
    }
    syslog(LOG_INFO, msg);
    free(msg);
}

```

The `syslog()` function first appeared in BSD 4.2 and is supported by Linux and other modern UNIX implementations. It is not available on Windows systems.

Compliant Solution (POSIX)

This compliant solution passes the untrusted user input as one of the variadic arguments to `syslog()` instead of including it in the format string:

```

#include <syslog.h>

void incorrect_password(const char *user) {
    static const char msg_format[] = "%s cannot be authenticated.\n";
    syslog(LOG_INFO, msg_format, user);
}

```

Risk Assessment

Failing to exclude user input from format specifiers may allow an attacker to crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location and consequently execute arbitrary code with the permissions of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FI030-C	High	Likely	Medium	P18	L1

Related Guidelines

CERT Oracle Secure Coding Standard for Java	IDS06-J. Exclude unsanitized user input from format strings
CERT Perl Secure Coding Standard	IDS30-PL. Exclude user input from format strings
ISO/IEC TR 24772:2013	Injection [RST]
ISO/IEC TS 17961:2013	Including tainted or out-of-domain input in a format string [usrfmt]
MITRE CWE	CWE-134 , Uncontrolled Format String CWE-20 , Improper Input Validation

Bibliography

[IEEE Std 1003.1:2013]	XSH, System Interfaces, <code>syslog</code>
[Seacord 2013b]	Chapter 6, "Formatted Output"
[Viega 2005]	Section 5.2.23, "Format String Problem"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/WwE>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
forbidden_one_argument_calls		['printf', 'wprintf']
forbidden_two_argument_calls		['fprintf', 'fwprintf', 'syslog']
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
tainted_string	Exclude user input from format strings.

CertC-FI034

Distinguish between characters read from a file and EOF or WEOF.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

The `EOF` macro represents a negative value that is used to indicate that the file is exhausted and no data remains when reading data from a file. `EOF` is an example of an [in-band error indicator](#). In-band error indicators are problematic to work with, and the creation of new in-band-error indicators is discouraged by [ERR02-C. Avoid in-band error indicators](#).

The byte I/O functions `fgetc()`, `getc()`, and `getchar()` all read a character from a stream and return it as an `int`. (See [STR00-C. Represent characters using an appropriate type](#).) If the stream is at the end of the file, the end-of-file indicator for the stream is set and the function returns `EOF`. If a read error occurs, the error indicator for the stream is set and the function returns `EOF`. If these functions succeed, they cast the character returned into an `unsigned char`.

Because `EOF` is negative, it should not match any `unsigned char` value. However, this is only true for [implementations](#) where the `int` type is wider than `char`. On an implementation where `int` and `char` have the same width, a character-reading function can read and return a valid character that has the same bit-pattern as `EOF`. This could occur, for example, if an attacker inserted a value that looked like `EOF` into the file or data stream to alter the behavior of the program.

The C Standard requires only that the `int` type be able to represent a maximum value of +32767 and that a `char` type be no larger than an `int`. Although uncommon, this situation can result in the integer constant expression `EOF` being indistinguishable from a valid character; that is, `(int)(unsigned char)65535 == -1`. Consequently, failing to use `feof()` and `ferror()` to detect end-of-file and file errors can result in incorrectly identifying the `EOF` character on rare implementations where `sizeof(int) == sizeof(char)`.

This problem is much more common when reading wide characters. The `fgetwc()`, `getwc()`, and `getwchar()` functions return a value of type `wint_t`. This value can represent the next wide character read, or it can represent `WEOF`, which indicates end-of-file for wide character streams. On most implementations, the `wchar_t` type has the same width as `wint_t`, and these functions can return a character indistinguishable from `WEOF`.

In the UTF-16 character set, `0xFFFF` is guaranteed not to be a character, which allows `WEOF` to be represented as the value `-1`. Similarly, all UTF-32 characters are positive when viewed as a signed 32-bit integer. All widely used character sets are designed with at least one value that does not represent a character. Consequently, it would require a custom character set designed without consideration of the C programming language for this problem to occur with wide characters or with ordinary characters that are as wide as `int`.

The C Standard `feof()` and `ferror()` functions are not subject to the problems associated with character and integer sizes and should be used to verify end-of-file and file errors for susceptible implementations [Kettlewell 2002]. Calling both functions on each iteration of a loop adds significant overhead, so a good strategy is to temporarily trust `EOF` and `WEOF` within the loop but verify them with `feof()` and `ferror()` following the loop.

Noncompliant Code Example

This noncompliant code example loops while the character `c` is not `EOF`:

```
#include <stdio.h>

void func(void) {
    int c;

    do {
        c = getchar();
    } while (c != EOF);
}
```

Although `EOF` is guaranteed to be negative and distinct from the value of any unsigned character, it is not guaranteed to be different from any such value when converted to an `int`. Consequently, when `int` has the same width as `char`, this loop may terminate prematurely.

Compliant Solution (Portable)

This compliant solution uses `feof()` to test for end-of-file and `ferror()` to test for errors:

```
#include <stdio.h>

void func(void) {
    int c;

    do {
        c = getchar();
    } while (c != EOF);
    if (feof(stdin)) {
        /* Handle end of file */
    } else if (ferror(stdin)) {
        /* Handle file error */
    } else {
        /* Received a character that resembles EOF; handle error */
    }
}
```

Noncompliant Code Example (Nonportable)

This noncompliant code example uses an assertion to ensure that the code is executed only on architectures where `int` is wider than `char` and `EOF` is guaranteed not to be a valid character value. However, this code example is noncompliant because the variable `c` is declared as a `char` rather than an `int`, making it possible for a valid character value to compare equal to the value of the `EOF` macro when `char` is signed because of sign extension:

```
#include <assert.h>
#include <limits.h>
#include <stdio.h>

void func(void) {
    char c;
    static_assert(UCHAR_MAX < UINT_MAX, "FIO34-C violation");

    do {
        c = getchar();
    } while (c != EOF);
}
```

Assuming that a `char` is a signed 8-bit type and an `int` is a 32-bit type, if `getchar()` returns the character value '`\xFF`' (decimal 255), it will be interpreted as `EOF` because this value is sign-extended to `0xFFFFFFFF` (the value of `EOF`) to perform the comparison. (See [STR34-C. Cast characters to unsigned char before converting to larger integer sizes](#).)

Compliant Solution (Nonportable)

This compliant solution declares `c` to be an `int`. Consequently, the loop will terminate only when the file is exhausted.

```
#include <assert.h>
#include <stdio.h>
#include <limits.h>
```

```

void func(void) {
    int c;
    static_assert(UCHAR_MAX < UINT_MAX, "FIO34-C violation");

    do {
        c = getchar();
    } while (c != EOF);
}

```

Noncompliant Code Example (Wide Characters)

In this noncompliant example, the result of the call to the C standard library function `getwc()` is stored into a variable of type `wchar_t` and is subsequently compared with `WEOF`:

```

#include <stddef.h>
#include <stdio.h>
#include <wchar.h>

enum { BUFFER_SIZE = 32 };

void g(void) {
    wchar_t buf[BUFFER_SIZE];
    wchar_t wc;
    size_t i = 0;

    while ((wc = getwc(stdin)) != L'\n' && wc != WEOF) {
        if (i < (BUFFER_SIZE - 1)) {
            buf[i++] = wc;
        }
    }
    buf[i] = L'\0';
}

```

This code suffers from two problems. First, the value returned by `getwc()` is immediately converted to `wchar_t` before being compared with `WEOF`. Second, there is no check to ensure that `wint_t` is wider than `wchar_t`. Both of these problems make it possible for an attacker to terminate the loop prematurely by supplying the wide-character value matching `WEOF` in the file.

Compliant Solution (Portable)

This compliant solution declares `c` to be a `wint_t` to match the integer type returned by `getwc()`. Furthermore, it does not rely on `WEOF` to determine end-of-file definitively.

```

#include <stddef.h>
#include <stdio.h>
#include <wchar.h>

enum {BUFFER_SIZE = 32 }

void g(void) {
    wchar_t buf[BUFFER_SIZE];
    wint_t wc;
    size_t i = 0;

    while ((wc = getwc(stdin)) != L'\n' && wc != WEOF) {
        if (i < BUFFER_SIZE - 1) {
            buf[i++] = wc;
        }
    }

    if (feof(stdin) || ferror(stdin)) {
        buf[i] = L'\0';
    } else {
        /* Received a wide character that resembles WEOF; handle error */
    }
}

```

Exceptions

FIO34-C-EX1: A number of C functions do not return characters but can return `EOF` as a status code. These functions include `fclose()`, `fflush()`, `fputs()`, `fscanf()`, `puts()`, `scanf()`, `sscanf()`, `vfscanf()`, and `vscanf()`. These return values can be compared to `EOF` without validating the result.

Risk Assessment

Incorrectly assuming characters from a file cannot match `EOF` or `WEOF` has resulted in significant vulnerabilities, including command injection attacks. (See the [*CA-1996-22](#) advisory.)

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO34-C	High	Probable	Medium	P12	L1

Related Guidelines

CERT C Secure Coding Standard	STR00-C. Represent characters using an appropriate type INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data
CERT Oracle Secure Coding Standard for Java	FI008-J. Use an int to capture the return value of methods that read a character or byte
ISO/IEC TS 17961:2013	Using character values that are indistinguishable from EOF [chreof]

Bibliography

[Kettlewell 2002]	Section 1.2, "<stdio.h> and Character Types"
[NIST 2006]	SAMATE Reference Dataset Test Case ID 000-000-088
[Summit 2005]	Question 12.2

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/dwGKBw>], Copyright [C] 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
functions_under_test	Function which comparison with EOF/WEOF might be unsafe	['fgetc', 'getc', 'getchar', 'fgetwc', 'getwc', 'getwchar']
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
limit_header_files	Header files which declare following constants: UCHAR_MAX, UINT_MAX, WCHAR_MAX, WINT_MAX.	['limits.h']
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
unsafe_eof	Distinguish between characters read from a file and EOF or WEOF.

CertC-FI037

Do not assume that fgets() or fgetws() returns a nonempty string when successful.

Input: IR

Source languages: C, C++

Details

Errors can occur when incorrect assumptions are made about the type of data being read. These assumptions may be violated, for example, when binary data has been read from a file instead of text from a user's terminal or the output of a process is piped to `stdin`. (See [FI014-C. Understand the difference between text mode and binary mode with file streams](#).) On some systems, it may also be possible to input a null byte (as well as other binary codes) from the keyboard.

Subclause 7.21.7.2 of the C Standard [[ISO/IEC 9899:2011](#)] says,

The `fgets` function returns `s` if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned.

The wide-character function `fgetws()` has the same behavior. Therefore, if `fgets()` or `fgetws()` returns a non-null pointer, it is safe to assume that the array contains data. However, it is erroneous to assume that the array contains a nonempty string because the data may contain null characters.

Noncompliant Code Example

This noncompliant code example attempts to remove the trailing newline (\n) from an input line. The `fgets()` function is typically used to read a newline-terminated line of input from a stream. It takes a size parameter for the destination buffer and copies, at most, `size - 1` characters from a stream to a character array.

```
#include <stdio.h>
#include <string.h>
```

```

enum { BUFFER_SIZE = 1024 };

void func(void) {
    char buf[BUFFER_SIZE];

    if (fgets(buf, sizeof(buf), stdin) == NULL) {
        /* Handle error */
    }
    buf[strlen(buf) - 1] = '\0';
}

```

The `strlen()` function computes the length of a string by determining the number of characters that precede the terminating null character. A problem occurs if the first character read from the input by `fgets()` happens to be a null character. This may occur, for example, if a binary data file is read by the `fgets()` call [Lai 2006]. If the first character in `buf` is a null character, `strlen(buf)` returns 0, the expression `strlen(buf) - 1` wraps around to a large positive value, and a write-outside-array-bounds error occurs.

Compliant Solution

This compliant solution uses `strchr()` to replace the newline character in the string if it exists:

```

#include <stdio.h>
#include <string.h>

enum { BUFFER_SIZE = 1024 };

void func(void) {
    char buf[BUFFER_SIZE];
    char *p;

    if (fgets(buf, sizeof(buf), stdin)) {
        p = strchr(buf, '\n');
        if (p) {
            *p = '\0';
        }
    } else {
        /* Handle error */
    }
}

```

Risk Assessment

Incorrectly assuming that character data has been read can result in an out-of-bounds memory write or other flawed logic.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
F1037-C	High	Probable	Medium	P12	L1

Related Guidelines

CERT C Secure Coding Standard	F1014-C. Understand the difference between text mode and binary mode with file streams F1020-C. Avoid unintentional truncation when using fgets() or fgetws()
MITRE CWE	CWE-119 , Improper Restriction of Operations within the Bounds of a Memory Buffer CWE-123 , Write-what-where Condition CWE-125 , Out-of-bounds Read CWE-241 , Improper Handling of Unexpected Data Type

Bibliography

[ISO/IEC 9899:2011]	Subclause 7.21.7.2, "The fgets Function" Subclause 7.29.3.2, "The fgetws Function"
[Lai 2006]	
[Seacord 2013]	Chapter 2, "Strings"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/dh>], Copyright (C) 1995–2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
strlen_functions	Functions which test for empty strings.	['strlen']

Possible Messages

Name	Message
empty_test	Do not assume that fgets() or fgetws() returns a nonempty string when successful.

CertC-FI038

Do not copy a FILE object.

Input: IR

Source languages: C, C++

Details

According to the C Standard, 7.21.3, paragraph 6 [[ISO/IEC 9899:2011](#)],

The address of the FILE object used to control a stream may be significant; a copy of a FILE object need not serve in place of the original.

Consequently, do not copy a FILE object.

Noncompliant Code Example

This noncompliant code example can fail because a by-value copy of stdout is being used in the call to fputs():

```
#include <stdio.h>

int main(void) {
    FILE my_stdout = *stdout;
    if (fputs("Hello, World!\n", &my_stdout) == EOF) {
        /* Handle error */
    }
    return 0;
}
```

When compiled under Microsoft Visual Studio 2013 and run on Windows, this noncompliant example results in an "access violation" at runtime.

Compliant Solution

In this compliant solution, a copy of the stdout pointer to the FILE object is used in the call to fputs():

```
#include <stdio.h>

int main(void) {
    FILE *my_stdout = stdout;
    if (fputs("Hello, World!\n", my_stdout) == EOF) {
        /* Handle error */
    }
    return 0;
}
```

Risk Assessment

Using a copy of a FILE object in place of the original may result in a crash, which can be used in a [denial-of-service attack](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FI038-C	Low	Probable	Medium	P4	L3

Related Guidelines

ISO/IEC TS 17961:2013	Copying a FILE object [filecpy]
---------------------------------------	---------------------------------

Bibliography

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/wAw>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
file_pointer_dereference	A pointer to a FILE object shall not be dereferenced

CertC-FI047

Use valid format strings.

Input: IR

Source languages: C, C++

Details

The formatted output functions (`fprintf()` and related functions) convert, format, and print their arguments under control of a *format string*. The C Standard, 7.21.6.1, paragraph 3 [ISO/IEC 9899:2011], specifies

The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

Each *conversion specification* is introduced by the % character followed (in order) by

- Zero or more *flags* (in any order), which modify the meaning of the conversion specification
- An optional minimum field *width*
- An optional *precision* that gives the minimum number of digits to appear for certain conversion specifiers
- An optional *length modifier* that specifies the size of the argument
- A *conversion specifier character* that indicates the type of conversion to be applied

Common mistakes in creating format strings include

- Providing an incorrect number of arguments for the format string
- Using invalid conversion specifiers
- Using a flag character that is incompatible with the conversion specifier
- Using a length modifier that is incompatible with the conversion specifier
- Mismatching the argument type and conversion specifier
- Using an argument of type other than `int` for *width* or *precision*

The following table summarizes the compliance of various conversion specifications. The first column contains one or more conversion specifier characters. The next four columns consider the combination of the specifier characters with the various flags (the apostrophe ['], -, +, the space character, #, and \). The next eight columns consider the combination of the specifier characters with the various length modifiers (h, hh, l, ll, j, z, t, and L). The last column denotes the expected types of arguments matched with the original specifier characters.

Valid combinations are marked with a type name; arguments matched with the conversion specification are interpreted as that type. For example, an argument matched with the specifier %hd is interpreted as a `short`, so short appears in the cell where d and h intersect. The last column denotes the expected types of arguments matched with the original specifier characters.

Valid and meaningful combinations are marked by the ✓ symbol (save for the length modifier columns, as described previously). Valid combinations that have no effect are labeled N/E. Using a combination marked by the ✗ symbol, using a specification not represented in the table, or using an argument of an unexpected type is undefined behavior. (See undefined behaviors [153](#), [155](#), [157](#), [158](#), [161](#), and [162](#).)

Conversion Specifier Character	<i>XSI</i>	- + SPACE	#	0	h	hh	l	ll	j	z	t	L	Argument Type
d, i	✓	✓	✗	✓	short	signed char	long	long long	intmax_t	size_t	ptrdiff_t	✗	Signed integer
o	✗	✓	✓	✓	unsigned short	unsigned char	unsigned long	unsigned long long	uintmax_t	size_t	ptrdiff_t	✗	Unsigned integer
u	✓	✓	✗	✓	unsigned short	unsigned char	unsigned long	unsigned long long	uintmax_t	size_t	ptrdiff_t	✗	Unsigned integer
x, X	✗	✓	✓	✓	unsigned short	unsigned char	unsigned long	unsigned long long	uintmax_t	size_t	ptrdiff_t	✗	Unsigned integer
f, F	✓	✓	✓	✓	✗	✗	N/E	N/E	✗	✗	✗	long double	double or long double
e, E	✗	✓	✓	✓	✗	✗	N/E	N/E	✗	✗	✗	long double	double or long double
g, G	✓	✓	✓	✓	✗	✗	N/E	N/E	✗	✗	✗	long double	double or long double
a, A	✓	✓	✓	✓	✗	✗	N/E	N/E	✗	✗	✗	long double	double or long double
c	✗	✓	✗	✗	✗	✗	wint_t	✗	✗	✗	✗	✗	int or wint_t
s	✗	✓	✗	✗	✗	✗	NTWS	✗	✗	✗	✗	✗	NTBS or NTWS
p	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	void*
n	✗	✓	✗	✗	short*	char*	long*	long long*	intmax_t*	size_t*	ptrdiff_t*	✗	Pointer to integer
c <i>XSI</i>	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	wint_t
s <i>XSI</i>	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	NTWS
%	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	None

SPACE: The space (" ") character

N/E: No effect

NTBS: `char*` argument pointing to a null-terminated character string

NTWS: `wchar_t*` argument pointing to a null-terminated wide character string

XSI: [ISO/IEC 9945-2003](#) XSI extension

The formatted input functions (`fscanf()` and related functions) use similarly specified format strings and impose similar restrictions on their format strings and arguments.

Do not supply an unknown or invalid conversion specification or an invalid combination of flag character, precision, length modifier, or conversion specifier to a formatted IO function. Likewise, do not provide a number or type of argument that does not match the argument type of the conversion specifier used in the format string.

Format strings are usually string literals specified at the call site, but they need not be. However, they should not contain [tainted values](#). (See [F1030-C. Exclude user input from format strings](#) for more information.)

Noncompliant Code Example

Mismatches between arguments and conversion specifications may result in [undefined behavior](#). Compilers may diagnose type mismatches in formatted output function invocations. In this noncompliant code example, the `error_type` argument to `printf()` is incorrectly matched with the `s` specifier rather than with the `a` specifier. Likewise, the `error_msg` argument is incorrectly matched with the `d` specifier instead of the `s` specifier. These usages result in [undefined behavior](#). One possible result of this invocation is that `printf()` will interpret the `error_type` argument as a pointer and try to read a string from the address that `error_type` contains, possibly resulting in an access violation.

```
#include <stdio.h>

void func(void) {
    const char *error_msg = "Resource not available to user.";
    int error_type = 3;
    /* ... */
}
```

```

    printf("Error (type %s): %d\n", error_type, error_msg);
    /* ... */
}

```

Compliant Solution

This compliant solution ensures that the arguments to the `printf()` function match their respective conversion specifications:

```
#include <stdio.h>

void func(void) {
    const char *error_msg = "Resource not available to user.";
    int error_type = 3;
    /* ... */
    printf("Error (type %d): %s\n", error_type, error_msg);

    /* ... */
}
```

Risk Assessment

Incorrectly specified format strings can result in memory corruption or [abnormal program termination](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FI047-C	High	Unlikely	Medium	P6	L2

Related Guidelines

SEI CERT C++ Coding Standard	FI000-CPP. Take care when creating format strings
ISO/IEC TS 17961:2013	Using invalid format strings [invfmtstr]
MITRE CWE	CWE-686 , Function Call with Incorrect Argument Type

Bibliography

[ISO/IEC 9899:2011]	Subclause 7.21.6.1, "The <code>fprintf</code> Function"
-------------------------------------	---

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/wQA1>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
allow_extra_args	Whether to allow additional arguments that are not used by the format string.	False
allow_gnu_extensions	Whether to allow the GNU extensions to format specifications.	True
allow_unknown_specs	Whether to allow unknown format specifications. It may be necessary to set this option when using implementation-specific extensions. Arguments are not checked when the format string contains unknown format specifications.	False
functions	A dictionary mapping the names of the functions to check, to a triple {function_kind, fmt_param_index, arg_start_index} where function_kind is either 'printf' or 'scanf', fmt_param_index is the index of the format-string parameter, and arg_start_index is the index of the first variadic argument.	dict{...}
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
arg_type_mismatch	{ } expects argument of type '{ }', but argument { } has type '{ }'
buffer_too_small	{ } may write up to { } characters to buffer of size { }. (disabled)
invalid_conversion	Invalid or non-standard conversion specification
matching_arg_expected	{ } expects a matching '{ }' argument
precision_for_conversion	Precision must not be used with %{ } conversion specifier
too_many_args	Too many arguments for format.
unknown_buffer_size	Potential buffer overflow: { } used with buffer of unknown size. (disabled)
unlimited_read	Potential buffer overflow: { } has no limit on amount of characters read. (disabled)
unsupported_assignment_suppression	%n does not support assignment suppression
unsupported_field_width	%n does not support field width
unsupported_flags	%n does not support flags
unsupported_flags_modifiers	Cannot use any flags or modifiers with '%%'
unsupported_hash	%{ } does not support the '#' flag
unsupported_i_flag	%{ } does not support the 'l' flag
unsupported_length_modifier	%{ } does not support the '{ }' length modifier
unsupported_tick	%{ } does not support the "" flag
unsupported_zero	%{ } does not support the '0' flag

CertC-SIG30

Call only asynchronous-safe functions within signal handlers.

Input: IR

Source languages: C, C++

Details

Call only [asynchronous-safe functions](#) within signal handlers. For [strictly conforming](#) programs, only the C standard library functions `abort()`, `_Exit()`, `quick_exit()`, and `signal()` can be safely called from within a signal handler.

The C Standard, 7.14.1.1, paragraph 5 [[ISO/IEC 9899:2011](#)], states that if the signal occurs other than as the result of calling the `abort()` or `raise()` function, the behavior is [undefined](#) if

...the signal handler calls any function in the standard library other than the `abort` function, the `_Exit` function, the `quick_exit` function, or the `signal` function with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler.

Implementations may define a list of additional asynchronous-safe functions. These functions can also be called within a signal handler. This restriction applies to library functions as well as application-defined functions.

According to the C Rationale, 7.14.1.1 [[C99 Rationale 2003](#)],

When a signal occurs, the normal flow of control of a program is interrupted. If a signal occurs that is being trapped by a signal handler, that handler is invoked. When it is finished, execution continues at the point at which the signal occurred. This arrangement can cause problems if the signal handler invokes a library function that was being executed at the time of the signal.

In general, it is not safe to invoke I/O functions from within signal handlers. Programmers should ensure a function is included in the list of an implementation's asynchronous-safe functions for all implementations the code will run on before using them in signal handlers.

Noncompliant Code Example

In this noncompliant example, the C standard library functions `fprintf()` and `free()` are called from the signal handler via the function `log_message()`. Neither function is [asynchronous-safe](#).

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
enum { MAXLINE = 1024 };
char *info = NULL;

void log_message(void) {
    fputs(info, stderr);
}
```

```

void handler(int signum) {
    log_message();
    free(info);
    info = NULL;
}

int main(void) {
    if (signal(SIGINT, handler) == SIG_ERR) {
        /* Handle error */
    }
    info = (char *)malloc(MAXLINE);
    if (info == NULL) {
        /* Handle Error */
    }

    while (1) {
        /* Main loop program code */

        log_message();

        /* More program code */
    }
    return 0;
}

```

Compliant Solution

Signal handlers should be as concise as possible-ideally by unconditionally setting a flag and returning. This compliant solution sets a flag of type `volatile sig_atomic_t` and returns; the `log_message()` and `free()` functions are called directly from `main()`:

```

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

enum { MAXLINE = 1024 };
volatile sig_atomic_t eflag = 0;
char *info = NULL;

void log_message(void) {
    fputs(info, stderr);
}

void handler(int signum) {
    eflag = 1;
}

int main(void) {
    if (signal(SIGINT, handler) == SIG_ERR) {
        /* Handle error */
    }
    info = (char *)malloc(MAXLINE);
    if (info == NULL) {
        /* Handle error */
    }

    while (!eflag) {
        /* Main loop program code */

        log_message();

        /* More program code */
    }

    log_message();
    free(info);
    info = NULL;
}

return 0;
}

```

Noncompliant Code Example (`longjmp()`)

Invoking the `longjmp()` function from within a signal handler can lead to [undefined behavior](#) if it results in the invocation of any non-[asynchronous-safe](#) functions. Consequently, neither `longjmp()` nor the POSIX `siglongjmp()` functions should ever be called from within a signal handler.

This noncompliant code example is similar to a [vulnerability](#) in an old version of Sendmail [VU #834865]. The intent is to execute code in a `main()` loop, which also logs some data. Upon receiving a `SIGINT`, the program transfers out of the loop, logs the error, and terminates.

However, an attacker can [exploit](#) this noncompliant code example by generating a `SIGINT` just before the second `if` statement in `log_message()`. The result is that `longjmp()` transfers control back to `main()`, where `log_message()` is called again. However, the first `if` statement would not be executed this time (because `buf` is not set to `NULL` as a result of the interrupt), and the program would write to the invalid memory location referenced by `buf0`.

```

#include <setjmp.h>
#include <signal.h>
#include <stdlib.h>

enum { MAXLINE = 1024 };
static jmp_buf env;

void handler(int signum) {
    longjmp(env, 1);
}

void log_message(char *info1, char *info2) {

```

```

static char *buf = NULL;
static size_t bufsize;
char buf0[MAXLINE];

if (buf == NULL) {
    buf = buf0;
    bufsize = sizeof(buf0);
}

/*
 * Try to fit a message into buf, else reallocate
 * it on the heap and then log the message.
 */

/* Program is vulnerable if SIGINT is raised here */

if (buf == buf0) {
    buf = NULL;
}
}

int main(void) {
    if (signal(SIGINT, handler) == SIG_ERR) {
        /* Handle error */
    }
    char *info1;
    char *info2;

    /* info1 and info2 are set by user input here */

    if (setjmp(env) == 0) {
        while (1) {
            /* Main loop program code */
            log_message(info1, info2);
            /* More program code */
        }
    } else {
        log_message(info1, info2);
    }

    return 0;
}

```

Compliant Solution

In this compliant solution, the call to `longjmp()` is removed; the signal handler sets an error flag instead:

```

#include <signal.h>
#include <stdlib.h>

enum { MAXLINE = 1024 };
volatile sig_atomic_t eflag = 0;

void handler(int signum) {
    eflag = 1;
}

void log_message(char *info1, char *info2) {
    static char *buf = NULL;
    static size_t bufsize;
    char buf0[MAXLINE];

    if (buf == NULL) {
        buf = buf0;
        bufsize = sizeof(buf0);
    }

    /*
     * Try to fit a message into buf, else reallocate
     * it on the heap and then log the message.
     */
    if (buf == buf0) {
        buf = NULL;
    }
}

int main(void) {
    if (signal(SIGINT, handler) == SIG_ERR) {
        /* Handle error */
    }
    char *info1;
    char *info2;

    /* info1 and info2 are set by user input here */

    while (!eflag) {
        /* Main loop program code */
        log_message(info1, info2);
        /* More program code */
    }

    log_message(info1, info2);

    return 0;
}

```

Noncompliant Code Example (`raise()`)

In this noncompliant code example, the `int_handler()` function is used to carry out tasks specific to `SIGINT` and then raises `SIGTERM`. However, there is a nested call to the `raise()` function, which is [undefined behavior](#).

```
#include <signal.h>
#include <stdlib.h>

void term_handler(int signum) {
    /* SIGTERM handler */
}

void int_handler(int signum) {
    /* SIGINT handler */
    if (raise(SIGTERM) != 0) {
        /* Handle error */
    }
}

int main(void) {
    if (signal(SIGTERM, term_handler) == SIG_ERR) {
        /* Handle error */
    }
    if (signal(SIGINT, int_handler) == SIG_ERR) {
        /* Handle error */
    }

    /* Program code */
    if (raise(SIGINT) != 0) {
        /* Handle error */
    }
    /* More code */

    return EXIT_SUCCESS;
}
```

Compliant Solution

In this compliant solution, `int_handler()` invokes `term_handler()` instead of raising `SIGTERM`:

```
#include <signal.h>
#include <stdlib.h>

void term_handler(int signum) {
    /* SIGTERM handler */
}

void int_handler(int signum) {
    /* SIGINT handler */
    /* Pass control to the SIGTERM handler */
    term_handler(SIGTERM);
}

int main(void) {
    if (signal(SIGTERM, term_handler) == SIG_ERR) {
        /* Handle error */
    }
    if (signal(SIGINT, int_handler) == SIG_ERR) {
        /* Handle error */
    }

    /* Program code */
    if (raise(SIGINT) != 0) {
        /* Handle error */
    }
    /* More code */

    return EXIT_SUCCESS;
}
```

Implementation Details

POSIX

The following table from the POSIX standard [[IEEE Std 1003.1:2013](#)] defines a set of functions that are [asynchronous-signal-safe](#). Applications may invoke these functions, without restriction, from a signal handler.

_Exit()	fexecve()	posix_trace_event()	sigprocmask()
_exit()	fork()	pselect()	sigqueue()
abort()	fstat()	pthread_kill()	sigset()
accept()	fstatat()	pthread_self()	sigsuspend()
access()	fsync()	pthread_sigmask()	sleep()
aio_error()	ftruncate()	raise()	sockatmark()
aio_return()	futimens()	read()	socket()
aio_suspend()	getegid()	readlink()	socketpair()
alarm()	geteuid()	readlinkat()	stat()
bind()	getgid()	recv()	symlink()
cfgetispeed()	getgroups()	recvfrom()	symlinkat()
cfgetospeed()	getpeername()	recvmsg()	tcdrain()
cfsetispeed()	getpgroup()	rename()	tcflow()
cfsetospeed()	getpid()	renameat()	tcflush()
chdir()	getppid()	rmdir()	tcgetattr()
chmod()	getsockname()	select()	tcgetpgrp()
chown()	getsockopt()	sem_post()	tcsendbreak()
clock_gettime()	getuid()	send()	tcsetattr()
close()	kill()	sendmsg()	tcsetpgrp()
connect()	link()	sendto()	time()
creat()	linkat()	setgid()	timer_getoverrun()
dup()	listen()	setpgid()	timer_gettime()
dup2()	lseek()	setsid()	timer_settime()
execl()	lstat()	setsockopt()	times()
execle()	mkdir()	setuid()	umask()
execv()	mkdirat()	shutdown()	uname()
execve()	mkfifo()	sigaction()	unlink()
faccessat()	mkfifoat()	sigaddset()	unlinkat()
fchdir()	mknod()	sigdelset()	utime()
fchmod()	mknodat()	sigemptyset()	utimensat()
fchmodat()	open()	sigfillset()	utimes()
fchown()	openat()	sigismember()	wait()
fchownat()	pause()	signal()	waitpid()
fcntl()	pipe()	sigpause()	write()
fdatasync()	poll()	sigpending()	

All functions not listed in this table are considered to be unsafe with respect to signals. In the presence of signals, all POSIX functions behave as defined when called from or interrupted by a signal handler, with a single exception: when a signal interrupts an unsafe function and the signal handler calls an unsafe function, the behavior is undefined.

The C Standard, 7.14.1.1, paragraph 4 [ISO/IEC 9899:2011], states

If the signal occurs as the result of calling the abort or raise function, the signal handler shall not call the raise function.

However, in the description of `signal()`, POSIX [IEEE Std 1003.1:2013] states

This restriction does not apply to POSIX applications, as POSIX.1-2008 requires `raise()` to be async-signal-safe.

See also [undefined behavior 131](#).

OpenBSD

The OpenBSD `signal()` manual page lists a few additional functions that are asynchronous-safe in OpenBSD but "probably not on other systems" [OpenBSD], including `snprintf()`, `vsnprintf()`, and `syslog_r()` but only when the `syslog_data` struct is initialized as a local variable.

Risk Assessment

Invoking functions that are not [asynchronous-safe](#) from within a signal handler is [undefined behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SIG30-C	High	Likely	Medium	P18	L1

Related Guidelines

ISO/IEC TS 17961:2013	Calling functions in the C Standard Library other than <code>abort</code> , <code>_Exit</code> , and <code>signal</code> from within a signal handler [asynccsig]
MITRE CWE	CWE-479 , Signal Handler Use of a Non-reentrant Function

Bibliography

[C99 Rationale 2003]	Subclause 5.2.3, "Signals and Interrupts" Subclause 7.14.1.1, "The <code>signal</code> Function"
[Dowd 2006]	Chapter 13, "Synchronization and State"
[Greenman 1997]	
[IEEE Std 1003.1:2013]	XSH, System Interfaces, <code>longjmp</code> XSH, System Interfaces, <code>raise</code>
[ISO/IEC 9899:2011]	7.14.1.1, "The <code>signal</code> Function"
[OpenBSD]	signal() Man Page
[VU #834865]	
[Zalewski 2001]	"Delivering Signals for Fun and Profit"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/34At>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
allow_posix_async_safe	Allow POSIX async-safe functions in addition to the configured whitelist. Value of this configuration option is either 0 (if POSIX not allowed), or the year of a POSIX standard. (2001, 2004, 2008, 2013 or 2016).	2013
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
report_unknown_calls	Report calls to unknown functions declared in user headers. Note: this option is automatically deactivated during single-file analysis.	True
signal_handler_registrations	Names of functions that are used to register signal handlers. All functions that are passed as arguments to one of the registration functions are considered signal handler functions.	('signal', 'sigaction')
whitelist	Async-safe functions	set(['signal', 'abort', '_Exit', 'quick_exit'])

Possible Messages

Name	Message
invalid_system_call	Signal handler should call only async-safe functions.
unknown_call	Signal handler calling unknown function, potentially not async-safe.

CertC-SIG31

Do not access shared objects in signal handlers.

Input: IR

Source languages: C, C++

Details

Accessing or modifying shared objects in signal handlers can result in race conditions that can leave data in an inconsistent state. The two exceptions (C Standard, 5.1.2.3, paragraph 5) to this rule are the ability to read from and write to lock-free atomic objects and variables of type `volatile sig_atomic_t`. Accessing any other type of object from a signal handler is [undefined behavior](#). (See [undefined behavior 131](#).)

The need for the `volatile` keyword is described in [DCL22-C. Use volatile for data that cannot be cached](#).

The type `sig_atomic_t` is the integer type of an object that can be accessed as an atomic entity even in the presence of asynchronous interrupts. The type of `sig_atomic_t` is [implementation-defined](#), though it provides some guarantees. Integer values ranging from `SIG_ATOMIC_MIN` through `SIG_ATOMIC_MAX`, inclusive, may be safely stored to a variable of the type. In addition, when `sig_atomic_t` is a signed integer type, `SIG_ATOMIC_MIN` must be no greater than -127 and `SIG_ATOMIC_MAX` no less than 127. Otherwise, `SIG_ATOMIC_MIN` must be 0 and `SIG_ATOMIC_MAX` must be no less than 255. The macros `SIG_ATOMIC_MIN` and `SIG_ATOMIC_MAX` are defined in the header `<stdint.h>`.

According to the C99 Rationale [[C99 Rationale 2003](#)], other than calling a limited, prescribed set of library functions,

the C89 Committee concluded that about the only thing a [strictly conforming](#) program can do in a signal handler is to assign a value to a `volatile static` variable which can be written uninterruptedly and promptly return.

However, this issue was discussed at the April 2008 meeting of ISO/IEC WG14, and it was agreed that there are no known [implementations](#) in which it would be an error to read a value from a `volatile sig_atomic_t` variable, and the original intent of the committee was that both reading and writing variables of `volatile sig_atomic_t` would be strictly conforming.

The signal handler may also call a handful of functions, including `abort()`. (See [SIG30-C. Call only asynchronous-safe functions within signal handlers](#) for more information.)

Noncompliant Code Example

In this noncompliant code example, `err_msg` is updated to indicate that the `SIGINT` signal was delivered. The `err_msg` variable is a character pointer and not a variable of type `volatile sig_atomic_t`.

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

enum { MAX_MSG_SIZE = 24 };
char *err_msg;
```

```

void handler(int signum) {
    strcpy(err_msg, "SIGINT encountered.");
}

int main(void) {
    signal(SIGINT, handler);

    err_msg = (char *)malloc(MAX_MSG_SIZE);
    if (err_msg == NULL) {
        /* Handle error */
    }
    strcpy(err_msg, "No errors yet.");
    /* Main code loop */
    return 0;
}

```

Compliant Solution {Writing volatile sig_atomic_t}

For maximum portability, signal handlers should only unconditionally set a variable of type `volatile sig_atomic_t` and return, as in this compliant solution:

```

#include <signal.h>
#include <stdlib.h>
#include <string.h>

enum { MAX_MSG_SIZE = 24 };
volatile sig_atomic_t e_flag = 0;

void handler(int signum) {
    e_flag = 1;
}

int main(void) {
    char *err_msg = (char *)malloc(MAX_MSG_SIZE);
    if (err_msg == NULL) {
        /* Handle error */
    }
    signal(SIGINT, handler);
    strcpy(err_msg, "No errors yet.");
    /* Main code loop */
    if (e_flag) {
        strcpy(err_msg, "SIGINT received.");
    }
    return 0;
}

```

Compliant Solution {Lock-Free Atomic Access}

Signal handlers can refer to objects with static or thread storage durations that are lock-free atomic objects, as in this compliant solution:

```

#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <stdatomic.h>

#ifndef __STDC_NO_ATOMICS__
#error "Atomics are not supported"
#elif ATOMIC_INT_LOCK_FREE == 0
#error "int is never lock-free"
#endif

atomic_int e_flag = ATOMIC_VAR_INIT(0);

void handler(int signum) {
    e_flag = 1;
}

int main(void) {
    enum { MAX_MSG_SIZE = 24 };
    char err_msg[MAX_MSG_SIZE];
#if ATOMIC_INT_LOCK_FREE == 1
    if (!atomic_is_lock_free(&e_flag)) {
        return EXIT_FAILURE;
    }
#endif
    if (signal(SIGINT, handler) == SIG_ERR) {
        return EXIT_FAILURE;
    }
    strcpy(err_msg, "No errors yet.");
    /* Main code loop */
    if (e_flag) {
        strcpy(err_msg, "SIGINT received.");
    }
    return EXIT_SUCCESS;
}

```

Exceptions

SIG31-C-EX1: The C Standard, 7.14.1.1 paragraph 5 [ISO/IEC 9899:2011], makes a special exception for `errno` when a valid call to the `signal()` function results in a `SIG_ERR` return, allowing `errno` to take an indeterminate value. (See [ERR32-C. Do not rely on indeterminate values of errno](#).)

Risk Assessment

Accessing or modifying shared objects in signal handlers can result in accessing data in an inconsistent state. Michal Zalewski's paper "Delivering Signals for Fun and Profit" [Zalewski 2001] provides some examples of [vulnerabilities](#) that can result from violating this and other signal-handling rules.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SIG31-C	High	Likely	High	P9	L2

Related Guidelines

ISO/IEC TS 17961:2013	Accessing shared objects in signal handlers [accsig]
MITRE CWE	CWE-662 , Improper Synchronization

Bibliography

[C99 Rationale 2003]	5.2.3, "Signals and Interrupts"
[ISO/IEC 9899:2011]	Subclause 7.14.1.1, "The <code>signal</code> Function"
[Zalewski 2001]	

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/GIEt>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
access_kinds	Access kinds (e.g. Reading_Operand_Interface, Writing_Operand_Interface, Address_Operand_Interface).	['Reading_Operand_Interface', 'Writing_Operand_Interface']
allow_c11_atomics	If set, don't report races on C11 atomic variables.	True
allow_volatile_sig_atomic_t	If set, don't report races on variables of type "volatile sig_atomic_t".	True
debug_output	Option to provide diagnostic output.	False
enter_critical_functions	List of function names to enter a critical region.	[]
enter_critical_macros	List of macro names to enter a critical region (macros must expand to asm() statement).	[]
excluded_routines	List of functions that should be excluded from check.	[]
excluded_subgraphs	List of entry functions to subgraphs that should be excluded as subgraph from check.	[]
exit_critical_functions	List of function names to exit a critical region.	[]
exit_critical_macros	List of macro names to exit a critical region (macros must expand to asm() statement).	[]
inspect_pointers	Whether pointer targets should be inspected to detect more global variable uses.	False
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
nested_critical_regions	If set to True, critical regions nest; if set to False, a single exit-critical-region terminates all open critical regions.	True
output_safe_accesses	When enabled, outputs not only unsafe variable accesses, but also the safe ones.	False
partitions	Dict with partition name as key and dict as value which may contain keys 'entries' and/or 'vectors' with lists of entry points or vector table variables respectively. If special partition '*IRQ*' to configure interrupt handlers is missing, all functions not reached by any of the other options are treated as interrupt handlers.	dict(...)
priority	Priority based on the combination of severity, likelihood and remediation cost	9
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high
report_cfg_based_critical_region_issues	Report unbalanced lock/unlock pairs within a routine. This has the same intention, but is slightly less strict than the purely syntactic check performed by the rule Parallelism-IncorrectCriticalSection.	False
show_identical_access	When enabled, outputs variable accesses of same kind (i.e., R/R and W/W).	False
show_object_number	Option for debugging (shows internal node numbers).	False
treat_types_as_atomic	List of type-patterns. A type-pattern is either a regular expression of a type name, or a triple of {min. alignment, max. size, type name-regex}. Each of the triple's components may be None. None is interpreted as general wildcard.	[]

Possible Messages

Name	Message
multiple_lock_add	Lock is acquired while it is already locked.
removed_nonexisting_lock	Lock is released, although it is not currently locked.
unbalanced_locks_path	Different control flow paths have different sets of locks.
unbalanced_locks_routine	Routine may return with different lock set than it is entered with ({in_set} vs {out_set}).

CertC-SIG34

Do not call `signal()` from within interruptible signal handlers.

Input: IR

Source languages: C, C++

Details

A signal handler should not reassert its desire to handle its own signal. This is often done on *nonpersistent* platforms—that is, platforms that, upon receiving a signal, reset the handler for the signal to SIG_DFL before calling the bound signal handler. Calling `signal()` under these conditions presents a race condition. (See [SIG01-C. Understand implementation-specific details regarding signal handler persistence](#).)

A signal handler may call `signal()` only if it does not need to be [asynchronous-safe](#) (that is, if all relevant signals are masked so that the handler cannot be interrupted).

Noncompliant Code Example (POSIX)

On nonpersistent platforms, this noncompliant code example contains a race window, starting when the host environment resets the signal and ending when the handler calls `signal()`. During that time, a second signal sent to the program will trigger the default signal behavior, consequently defeating the persistent behavior implied by the call to `signal()` from within the handler to reassert the binding.

If the environment is persistent (that is, it does not reset the handler when the signal is received), the `signal()` call from within the `handler()` function is redundant.

```
#include <signal.h>

void handler(int signum) {
    if (signal(signum, handler) == SIG_ERR) {
        /* Handle error */
    }
    /* Handle signal */
}

void func(void) {
    if (signal(SIGUSR1, handler) == SIG_ERR) {
        /* Handle error */
    }
}
```

Compliant Solution (POSIX)

Calling the `signal()` function from within the signal handler to reassert the binding is unnecessary for persistent platforms, as in this compliant solution:

```
#include <signal.h>

void handler(int signum) {
    /* Handle signal */
}

void func(void) {
    if (signal(SIGUSR1, handler) == SIG_ERR) {
        /* Handle error */
    }
}
```

Compliant Solution (POSIX)

POSIX defines the `sigaction()` function, which assigns handlers to signals in a similar manner to `signal()` but allows the caller to explicitly set persistence. Consequently, the `sigaction()` function can be used to eliminate the race window on nonpersistent platforms, as in this compliant solution:

```
#include <signal.h>
#include <stddef.h>

void handler(int signum) {
    /* Handle signal */
}

void func(void) {
    struct sigaction act;
    act.sa_handler = handler;
    act.sa_flags = 0;
    if (sigemptyset(&act.sa_mask) != 0) {
        /* Handle error */
    }
}
```

```

if (sigaction(SIGUSR1, &act, NULL) != 0) {
    /* Handle error */
}

```

Although the handler in this example does not call `signal()`, it could do so safely because the signal is masked and the handler cannot be interrupted. If the same handler is installed for more than one signal, the signals must be masked explicitly in `act.sa_mask` to ensure that the handler cannot be interrupted because the system masks only the signal being delivered.

POSIX recommends that new applications should use `sigaction()` rather than `signal()`. The `sigaction()` function is not defined by the C Standard and is not supported on some platforms, including Windows.

Compliant Solution [Windows]

There is no safe way to implement persistent signal-handler behavior on Windows platforms, and it should not be attempted. If a design depends on this behavior, and the design cannot be altered, it may be necessary to claim a deviation from this rule after completing an appropriate risk analysis.

The reason for this is that Windows is a nonpersistent platform as discussed above. Just before calling the current handler function, Windows resets the handler for the next occurrence of the same signal to `SIG_DFL`. If the handler calls `signal()` to reinstall itself, there is still a race window. A signal might occur between the start of the handler and the call to `signal()`, which would invoke the default behavior instead of the desired handler.

Exceptions

SIG34-C-EX1: For implementations with persistent signal handlers, it is safe for a handler to modify the behavior of its own signal. Behavior modifications include ignoring the signal, resetting to the default behavior, and having the signal handled by a different handler. A handler reasserting its binding is also safe but unnecessary.

The following code example resets a signal handler to the system's default behavior:

```

#include <signal.h>

void handler(int signum) {
    #if !defined(_WIN32)
    if (signal(signum, SIG_DFL) == SIG_ERR) {
        /* Handle error */
    }
    #endif
    /* Handle signal */
}

void func(void) {
    if (signal(SIGUSR1, handler) == SIG_ERR) {
        /* Handle error */
    }
}

```

Risk Assessment

Two signals in quick succession can trigger a race condition on nonpersistent platforms, causing the signal's default behavior despite a handler's attempt to override it.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SIG34-C	Low	Unlikely	Low	P3	L3

Related Guidelines

CERT C Secure Coding Standard	SIG01-C. Understand implementation-specific details regarding signal handler persistence
ISO/IEC TS 17961:2013	Calling signal from interruptible signal handlers [sigcall]
MITRE CWE	CWE-479, Signal Handler Use of a Non-reentrant Function

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/r1Dp>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
signal_call_in_interruptible_signal_handler	Do not call signal() from within interruptible signal handlers.

CertC-SIG35

Do not return from a computational exception signal handler.

Input: IR

Source languages: C, C++

Details

According to the C Standard, 7.14.1.1 [[ISO/IEC 9899:2011](#)], if a signal handler returns when it has been entered as a result of a computational exception (that is, with the value of its argument of SIGFPE, SIGILL, SIGSEGV, or any other implementation-defined value corresponding to such an exception) returns, then the behavior is undefined. (See [undefined behavior 130](#).)

The Portable Operating System Interface (POSIX®), Base Specifications, Issue 7 [[IEEE Std 1003.1:2013](#)], adds SIGBUS to the list of computational exception signal handlers:

The behavior of a process is undefined after it returns normally from a signal-catching function for a SIGBUS, SIGFPE, SIGILL, or SIGSEGV signal that was not generated by kill(), sigqueue(), or raise().

Do not return from SIGFPE, SIGILL, SIGSEGV, or any other implementation-defined value corresponding to a computational exception, such as SIGBUS on POSIX systems, regardless of how the signal was generated.

Noncompliant Code Example

In this noncompliant code example, the division operation has undefined behavior if denom equals 0. (See [INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors](#)) and may result in a SIGFPE signal to the program.)

```
#include <errno.h>
#include <limits.h>
#include <signal.h>
#include <stdlib.h>

volatile sig_atomic_t denom;

void sighandle(int s) {
    /* Fix the offending volatile */
    if (denom == 0) {
        denom = 1;
    }
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        return 0;
    }

    char *end = NULL;
    long temp = strtol(argv[1], &end, 10);

    if (end == argv[1] || 0 != *end ||
        ((LONG_MIN == temp || LONG_MAX == temp) && errno == ERANGE)) {
        /* Handle error */
    }

    denom = (sig_atomic_t)temp;
    signal(SIGFPE, sighandle);

    long result = 100 / (long)denom;
    return 0;
}
```

When compiled with some implementations, this noncompliant code example will loop infinitely if given the input 0. It illustrates that even when a SIGFPE handler attempts to fix the error condition while obeying all other rules of signal handling, the program still does not behave as expected.

Compliant Solution

The only portably safe way to leave a SIGFPE, SIGILL, or SIGSEGV handler is to invoke abort(), quick_exit(), or _Exit(). In the case of SIGFPE, the default action is abnormal termination, so no user-defined handler is required:

```
#include <errno.h>
#include <limits.h>
#include <signal.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        return 0;
    }

    char *end = NULL;
    long denom = strtol(argv[1], &end, 10);
```

```

if (end == argv[1] || 0 != *end ||
    ((LONG_MIN == denom || LONG_MAX == denom) && errno == ERANGE)) {
    /* Handle error */
}

long result = 100 / denom;
return 0;
}

```

Implementation Details

Some implementations define useful behavior for programs that return from one or more of these signal handlers. For example, Solaris provides the `sigfpe()` function specifically to set a `SIGFPE` handler that a program may safely return from. Oracle also provides platform-specific computational exceptions for the `SIGTRAP`, `SIGBUS`, and `SIGEMT` signals. Finally, GNU libsigsegv takes advantage of the ability to return from a `SIGSEGV` handler to implement page-level memory management in user mode.

Risk Assessment

Returning from a computational exception signal handler is [undefined behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SIG35-C	Low	Unlikely	High	P1	L3

Bibliography

[IEEE Std 1003.1:2013]	2.4.1, Signal Generation and Delivery
[ISO/IEC 9899:2011]	Subclause 7.14.1.1, "The <code>signal</code> Function"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/QgGRAg>], Copyright [C] 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	1
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high
signal_names	List of signal names that represent computational exceptions.	('SIGBUS', 'SIGFPE', 'SIGILL', 'SIGSEGV')

Possible Messages

Name	Message
noreturn_computational_exception	Do not return from a computational exception signal handler.

CertC-ERR30

Set `errno` to zero before calling a library function known to set `errno`, and check `errno` only after the function returns a value indicating failure.

Input: IR

Source languages: C, C++

Details

The value of `errno` is initialized to zero at program startup, but it is never subsequently set to zero by any C standard library function. The value of `errno` may be set to nonzero by a C standard library function call whether or not there is an error, provided the use of `errno` is not documented in the description of the function. It is meaningful for a program to inspect the contents of `errno` only after an error might have occurred. More precisely, `errno` is meaningful only after a library function that sets `errno` on error has returned an error code.

According to Question 20.4 of C-FAQ [[Summit 2005](#)],

In general, you should detect errors by checking return values, and use `errno` only to distinguish among the various causes of an error, such as "File not found" or "Permission denied." (Typically, you use `perror` or `strerror` to print these discriminating error messages.) It's only necessary to detect errors with `errno` when a function does not have a unique, unambiguous, out-of-band error return (that is, because all of its possible

return values are valid; one example is `atoi` [*sic*]). In these cases (and in these cases only; check the documentation to be sure whether a function allows this), you can detect errors by setting `errno` to 0, calling the function, and then testing `errno`. (Setting `errno` to 0 first is important, as no library function ever does that for you.)

Note that `atoi()` is not required to set the value of `errno`.

Library functions fall into the following categories:

- Those that set `errno` and return and [out-of-band error indicator](#)
- Those that set `errno` and return and [in-band error indicator](#)
- Those that do not promise to set `errno`
- Those with differing standards documentation

Library Functions that Set `errno` and Return an Out-of-Band Error Indicator

The C Standard specifies that the functions listed in the following table set `errno` and return an [out-of-band error indicator](#). That is, their return value on error can never be returned by a successful call.

A program may set and check `errno` for these library functions but is not required to do so. The program should not check the value of `errno` without first verifying that the function returned an error indicator. For example, `errno` should not be checked after calling `signal()` without first ensuring that `signal()` actually returned `SIG_ERR`.

Functions That Set `errno` and Return an Out-of-Band Error Indicator

Function Name	Return Value	<code>errno</code> Value
<code>fgetpos()</code> , <code>fsetpos()</code>	-1L	Positive
<code>mbrtowc()</code> , <code>mbsrtowcs()</code>	(<code>size_t</code>) (-1)	EILSEQ
<code>signal()</code>	<code>SIG_ERR</code>	Positive
<code>wcrtomb()</code> , <code>wcsrtombs()</code>	(<code>size_t</code>) (-1)	EILSEQ
<code>mbrtoc16()</code> , <code>mbrtoc32()</code>	(<code>size_t</code>) (-1)	EILSEQ
<code>c16rtomb()</code> , <code>cr32rtomb()</code>	(<code>size_t</code>) (-1)	EILSEQ

Library Functions that Set `errno` and Return an In-Band Error Indicator

The C Standard specifies that the functions listed in the following table set `errno` and return an [in-band error indicator](#). That is, the return value when an error occurs is also a valid return value for successful calls. For example, the `strtoul()` function returns `ULONG_MAX` and sets `errno` to `ERANGE` if an error occurs. Because `ULONG_MAX` is a valid return value, `errno` must be used to check whether an error actually occurred. A program that uses `errno` for error checking must set it to 0 before calling one of these library functions and then inspect `errno` before a subsequent library function call.

The `fgetwc()` and `fputwc()` functions return `WEOF` in multiple cases, only one of which results in setting `errno`. The string conversion functions will return the maximum or minimum representable value and set `errno` to `ERANGE` if the converted value cannot be represented by the data type. However, if the conversion cannot happen because the input is invalid, the function will return 0, and the output pointer parameter will be assigned the value of the input pointer parameter, provided the output parameter is non-null.

Functions that Set `errno` and Return an In-Band Error Indicator

Function Name	Return Value	<code>errno</code> Value
<code>fgetwc()</code> , <code>fputwc()</code>	<code>WEOF</code>	EILSEQ
<code>strtol()</code> , <code>wcstol()</code>	<code>LONG_MIN</code> OR <code>LONG_MAX</code>	ERANGE
<code>strtoll()</code> , <code>wcstoll()</code>	<code>LLONG_MIN</code> OR <code>LLONG_MAX</code>	ERANGE
<code>strtoul()</code> , <code>wcstoul()</code>	<code>ULONG_MAX</code>	ERANGE
<code>strtoull()</code> , <code>wcstoull()</code>	<code>ULLONG_MAX</code>	ERANGE
<code>strtoimax()</code> , <code>wcstoimax()</code>	<code>UINTMAX_MAX</code>	ERANGE
<code>strtod()</code> , <code>wcstod()</code>	0 OR $\pm\text{HUGE_VAL}$	ERANGE
<code>strtof()</code> , <code>wcstof()</code>	0 OR $\pm\text{HUGE_VALF}$	ERANGE
<code>strtold()</code> , <code>wcstold()</code>	0 OR $\pm\text{HUGE_VALL}$	ERANGE
<code>strtoimax()</code> , <code>wcstoimax()</code>	<code>INTMAX_MIN</code> , <code>INTMAX_MAX</code>	ERANGE

Library Functions that Do Not Promise to Set `errno`

The C Standard fails to document the behavior of `errno` for some functions. For example, the `setlocale()` function normally returns a null pointer in the event of an error, but no guarantees are made about setting `errno`.

After calling one of these functions, a program should not rely solely on the value of `errno` to determine if an error occurred. The function might have altered `errno`, but this does not ensure that `errno` will properly indicate an error condition.

Library Functions with Differing Standards Documentation

Some functions behave differently regarding `errno` in various standards. The `fopen()` function is one such example. When `fopen()` encounters an error, it returns a null pointer. The C Standard makes no mention of `errno` when describing `fopen()`. However, POSIX.1 declares that when `fopen()` encounters an error, it returns a null pointer and sets `errno` to a value indicating the error [[IEEE Std 1003.1-2013](#)]. The implication is that a program conforming to C but not to POSIX (such as a Windows program) should not check `errno` after calling `fopen()`, but a POSIX program may check `errno` if `fopen()` returns a null pointer.

Library Functions and `errno`

The following uses of `errno` are documented in the C Standard:

- Functions defined in `<complex.h>` may set `errno` but are not required to.
- For numeric conversion functions in the `strtod`, `strtol`, `wcstod`, and `wcstol` families, if the correct result is outside the range of representable values, an appropriate minimum or maximum value is returned and the value `ERANGE` is stored in `errno`. For floating-point conversion functions in the `strtod` and `wcstod` families, if an underflow occurs, whether `errno` acquires the value `ERANGE` is [implementation-defined](#). If the conversion fails, `0` is returned and `errno` is not set.
- The numeric conversion function `atof()` and those in the `atoi` family "need not affect the value of" `errno`.
- For mathematical functions in `<math.h>`, if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, on a domain error, `errno` acquires the value `EDOM`; on an overflow with default rounding or if the mathematical result is an exact infinity from finite arguments, `errno` acquires the value `ERANGE`; and on an underflow, whether `errno` acquires the value `ERANGE` is implementation-defined.
- If a request made by calling `signal()` cannot be honored, a value of `SIG_ERR` is returned and a positive value is stored in `errno`.
- The byte I/O functions, wide-character I/O functions, and multibyte conversion functions store the value of the macro `EILSEQ` in `errno` if and only if an encoding error occurs.
- On failure, `fgetpos()` and `fsetpos()` return nonzero and store an [implementation-defined](#) positive value in `errno`.
- On failure, `ftell()` returns `-1L` and stores an [implementation-defined](#) positive value in `errno`.
- The `perror()` function maps the error number in `errno` to a message and writes it to `stderr`.

The POSIX.1 standard defines the use of `errno` by many more functions (including the C standard library function). POSIX also has a small set of functions that are exceptions to the rule. These functions have no return value reserved to indicate an error, but they still set `errno` on error. To detect an error, an application must set `errno` to `0` before calling the function and check whether it is nonzero after the call. Affected functions include `strcoll()`, `strxfrm()`, `strerror()`, `wcscol()`, `wcsxfrm()`, and `fwide()`. The C Standard allows these functions to set `errno` to a nonzero value on success. Consequently, this type of error checking should be performed only on POSIX systems.

Noncompliant Code Example (`strtoul()`)

This noncompliant code example fails to set `errno` to `0` before invoking `strtoul()`. If an error occurs, `strtoul()` returns a valid value (`ULONG_MAX`), so `errno` is the only means of determining if `strtoul()` ran successfully.

```
#include <errno.h>
#include <limits.h>
#include <stdlib.h>

void func(const char *c_str) {
    unsigned long number;
    char *endptr;

    number = strtoul(c_str, &endptr, 0);
    if (endptr == c_str || (number == ULONG_MAX
                           && errno == ERANGE)) {
        /* Handle error */
    } else {
        /* Computation succeeded */
    }
}
```

Any error detected in this manner may have occurred earlier in the program or may not represent an actual error.

Compliant Solution (`strtoul()`)

This compliant solution sets `errno` to `0` before the call to `strtoul()` and inspects `errno` after the call:

```
#include <errno.h>
#include <limits.h>
#include <stdlib.h>

void func(const char *c_str) {
    unsigned long number;
    char *endptr;

    errno = 0;
    number = strtoul(c_str, &endptr, 0);
    if (endptr == c_str || (number == ULONG_MAX
                           && errno == ERANGE)) {
        /* Handle error */
    } else {
        /* Computation succeeded */
    }
}
```

Noncompliant Code Example (`fopen()`)

This noncompliant code example may fail to diagnose errors because `fopen()` might not set `errno` even if an error occurs:

```
#include <errno.h>
#include <stdio.h>

void func(const char *filename) {
    FILE *fileptr;

    errno = 0;
    fileptr = fopen(filename, "rb");
    if (errno != 0) {
        /* Handle error */
    }
}
```

Compliant Solution (`fopen()`, C)

The C Standard makes no mention of `errno` when describing `fopen()`. In this compliant solution, the results of the call to `fopen()` are used to determine failure and `errno` is not checked:

```
#include <stdio.h>

void func(const char *filename) {
    FILE *fileptr = fopen(filename, "rb");
    if (fileptr == NULL) {
        /* An error occurred in fopen() */
    }
}
```

Compliant Solution (`fopen()`, POSIX)

In this compliant solution, `errno` is checked only after an error has already been detected by another means:

```
#include <errno.h>
#include <stdio.h>

void func(const char *filename) {
    FILE *fileptr;

    errno = 0;
    fileptr = fopen(filename, "rb");
    if (fileptr == NULL) {
        /*
         * An error occurred in fopen(); now it's valid
         * to examine errno.
         */
        perror(filename);
    }
}
```

Risk Assessment

The improper use of `errno` may result in failing to detect an error condition or in incorrectly identifying an error condition when none exists.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR30-C	Medium	Probable	Medium	P8	L2

Related Guidelines

CERT C Secure Coding Standard	EXP12-C. Do not ignore values returned by functions
ISO/IEC TS 17961:2013	Incorrectly setting and using <code>errno</code> [inverrno]
MITRE CWE	CWE-456 , Missing Initialization of a Variable

Bibliography

[Brainbell.com]	Macros and Miscellaneous Pitfalls
[Horton 1990]	Section 11, p. 168 Section 14, p. 254
[IEEE Std 1003.1-2013]	XSH, System Interfaces, <code>fopen</code>
[Koenig 1989]	Section 5.4, p. 73
[Summit 2005]	

Configuration

Name	Explanation	Value
errno_clearers		[]
errno_readers		['perror']
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
standard_posix		False
variant_cert		True

Possible Messages

Name	Message
misplaced_errno_check	errno should only be tested after a call to an errno-setting function.
missing_error_handling	The program should not check the value of errno without first verifying that the function returned an error indicator.
uncleared_errno	errno should be set to zero before calling an errno-setting function.

CertC-ERR32

Do not rely on indeterminate values of errno.

Input: IR

Source languages: C, C++

Details

According to the C Standard [[ISO/IEC 9899:2011](#)], the behavior of a program is [undefined](#) when

the value of `errno` is referred to after a signal occurred other than as the result of calling the `abort` or `raise` function and the corresponding signal handler obtained a `SIG_ERR` return from a call to the `signal` function.

See [undefined behavior 133](#).

A signal handler is allowed to call `signal()`; if that fails, `signal()` returns `SIG_ERR` and sets `errno` to a positive value. However, if the event that caused a signal was external (not the result of the program calling `abort()` or `raise()`), the only functions the signal handler may call are `_Exit()` or `abort()`, or it may call `signal()` on the signal currently being handled; if `signal()` fails, the value of `errno` is [indeterminate](#).

This rule is also a special case of [SIG31-C. Do not access shared objects in signal handlers](#). The object designated by `errno` is of static storage duration and is not a volatile `sig_atomic_t`. As a result, performing any action that would require `errno` to be set would normally cause [undefined behavior](#). The C Standard, 7.14.1.1, paragraph 5, makes a special exception for `errno` in this case, allowing `errno` to take on an indeterminate value but specifying that there is no other [undefined behavior](#). This special exception makes it possible to call `signal()` from within a signal handler without risking [undefined behavior](#), but the handler, and any code executed after the handler returns, must not depend on the value of `errno` being meaningful.

Noncompliant Code Example

The `handler()` function in this noncompliant code example attempts to restore default handling for the signal indicated by `signum`. If the request to set the signal to default can be honored, the `signal()` function returns the value of the signal handler for the most recent successful call to the `signal()` function for the specified signal. Otherwise, a value of `SIG_ERR` is returned and a positive value is stored in `errno`. Unfortunately, the value of `errno` is indeterminate because the `handler()` function is called when an external signal is raised, so any attempt to read `errno` (for example, by the `perror()` function) is [undefined behavior](#):

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

typedef void (*pfv)(int);

void handler(int signum) {
    pfv old_handler = signal(signum, SIG_DFL);
    if (old_handler == SIG_ERR) {
        perror("SIGINT handler"); /* Undefined behavior */
    }
}
```

```

    /* Handle error */
}

int main(void) {
    pfv old_handler = signal(SIGINT, handler);
    if (old_handler == SIG_ERR) {
        perror("SIGINT handler");
        /* Handle error */
    }

    /* Main code loop */

    return EXIT_SUCCESS;
}

```

The call to `perror()` from `handler()` also violates [SIG30-C. Call only asynchronous-safe functions within signal handlers.](#)

Compliant Solution

This compliant solution does not reference `errno` and does not return from the signal handler if the `signal()` call fails:

```

#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

typedef void (*pfv)(int);

void handler(int signum) {
    pfv old_handler = signal(signum, SIG_DFL);
    if (old_handler == SIG_ERR) {
        abort();
    }
}

int main(void) {
    pfv old_handler = signal(SIGINT, handler);
    if (old_handler == SIG_ERR) {
        perror("SIGINT handler");
        /* Handle error */
    }

    /* Main code loop */

    return EXIT_SUCCESS;
}

```

Noncompliant Code Example (POSIX)

POSIX is less restrictive than C about what applications can do in signal handlers. It has a long list of [asynchronous-safe](#) functions that can be called. (See [SIG30-C. Call only asynchronous-safe functions within signal handlers.](#)) Many of these functions set `errno` on error, which can lead to a signal handler being executed between a call to a failed function and the subsequent inspection of `errno`. Consequently, the value inspected is not the one set by that function but the one set by a function call in the signal handler. POSIX applications can avoid this problem by ensuring that signal handlers containing code that might alter `errno`; always save the value of `errno` on entry and restore it before returning.

The signal handler in this noncompliant code example alters the value of `errno`. As a result, it can cause incorrect error handling if executed between a failed function call and the subsequent inspection of `errno`:

```

#include <signal.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>

void reaper(int signum) {
    errno = 0;
    for (;;) {
        int rc = waitpid(-1, NULL, WNOHANG);
        if ((0 == rc) || (-1 == rc && EINTR != errno)) {
            break;
        }
    }
    if (ECHILD != errno) {
        /* Handle error */
    }
}

int main(void) {
    struct sigaction act;
    act.sa_handler = reaper;
    act.sa_flags = 0;
    if (sigemptyset(&act.sa_mask) != 0) {
        /* Handle error */
    }
    if (sigaction(SIGCHLD, &act, NULL) != 0) {
        /* Handle error */
    }

    /* ... */

    return EXIT_SUCCESS;
}

```

Compliant Solution (POSIX)

This compliant solution saves and restores the value of `errno` in the signal handler:

```
#include <signal.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>

void reaper(int signum) {
    errno_t save_errno = errno;
    errno = 0;
    for (;;) {
        int rc = waitpid(-1, NULL, WNOHANG);
        if ((0 == rc) || (-1 == rc && EINTR != errno)) {
            break;
        }
    }
    if (ECHILD != errno) {
        /* Handle error */
    }
    errno = save_errno;
}

int main(void) {
    struct sigaction act;
    act.sa_handler = reaper;
    act.sa_flags = 0;
    if (sigemptyset(&act.sa_mask) != 0) {
        /* Handle error */
    }
    if (sigaction(SIGCHLD, &act, NULL) != 0) {
        /* Handle error */
    }
    /* ... */
    return EXIT_SUCCESS;
}
```

Risk Assessment

Referencing indeterminate values of `errno` is [undefined behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR32-C	Low	Unlikely	Low	P3	L3

Related Guidelines

CERT C Secure Coding Standard	SIG30-C. Call only asynchronous-safe functions within signal handlers SIG31-C. Do not access shared objects in signal handlers
-------------------------------	---

Bibliography

[ISO/IEC 9899:2011]	Subclause 7.14.1.1, "The <code>signal</code> Function"
---------------------	--

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/NABJ>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
errno_readers		[' perror']
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
errno_referred_in_signal_handler	Do not rely on <code>errno</code> value in signal handler.

CertC-ERR33

Detect and handle standard library errors.

Input: IR

Source languages: C, C++

Details

The majority of the standard library functions, including I/O functions and memory allocation functions, return either a valid value or a value of the correct return type that indicates an error (for example, -1 or a null pointer). Assuming that all calls to such functions will succeed and failing to check the return value for an indication of an error is a dangerous practice that may lead to [unexpected](#) or [undefined behavior](#) when an error occurs. It is essential that programs detect and appropriately handle all errors in accordance with an error-handling policy, as discussed in [ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy](#).

The successful completion or failure of each of the standard library functions listed in the following table shall be determined either by comparing the function's return value with the value listed in the column labeled "Error Return" or by calling one of the library functions mentioned in the footnotes.

Standard Library Functions

Function	Successful Return	Error Return
aligned_alloc()	Pointer to space	NULL
asctime_s()	0	Nonzero
at_quick_exit()	0	Nonzero
atexit()	0	Nonzero
bsearch()	Pointer to matching element	NULL
bsearch_s()	Pointer to matching element	NULL
btowc()	Converted wide character	WEOF
c16rtomb()	Number of bytes	(size_t) (-1)
c32rtomb()	Number of bytes	(size_t) (-1)
calloc()	Pointer to space	NULL
clock()	Processor time	(clock_t) (-1)
cond_broadcast()	thrd_success	thrd_error
cond_init()	thrd_success	thrd_nomem OR thrd_error
cond_signal()	thrd_success	thrd_error
cond_timedwait()	thrd_success	thrd_timedout OR thrd_error
cond_wait()	thrd_success	thrd_error
ctime_s()	0	Nonzero
fclose()	0	EOF (negative)
fflush()	0	EOF (negative)
fgetc()	Character read	EOF ¹
fgetpos()	0	Nonzero, errno > 0
fgets()	Pointer to string	NULL
fgetwc()	Wide character read	WEOF ¹
fopen()	Pointer to stream	NULL
fopen_s()	0	Nonzero
fprintf()	Number of characters (nonnegative)	Negative
fprintf_s()	Number of characters (nonnegative)	Negative
fputc()	Character written	EOF ²

fputs()	Nonnegative	EOF (negative)
fputwc()	Wide character written	WEOF
fputws()	Nonnegative	EOF (negative)
fread()	Elements read	Elements read
freopen()	Pointer to stream	NULL
freopen_s()	0	Nonzero
fscanf()	Number of conversions (nonnegative)	EOF (negative)
fscanf_s()	Number of conversions (nonnegative)	EOF (negative)
fseek()	0	Nonzero
fsetpos()	0	Nonzero, errno > 0
ftell()	File position	-1L, errno > 0
fwprintf()	Number of wide characters (nonnegative)	Negative
fwprintf_s()	Number of wide characters (nonnegative)	Negative
fwrite()	Elements written	Elements written
fwscanf()	Number of conversions (nonnegative)	EOF (negative)
fwscanf_s()	Number of conversions (nonnegative)	EOF (negative)
getc()	Character read	EOF ¹
getchar()	Character read	EOF ¹
getenv()	Pointer to string	NULL
getenv_s()	Pointer to string	NULL
gets_s()	Pointer to string	NULL
getwc()	Wide character read	WEOF
getwchar()	Wide character read	WEOF
gmtime()	Pointer to broken-down time	NULL
gmtime_s()	Pointer to broken-down time	NULL
localtime()	Pointer to broken-down time	NULL
localtime_s()	Pointer to broken-down time	NULL
malloc()	Pointer to space	NULL
mblen(), s != NULL	Number of bytes	-1
mbrlen(), s != NULL	Number of bytes or status	(size_t) (-1)
mbrtoc16()	Number of bytes or status	(size_t) (-1), errno == EILSEQ
mbrtoc32()	Number of bytes or status	(size_t) (-1), errno == EILSEQ
mbtowc(), s != NULL	Number of bytes or status	(size_t) (-1), errno == EILSEQ
mbsrtowcs()	Number of non-null elements	(size_t) (-1), errno == EILSEQ
mbsrtowcs_s()	0	Nonzero
mbstowcs()	Number of non-null elements	(size_t) (-1)
mbstowcs_s()	0	Nonzero
mbtowc(), s != NULL	Number of bytes	-1

memchr()	Pointer to located character	NULL
mkttime()	Calendar time	(time_t)(-1)
mtx_init()	thrd_success	thrd_error
mtx_lock()	thrd_success	thrd_error
mtx_timedlock()	thrd_success	thrd_timedout OR thrd_error
mtx_trylock()	thrd_success	thrd_busy OR thrd_error
mtx_unlock()	thrd_success	thrd_error
printf_s()	Number of characters (nonnegative)	Negative
putc()	Character written	EOF ²
putwc()	Wide character written	WEOF
raise()	0	Nonzero
realloc()	Pointer to space	NULL
remove()	0	Nonzero
rename()	0	Nonzero
setlocale()	Pointer to string	NULL
setvbuf()	0	Nonzero
scanf()	Number of conversions (nonnegative)	EOF (negative)
scanf_s()	Number of conversions (nonnegative)	EOF (negative)
signal()	Pointer to previous function	SIG_ERR, errno > 0
snprintf()	Number of characters that would be written (nonnegative)	Negative
snprintf_s()	Number of characters that would be written (nonnegative)	Negative
sprintf()	Number of non-null characters written	Negative
sprintf_s()	Number of non-null characters written	Negative
sscanf()	Number of conversions (nonnegative)	EOF (negative)
sscanf_s()	Number of conversions (nonnegative)	EOF (negative)
strchr()	Pointer to located character	NULL
strerror_s()	0	Nonzero
strftime()	Number of non-null characters	0
strupbrk()	Pointer to located character	NULL
strrchr()	Pointer to located character	NULL
strstr()	Pointer to located string	NULL
strtod()	Converted value	0, errno == ERANGE
strtodf()	Converted value	0, errno == ERANGE
strtoimax()	Converted value	INTMAX_MAX OR INTMAX_MIN, errno == ERANGE
strtok()	Pointer to first character of a token	NULL
strtok_s()	Pointer to first character of a token	NULL
strtol()	Converted value	LONG_MAX OR LONG_MIN, errno == ERANGE
strtold()	Converted value	0, errno == ERANGE

<code>strtoll()</code>	Converted value	<code>LLONG_MAX OR LLONG_MIN, errno == ERANGE</code>
<code>strtoumax()</code>	Converted value	<code>UINTMAX_MAX, errno == ERANGE</code>
<code>strtoul()</code>	Converted value	<code>ULONG_MAX, errno == ERANGE</code>
<code>strtoull()</code>	Converted value	<code>ULLONG_MAX, errno == ERANGE</code>
<code>strxfrm()</code>	Length of transformed string	<code>>= n</code>
<code>swprintf()</code>	Number of non-null wide characters	Negative
<code>swprintf_s()</code>	Number of non-null wide characters	Negative
<code>swscanf()</code>	Number of conversions (nonnegative)	<code>EOF (negative)</code>
<code>swscanf_s()</code>	Number of conversions (nonnegative)	<code>EOF (negative)</code>
<code>thrd_create()</code>	<code>thrd_success</code>	<code>thrd_nomem OR thrd_error</code>
<code>thrd_detach()</code>	<code>thrd_success</code>	<code>thrd_error</code>
<code>thrd_join()</code>	<code>thrd_success</code>	<code>thrd_error</code>
<code>thrd_sleep()</code>	0	Negative
<code>time()</code>	Calendar time	<code>(time_t)(-1)</code>
<code>timespec_get()</code>	Base	0
<code>tmpfile()</code>	Pointer to stream	<code>NULL</code>
<code>tmpfile_s()</code>	0	Nonzero
<code>tmpnam()</code>	Non-null pointer	<code>NULL</code>
<code>tmpnam_s()</code>	0	Nonzero
<code>tss_create()</code>	<code>thrd_success</code>	<code>thrd_error</code>
<code>tss_get()</code>	Value of thread-specific storage	0
<code>tss_set()</code>	<code>thrd_success</code>	<code>thrd_error</code>
<code>ungetc()</code>	Character pushed back	<code>EOF (see below)</code>
<code>ungetwc()</code>	Character pushed back	<code>WEOF</code>
<code>vfprintf()</code>	Number of characters (nonnegative)	Negative
<code>vfprintf_s()</code>	Number of characters (nonnegative)	Negative
<code>vfscanf()</code>	Number of conversions (nonnegative)	<code>EOF (negative)</code>
<code>vfscanf_s()</code>	Number of conversions (nonnegative)	<code>EOF (negative)</code>
<code>vwprintf()</code>	Number of wide characters (nonnegative)	Negative
<code>vwprintf_s()</code>	Number of wide characters (nonnegative)	Negative
<code>vwscanf()</code>	Number of conversions (nonnegative)	<code>EOF (negative)</code>
<code>vwscanf_s()</code>	Number of conversions (nonnegative)	<code>EOF (negative)</code>
<code>vprintf_s()</code>	Number of characters (nonnegative)	Negative
<code>vscanf()</code>	Number of conversions (nonnegative)	<code>EOF (negative)</code>
<code>vscanf_s()</code>	Number of conversions (nonnegative)	<code>EOF (negative)</code>
<code>vsnprintf()</code>	Number of characters that would be written (nonnegative)	Negative
<code>vsnprintf_s()</code>	Number of characters that would be written (nonnegative)	Negative
<code>vsprintf()</code>	Number of non-null characters (nonnegative)	Negative

<code>vsnprintf_s()</code>	Number of non-null characters (nonnegative)	Negative
<code>vsscanf()</code>	Number of conversions (nonnegative)	<code>EOF</code> (negative)
<code>vscanf_s()</code>	Number of conversions (nonnegative)	<code>EOF</code> (negative)
<code>vswprintf()</code>	Number of non-null wide characters	Negative
<code>vswprintf_s()</code>	Number of non-null wide characters	Negative
<code>vswscanf()</code>	Number of conversions (nonnegative)	<code>EOF</code> (negative)
<code>vswscanf_s()</code>	Number of conversions (nonnegative)	<code>EOF</code> (negative)
<code>vwprintf_s()</code>	Number of wide characters (nonnegative)	Negative
<code>vwscanf()</code>	Number of conversions (nonnegative)	<code>EOF</code> (negative)
<code>vwscanf_s()</code>	Number of conversions (nonnegative)	<code>EOF</code> (negative)
<code>wcrtomb()</code>	Number of bytes stored	<code>(size_t) (-1)</code>
<code>wcschr()</code>	Pointer to located wide character	NULL
<code>wcsftime()</code>	Number of non-null wide characters	0
<code>wcspbrk()</code>	Pointer to located wide character	NULL
<code>wcsrchr()</code>	Pointer to located wide character	NULL
<code>wcsrtombs()</code>	Number of non-null bytes	<code>(size_t) (-1), errno == EILSEQ</code>
<code>wcsrtombs_s()</code>	0	Nonzero
<code>wcsstr()</code>	Pointer to located wide string	NULL
<code>wcstod()</code>	Converted value	<code>0, errno == ERANGE</code>
<code>wcstof()</code>	Converted value	<code>0, errno == ERANGE</code>
<code>wcstoiimax()</code>	Converted value	<code>INTMAX_MAX OR INTMAX_MIN, errno == ERANGE</code>
<code>wcstok()</code>	Pointer to first wide character of a token	NULL
<code>wcstok_s()</code>	Pointer to first wide character of a token	NULL
<code>wcstol()</code>	Converted value	<code>LONG_MAX OR LONG_MIN, errno == ERANGE</code>
<code>wcstold()</code>	Converted value	<code>0, errno == ERANGE</code>
<code>wcstoll()</code>	Converted value	<code>LLONG_MAX OR LLONG_MIN, errno == ERANGE</code>
<code>wcstombs()</code>	Number of non-null bytes	<code>(size_t) (-1)</code>
<code>wcstombs_s()</code>	0	Nonzero
<code>wcstoumax()</code>	Converted value	<code>UINTMAX_MAX, errno == ERANGE</code>
<code>wcstoul()</code>	Converted value	<code>ULONG_MAX, errno == ERANGE</code>
<code>wcstoull()</code>	Converted value	<code>ULLONG_MAX, errno == ERANGE</code>
<code>wcsxfrm()</code>	Length of transformed wide string	<code>>= n</code>
<code>wctob()</code>	Converted character	<code>EOF</code>
<code>wctomb(s, s != NULL)</code>	Number of bytes stored	-1
<code>wctomb_s(s, s != NULL)</code>	Number of bytes stored	-1
<code>wctrans()</code>	Valid argument to <code>towctrans</code>	0
<code>wctype()</code>	Valid argument to <code>iswctype</code>	0
<code>wmemchr()</code>	Pointer to located wide character	NULL

wprintf_s()	Number of wide characters (nonnegative)	Negative
wscanf()	Number of conversions (nonnegative)	EOF (negative)
wscanf_s()	Number of conversions (nonnegative)	EOF (negative)

Note: According to [F1035-C. Use feof\(\) and perror\(\) to detect end-of-file and file errors when sizeof\(int\) == sizeof\(char\)](#), callers should verify end-of-file and file errors for the functions in this table as follows:

¹ By calling perror() and feof()

² By calling perror()

The ungetc() function does not set the error indicator even when it fails, so it is not possible to check for errors reliably unless it is known that the argument is not equal to EOF. The C Standard [\[ISO/IEC 9899:2011\]](#) states that "one character of pushback is guaranteed," so this should not be an issue if, at most, one character is ever pushed back before reading again. (See [F1013-C. Never push back anything other than one read character](#).)

Noncompliant Code Example {setlocale()}

In this noncompliant code example, the function utf8_to_wcs() attempts to convert a sequence of UTF-8 characters to wide characters. It first invokes setlocale() to set the global locale to the implementation-defined en_US.UTF-8 but does not check for failure. The setlocale() function will fail by returning a null pointer, for example, when the locale is not installed. The function may fail for other reasons as well, such as the lack of resources. Depending on the sequence of characters pointed to by utf8, the subsequent call to mbstowcs() may fail or result in the function storing an unexpected sequence of wide characters in the supplied buffer wcs.

```
#include <locale.h>
#include <stdlib.h>

int utf8_to_wcs(wchar_t *wcs, size_t n, const char *utf8,
                size_t *size) {
    if (NULL == size) {
        return -1;
    }
    setlocale(LC_CTYPE, "en_US.UTF-8");
    *size = mbstowcs(wcs, utf8, n);
    return 0;
}
```

Compliant Solution {setlocale()}

This compliant solution checks the value returned by setlocale() and avoids calling mbstowcs() if the function fails. The function also takes care to restore the locale to its initial setting before returning control to the caller.

```
#include <locale.h>
#include <stdlib.h>

int utf8_to_wcs(wchar_t *wcs, size_t n, const char *utf8,
                size_t *size) {
    if (NULL == size) {
        return -1;
    }
    const char *save = setlocale(LC_CTYPE, "en_US.UTF-8");
    if (NULL == save) {
        return -1;
    }

    *size = mbstowcs(wcs, utf8, n);
    if (NULL == setlocale(LC_CTYPE, save)) {
        return -1;
    }
    return 0;
}
```

Noncompliant Code Example {calloc()}

In this noncompliant code example, temp_num, tmp2, and num_of_records are derived from a [tainted source](#). Consequently, an attacker can easily cause calloc() to fail by providing a large value for num_of_records.

```
#include <stdlib.h>
#include <string.h>

enum { SIG_DESC_SIZE = 32 };

typedef struct {
    char sig_desc[SIG_DESC_SIZE];
} signal_info;

void func(size_t num_of_records, size_t temp_num,
          const char *tmp2, size_t tmp2_size_bytes) {
    signal_info *start = (signal_info *)calloc(num_of_records,
                                                sizeof(signal_info));

    if (tmp2 == NULL) {
        /* Handle error */
    } else if (temp_num > num_of_records) {
        /* Handle error */
    } else if (tmp2_size_bytes < SIG_DESC_SIZE) {
        /* Handle error */
    }
}
```

```

    signal_info *point = start + temp_num - 1;
    memcpy(point->sig_desc, tmp2, SIG_DESC_SIZE);
    point->sig_desc[SIG_DESC_SIZE - 1] = '\0';
    /* ... */
    free(start);
}

```

When `calloc()` fails, it returns a null pointer that is assigned to `start`. If `start` is null, an attacker can provide a value for `temp_num` that, when scaled by `sizeof(signal_info)`, references a writable address to which control is eventually transferred. The contents of the string referenced by `tmp2` can then be used to overwrite the address, resulting in an arbitrary code execution [vulnerability](#).

Compliant Solution (`calloc()`)

To correct this error, ensure the pointer returned by `calloc()` is not null:

```

#include <stdlib.h>
#include <string.h>

enum { SIG_DESC_SIZE = 32 };

typedef struct {
    char sig_desc[SIG_DESC_SIZE];
} signal_info;

void func(size_t num_of_records, size_t temp_num,
          const char *tmp2, size_t tmp2_size_bytes) {
    signal_info *start = (signal_info *)calloc(num_of_records,
                                              sizeof(signal_info));

    if (start == NULL) {
        /* Handle allocation error */
    } else if (tmp2 == NULL) {
        /* Handle error */
    } else if (temp_num > num_of_records) {
        /* Handle error */
    } else if (tmp2_size_bytes < SIG_DESC_SIZE) {
        /* Handle error */
    }

    signal_info *point = start + temp_num - 1;
    memcpy(point->sig_desc, tmp2, SIG_DESC_SIZE);
    point->sig_desc[SIG_DESC_SIZE - 1] = '\0';
    /* ... */
    free(start);
}

```

Noncompliant Code Example (`realloc()`)

This noncompliant code example calls `realloc()` to resize the memory referred to by `p`. However, if `realloc()` fails, it returns a null pointer and the connection between the original block of memory and `p` is lost, resulting in a memory leak.

```

#include <stdlib.h>

void *p;
void func(size_t new_size) {
    if (new_size == 0) {
        /* Handle error */
    }
    p = realloc(p, new_size);
    if (p == NULL) {
        /* Handle error */
    }
}

```

This code example complies with [MEM04-C. Do not perform zero-length allocations](#).

Compliant Solution (`realloc()`)

In this compliant solution, the result of `realloc()` is assigned to the temporary pointer `q` and validated before it is assigned to the original pointer `p`:

```

#include <stdlib.h>

void *p;
void func(size_t new_size) {
    void *q;

    if (new_size == 0) {
        /* Handle error */
    }

    q = realloc(p, new_size);
    if (q == NULL) {
        /* Handle error */
    } else {
        p = q;
    }
}

```

Noncompliant Code Example (`fseek()`)

In this noncompliant code example, the `fseek()` function is used to set the file position to a location `offset` in the file referred to by `file` prior to reading a sequence of bytes from the file. However, if an I/O error occurs during the seek operation, the subsequent read will fill the buffer with the wrong contents.

```
#include <stdio.h>

size_t read_at(FILE *file, long offset,
                void *buf, size_t nbytes) {
    fseek(file, offset, SEEK_SET);
    return fread(buf, 1, nbytes, file);
}
```

Compliant Solution (`fseek()`)

According to the C Standard, the `fseek()` function returns a nonzero value to indicate that an error occurred. This compliant solution tests for this condition before reading from a file to eliminate the chance of operating on the wrong portion of the file if `fseek()` fails:

```
#include <stdio.h>

size_t read_at(FILE *file, long offset,
                void *buf, size_t nbytes) {
    if (fseek(file, offset, SEEK_SET) != 0) {
        /* Indicate error to caller */
        return 0;
    }
    return fread(buf, 1, nbytes, file);
}
```

Noncompliant Code Example (`snprintf()`)

In this noncompliant code example, `snprintf()` is assumed to succeed. However, if the call fails (for example, because of insufficient memory, as described in GNU libc bug [441945](#)), the subsequent call to `log_message()` has [undefined behavior](#) because the character buffer is uninitialized and need not be null-terminated.

```
#include <stdio.h>

extern void log_message(const char *);

void f(int i, int width, int prec) {
    char buf[40];
    snprintf(buf, sizeof(buf), "i = %.*i", width, prec, i);
    log_message(buf);
    /* ... */
}
```

Compliant Solution (`snprintf()`)

This compliant solution does not assume that `snprintf()` will succeed regardless of its arguments. It tests the return value of `snprintf()` before subsequently using the formatted buffer. This compliant solution also treats the case where the static buffer is not large enough for `snprintf()` to append the terminating null character as an error.

```
#include <stdio.h>
#include <string.h>

extern void log_message(const char *);

void f(int i, int width, int prec) {
    char buf[40];
    int n;
    n = snprintf(buf, sizeof(buf), "i = %.*i", width, prec, i);
    if (n < 0 || n >= sizeof(buf)) {
        /* Handle snprintf() error */
        strcpy(buf, "unknown error");
    }
    log_message(buf);
}
```

Compliant Solution (`snprintf(null)`)

If unknown, the length of the formatted string can be discovered by invoking `snprintf()` with a null buffer pointer to determine the size required for the output, then dynamically allocating a buffer of sufficient size, and finally calling `snprintf()` again to format the output into the dynamically allocated buffer. Even with this approach, the success of all calls still needs to be tested, and any errors must be appropriately handled. A possible optimization is to first attempt to format the string into a reasonably small buffer allocated on the stack and, only when the buffer turns out to be too small, dynamically allocate one of a sufficient size:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern void log_message(const char *);

void f(int i, int width, int prec) {
    char buffer[20];
    char *buf = buffer;
    int n = sizeof(buffer);
    const char fmt[] = "i = %.*i";
    n = snprintf(buf, n, fmt, width, prec, i);
    if (n < 0) {
```

```

/* Handle sprintf() error */
strcpy(buffer, "unknown error");
goto write_log;
}

if (n < sizeof(buffer)) {
    goto write_log;
}

buf = (char *)malloc(n + 1);
if (NULL == buf) {
    /* Handle malloc() error */
    strcpy(buffer, "unknown error");
    goto write_log;
}

n = snprintf(buf, n, fmt, width, prec, i);
if (n < 0) {
    /* Handle snprintf() error */
    strcpy(buffer, "unknown error");
}

write_log:
    log_message(buf);

    if (buf != buffer) {
        free(buf);
    }
}

```

This solution uses the `goto` statement, as suggested in [MEM12-C. Consider using a goto chain when leaving a function on error when using and releasing resources.](#)

Exceptions

ERR33-C-EX1: It is acceptable to ignore the return value of a function that cannot fail, or a function whose return value is inconsequential, or when an error condition need not be diagnosed. The function's results should be explicitly cast to `void` to signify programmer intent. Return values from the functions in the following table do not need to be checked because their historical use has overwhelmingly omitted error checking and the consequences are not relevant to security.

Functions for which Return Values Need Not Be Checked

Function	Successful Return	Error Return
<code>putchar()</code>	Character written	<code>EOF</code>
<code>putwchar()</code>	Wide character written	<code>WEOF</code>
<code>puts()</code>	Nonnegative	<code>EOF</code> (negative)
<code>printf()</code> , <code>vprintf()</code>	Number of characters (nonnegative)	Negative
<code>wprintf()</code> , <code>vwprintf()</code>	Number of wide characters (nonnegative)	Negative
<code>kill_dependency()</code>	The input parameter	NA
<code>memcpy()</code> , <code>wmemcpy()</code>	The destination input parameter	NA
<code>memmove()</code> , <code>wmemmove()</code>	The destination input parameter	NA
<code>strncpy()</code> , <code>wcsncpy()</code>	The destination input parameter	NA
<code>strcat()</code> , <code>wcsconcat()</code>	The destination input parameter	NA
<code>strncat()</code> , <code>wcsncat()</code>	The destination input parameter	NA
<code>memset()</code> , <code>wmemset()</code>	The destination input parameter	NA

Risk Assessment

Failing to detect error conditions can lead to unpredictable results, including [abnormal program termination](#) and [denial-of-service attacks](#) or, in some situations, could even allow an attacker to run arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR33-C	High	Likely	Medium	P18	L1

Related Guidelines

CERT C Secure Coding Standard	ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy EXP34-C. Do not dereference null pointers FI013-C. Never push back anything other than one read character MEM04-C. Do not perform zero-length allocations MEM12-C. Consider using a goto chain when leaving a function on error when using and releasing resources
SEI CERT C++ Coding Standard	ERR10-CPP. Check for error conditions FI004-CPP. Detect and handle input and output errors
ISO/IEC TS 17961:2013	Failing to detect and handle standard library errors [liberr]
MITRE CWE	CWE-252 , Unchecked Return Value CWE-253 , Incorrect Check of Function Return Value CWE-390 , Detection of Error Condition without Action CWE-391 , Unchecked Error Condition CWE-476 , NULL Pointer Dereference

Bibliography

[DHS 2006]	Handle All Errors Safely
[Henricson 1997]	Recommendation 12.1, "Check for All Errors Reported from Functions"
[ISO/IEC 9899:2011]	Subclause 7.21.7.10, "The <code>ungetc</code> Function"
[VU#159523]	

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/w4C4Ag>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
allowed_functions		frozenset(['strncat', 'memmove', 'memset', 'strcat', 'strcpy', 'memcpy', 'strncpy'])
errno_headers	Header names were 'errno', 'EILSEQ' and 'ERANGE' might be located.	['errno.h', 'errno-base.h']
function_argument_lookup	Function name to argument position mapping where argument and return value are forbidden to share the same variable.	dict{...}
functions	Allows to declare function names for which a check must exist. The check is expressed as an IR pattern.	dict{...}
inspect_template_instances	Whether calls in template instances should be reported.	False
known_check_functions	Collection of functions which are known to test return values of functions under test.	[]
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
stdio_headers	Header names were 'EOF' might be located.	['stdio.h', 'libio.h']
threads_headers	Header names were thread related macros and functions might be located.	['threads.h']
wchar_headers	Header names were 'WEOF' might be located.	['wchar.h', 'corecrt_wctype.h', '_wctype.h', '_types.h']

Possible Messages

Name	Message
discarded_return_with_entity	Return value of function discarded.
discarded_return_without_entity	Return value of function discarded.
possible_leak	Possible memory leak in function error case when overriding a pointer.
unhandled_return_value	Return value of function call not properly checked.

CertC-ERR34

Detect errors when converting a string to a number.

Input: IR

Source languages: C, C++

Details

The process of parsing an integer or floating-point number from a string can produce many errors. The string might not contain a number. It might contain a number of the correct type that is out of range (such as an integer that is larger than `INT_MAX`). The string may also contain extra information after the number, which may or may not be useful after the conversion. These error conditions must be detected and addressed when a string-to-number conversion is performed using a C Standard Library function.

The `strtol()`, `strtoll()`, `strtoimax()`, `strtoul()`, `strtoull()`, `strtoimax()`, `strtof()`, `strtod()`, and `strtold()` functions convert the initial portion of a null-terminated byte string to a `long int`, `long long int`, `intmax_t`, `unsigned long int`, `unsigned long long int`, `uintmax_t`, `float`, `double`, and `long double` representation, respectively.

Use one of the C Standard Library `strto*` functions to parse an integer or floating-point number from a string. These functions provide more robust error handling than alternative solutions. Also, use the `strtol()` function to convert to a smaller signed integer type such as `signed int`, `signed short`, and `signed char`, testing the result against the range limits for that type. Likewise, use the `strtoul()` function to convert to a smaller unsigned integer type such as `unsigned int`, `unsigned short`, and `unsigned char`, and test the result against the range limits for that type. These range tests do nothing if the smaller type happens to have the same size and representation for a particular implementation.

Noncompliant Code Example (`atoi()`)

This noncompliant code example converts the string token stored in the `buff` to a signed integer value using the `atoi()` function:

```
#include <stdlib.h>

void func(const char *buff) {
    int si;

    if (buff) {
        si = atoi(buff);
    } else {
        /* Handle error */
    }
}
```

The `atoi()`, `atol()`, `atoll()`, and `atof()` functions convert the initial portion of a string token to `int`, `long int`, `long long int`, and `double` representation, respectively. Except for the behavior on error, they are equivalent to

<code>atoi: (int)strtol(nptr, (char **)NULL, 10)</code>
<code>atol: strtol(nptr, (char **)NULL, 10)</code>
<code>atoll: strtoll(nptr, (char **)NULL, 10)</code>
<code>atof: strtod(nptr, (char **)NULL)</code>

Unfortunately, `atoi()` and related functions lack a mechanism for reporting errors for invalid values. Specifically, these functions:

- do not need to set `errno` on an error;
- have undefined behavior if the value of the result cannot be represented;
- return 0 (or 0.0) if the string does not represent an integer (or decimal), which is indistinguishable from a correctly formatted, zero-denoting input string.

Noncompliant Example (`sscanf()`)

This noncompliant example uses the `sscanf()` function to convert a string token to an integer. The `sscanf()` function has the same limitations as `atoi()`:

```
#include <stdio.h>

void func(const char *buff) {
    int matches;
    int si;

    if (buff) {
        matches = sscanf(buff, "%d", &si);
        if (matches != 1) {
            /* Handle error */
        }
    } else {
        /* Handle error */
    }
}
```

The `sscanf()` function returns the number of input items successfully matched and assigned, which can be fewer than provided for, or even 0 in the event of an early matching failure. However, `sscanf()` fails to report the other errors reported by `strtol()`, such as numeric overflow.

Compliant Solution (`strtol()`)

The `strtol()`, `strtoll()`, `strtoimax()`, `strtoul()`, `strtoull()`, `strtoumax()`, `strtof()`, `strtod()`, and `strtold()` functions convert a null-terminated byte string to `long int`, `long long int`, `intmax_t`, `unsigned long int`, `unsigned long long int`, `uintmax_t`, `float`, `double`, and `long double` representation, respectively.

This compliant solution uses `strtol()` to convert a string token to an integer and ensures that the value is in the range of `int`:

```
#include <errno.h>
#include <limits.h>
#include <stdlib.h>
#include <stdio.h>

void func(const char *buff) {
    char *end;
    int si;

    errno = 0;

    const long sl = strtol(buff, &end, 10);

    if (end == buff) {
        fprintf(stderr, "%s: not a decimal number\n", buff);
    } else if ('\0' != *end) {
        fprintf(stderr, "%s: extra characters at end of input: %s\n", buff, end);
    } else if ((LONG_MIN == sl || LONG_MAX == sl) && ERANGE == errno) {
        fprintf(stderr, "%s out of range of type long\n", buff);
    } else if (sl > INT_MAX) {
        fprintf(stderr, "%ld greater than INT_MAX\n", sl);
    } else if (sl < INT_MIN) {
        fprintf(stderr, "%ld less than INT_MIN\n", sl);
    } else {
        si = (int)sl;

        /* Process si */
    }
}
```

Risk Assessment

It is rare for a violation of this rule to result in a security [vulnerability](#) unless it occurs in security-sensitive code. However, violations of this rule can easily result in lost or misinterpreted data.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
ERR34-C	Medium	Unlikely	Medium	P4	L3

Related Guidelines

SEI CERT C++ Coding Standard	INT06-CPP. Use strtol() or a related function to convert a string token to an integer
MITRE CWE	CWE-676 , Use of potentially dangerous function CWE-20 , Insufficient input validation

Bibliography

[ISO/IEC 9899:2011]	Subclause 7.22.1, "Numeric conversion functions" Subclause 7.21.6, "Formatted input/output functions"
[Klein 2002]	

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/6AQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
scanf_functions		dict(...)
symbol_header	Name of the header file of which the symbols should not be used.	['stdlib', 'stdio']
symbols	Names of symbols which are forbidden.	['atof', 'atoi', 'atol', 'atoll']
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.
scanf_conversion_to_number	Potential numeric overflow: do not use functions of scanf() family to convert a string to number.

CertC-CON32

Prevent data races when accessing bit-fields from multiple threads.

Input: IR

Source languages: C, C++

Details

When accessing a bit-field, a thread may inadvertently access a separate bit-field in adjacent memory. This is because compilers are required to store multiple adjacent bit-fields in one storage unit whenever they fit. Consequently, data races may exist not just on a bit-field accessed by multiple threads but also on other bit-fields sharing the same byte or word. A similar problem is discussed in [CON43-C. Do not allow data races in multithreaded code](#), but the issue described by this rule can be harder to diagnose because it may not be obvious that the same memory location is being modified by multiple threads.

One approach for preventing data races in concurrent programming is to use a mutex. When properly observed by all threads, a mutex can provide safe and secure access to a shared object. However, mutexes provide no guarantees with regard to other objects that might be accessed when the mutex is not controlled by the accessing thread. Unfortunately, there is no portable way to determine which adjacent bit-fields may be stored along with the desired bit-field.

Another approach is to insert a non-bit-field member between any two bit-fields to ensure that each bit-field is the only one accessed within its storage unit. This technique effectively guarantees that no two bit-fields are accessed simultaneously.

Noncompliant Code Example (Bit-field)

Adjacent bit-fields may be stored in a single memory location. Consequently, modifying adjacent bit-fields in different threads is [undefined behavior](#), as shown in this noncompliant code example:

```
struct multi_threaded_flags {
    unsigned int flag1 : 2;
    unsigned int flag2 : 2;
};

struct multi_threaded_flags flags;

int thread1(void *arg) {
    flags.flag1 = 1;
    return 0;
}

int thread2(void *arg) {
    flags.flag2 = 2;
    return 0;
}
```

The C Standard, 3.14, paragraph 3 [[ISO/IEC 9899:2011](#)], states

NOTE 2 A bit-field and an adjacent non-bit-field member are in separate memory locations. The same applies to two bit-fields, if one is declared inside a nested structure 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 member declaration. It is not safe to concurrently update two non-atomic bit-fields in the same structure if all members declared between them are also (non-zero-length) bit-fields, no matter what the sizes of those intervening bit-fields happen to be.

For example, the following instruction sequence is possible:

```
Thread 1: register 0 = flags
  Thread 1: register 0 &= ~mask(flag1)
  Thread 2: register 0 = flags
  Thread 2: register 0 &= ~mask(flag2)
  Thread 1: register 0 |= 1 << shift(flag1)
  Thread 1: flags = register 0
  Thread 2: register 0 |= 2 << shift(flag2)
  Thread 2: flags = register 0
```

Compliant Solution (Bit-field, C11, Mutex)

This compliant solution protects all accesses of the flags with a mutex, thereby preventing any data races:

```
#include <threads.h>

struct multi_threaded_flags {
    unsigned int flag1 : 2;
    unsigned int flag2 : 2;
};

struct mtf_mutex {
    struct multi_threaded_flags s;
    mtx_t mutex;
};

struct mtf_mutex flags;

int thread1(void *arg) {
    if (thrd_success != mtx_lock(&flags.mutex)) {
        /* Handle error */
    }
    flags.s.flag1 = 1;
    if (thrd_success != mtx_unlock(&flags.mutex)) {
        /* Handle error */
    }
    return 0;
}

int thread2(void *arg) {
    if (thrd_success != mtx_lock(&flags.mutex)) {
        /* Handle error */
    }
    flags.s.flag2 = 2;
    if (thrd_success != mtx_unlock(&flags.mutex)) {
        /* Handle error */
    }
    return 0;
}
```

Compliant Solution (C11)

In this compliant solution, two threads simultaneously modify two distinct non-bit-field members of a structure. Because the members occupy different bytes in memory, no concurrency protection is required.

```
struct multi_threaded_flags {
    unsigned char flag1;
    unsigned char flag2;
};

struct multi_threaded_flags flags;

int thread1(void *arg) {
    flags.flag1 = 1;
    return 0;
}

int thread2(void *arg) {
    flags.flag2 = 2;
    return 0;
}
```

Unlike C99, C11 explicitly defines a memory location and provides the following note in subclause 3.14.2 [ISO/IEC 9899:2011]:

NOTE 1 Two threads of execution can update and access separate memory locations without interfering with each other.

It is almost certain that `flag1` and `flag2` are stored in the same word. Using a compiler that conforms to C99 or earlier, if both assignments occur on a thread-scheduling interleaving that ends with both stores occurring after one another, it is possible that only one of the flags will be set as intended. The other flag will contain its previous value because both members are represented by the same word, which is the smallest unit the processor can work on. Before the changes were made to the C Standard for C11, there were no guarantees that these flags could be modified concurrently.

Risk Assessment

Although the race window is narrow, an assignment or an expression can evaluate improperly because of misinterpreted data resulting in a corrupted running state or unintended information disclosure.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON32-C	Medium	Probable	Medium	P8	L2

Bibliography

[ISO/IEC 9899:2011]	3.14, "Memory Location"
Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [https://www.securecoding.cert.org/confluence/x/LAAV], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.	

Configuration

Name	Explanation	Value
access_kinds	Access kinds (e.g. Reading_Operand_Interface, Writing_Operand_Interface, Address_Operand_Interface).	['Reading_Operand_Interface', 'Writing_Operand_Interface']
allow_c11_atomics	If set, don't report races on C11 atomic variables.	True
allow_volatile_sig_atomic_t	If set, don't report races on variables of type "volatile sig_atomic_t".	False
debug_output	Option to provide diagnostic output.	False
enter_critical_functions	List of function names to enter a critical region.	[]
enter_critical_macros	List of macro names to enter a critical region (macros must expand to asm() statement).	[]
excluded_routines	List of functions that should be excluded from check.	[]
excluded_subgraphs	List of entry functions to subgraphs that should be excluded as subgraph from check.	[]
exit_critical_functions	List of function names to exit a critical region.	[]
exit_critical_macros	List of macro names to exit a critical region (macros must expand to asm() statement).	[]
inspect_pointers	Whether pointer targets should be inspected to detect more global variable uses.	False
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
nested_critical_regions	If set to True, critical regions nest; if set to False, a single exit-critical-region terminates all open critical regions.	True
output_safe_accesses	When enabled, outputs not only unsafe variable accesses, but also the safe ones.	False
partitions	Dict with partition name as key and dict as value which may contain keys 'entries' and/or 'vectors' with lists of entry points or vector table variables respectively. If special partition '*IRQ*' to configure interrupt handlers is missing, all functions not reached by any of the other options are treated as interrupt handlers.	dict(...)
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
report_cfg_based_critical_region_issues	Report unbalanced lock/unlock pairs within a routine. This has the same intention, but is slightly less strict than the purely syntactic check performed by the rule Parallelism-IncorrectCriticalSection.	False
show_identical_access	When enabled, outputs variable accesses of same kind (i.e., R/R and W/W).	True
show_object_number	Option for debugging (shows internal node numbers).	False
treat_types_as_atomic	List of type-patterns. A type-pattern is either a regular expression of a type name, or a triple of {min. alignment, max. size, type name-regex}. Each of the triple's components may be None. None is interpreted as general wildcard.	[]

Possible Messages

Name	Message
data-race-on-bitfields	Prevent data races when accessing bit-fields from multiple threads.
multiple_lock_add	Lock is acquired while it is already locked.
removed_nonexisting_lock	Lock is released, although it is not currently locked.
unbalanced_locks_path	Different control flow paths have different sets of locks.
unbalanced_locks_routine	Routine may return with different lock set than it is entered with {{in_set} vs {out_set}}.

CertC-CON40

Do not refer to an atomic variable twice in an expression.

Input: IR

Source languages: C, C++

Details

A consistent locking policy guarantees that multiple threads cannot simultaneously access or modify shared data. Atomic variables eliminate the need for locks by guaranteeing thread safety when certain operations are performed on them. The thread-safe operations on atomic variables are specified in the C Standard, subclauses 7.17.7 and 7.17.8 [ISO/IEC 9899:2011]. While atomic operations can be combined, combined operations do not provide the thread safety provided by individual atomic operations.

Every time an atomic variable appears on the left side of an assignment operator, including a compound assignment operator such as *=, an atomic write is performed on the variable. The use of the increment (++) or decrement (--) operators on an atomic variable constitutes an atomic read-and-write operation and is consequently thread-safe. Any reference of an atomic variable anywhere else in an expression indicates a distinct atomic read on the variable.

If the same atomic variable appears twice in an expression, then two atomic reads, or an atomic read and an atomic write, are required. Such a pair of atomic operations is not thread-safe, as another thread can modify the atomic variable between the two operations. Consequently, an atomic variable must not be referenced twice in the same expression.

Noncompliant Code Example [atomic_bool]

This noncompliant code example declares a shared `atomic_bool flag` variable and provides a `toggle_flag()` method that negates the current value of `flag`:

```
#include <stdatomic.h>
#include <stdbool.h>

static atomic_bool flag = ATOMIC_VAR_INIT(false);

void init_flag(void) {
    atomic_init(&flag, false);
}

void toggle_flag(void) {
    bool temp_flag = atomic_load(&flag);
    temp_flag = !temp_flag;
    atomic_store(&flag, temp_flag);
}

bool get_flag(void) {
    return atomic_load(&flag);
}
```

Execution of this code may result in unexpected behavior because the value of `flag` is read, negated, and written back. This occurs even though the read and write are both atomic.

Consider, for example, two threads that call `toggle_flag()`. The expected effect of toggling `flag` twice is that it is restored to its original value. However, the scenario in the following table leaves `flag` in the incorrect state.

toggle_flag() without Compare-and-Exchange

Time	flag	Thread	Action
1	true	t_1	Reads the current value of <code>flag</code> , which is <code>true</code> , into a cache
2	true	t_2	Reads the current value of <code>flag</code> , which is still <code>true</code> , into a different cache
3	true	t_1	Toggles the temporary variable in the cache to <code>false</code>
4	true	t_2	Toggles the temporary variable in the different cache to <code>false</code>
5	false	t_1	Writes the cache variable's value to <code>flag</code>
6	false	t_2	Writes the different cache variable's value to <code>flag</code>

As a result, the effect of the call by t_2 is not reflected in `flag`; the program behaves as if `toggle_flag()` was called only once, not twice.

Compliant Solution (`atomic_compare_exchange_weak()`)

This compliant solution uses a compare-and-exchange to guarantee that the correct value is stored in `flag`. All updates are visible to other threads. The call to `atomic_compare_exchange_weak()` is in a loop in conformance with [CON41-C. Wrap functions that can fail spuriously in a loop](#).

```
#include <stdatomic.h>
#include <stdbool.h>

static atomic_bool flag = ATOMIC_VAR_INIT(false);

void init_flag(void) {
    atomic_init(&flag, false);
}

void toggle_flag(void) {
    bool old_flag = atomic_load(&flag);
    bool new_flag;
    do {
        new_flag = !old_flag;
    } while (!atomic_compare_exchange_weak(&flag, &old_flag, new_flag));
}

bool get_flag(void) {
    return atomic_load(&flag);
}
```

An alternative solution is to use the `atomic_flag` data type for managing Boolean values atomically. However, `atomic_flag` does not support a toggle operation.

Compliant Solution (Compound Assignment)

This compliant solution uses the `^=` assignment operation to toggle `flag`. This operation is guaranteed to be atomic, according to the C Standard, 6.5.16.2, paragraph 3. This operation performs a bitwise-exclusive-or between its arguments, but for Boolean arguments, this is equivalent to negation.

```
#include <stdatomic.h>
#include <stdbool.h>

static atomic_bool flag = ATOMIC_VAR_INIT(false);

void toggle_flag(void) {
    flag ^= 1;
}

bool get_flag(void) {
    return flag;
}
```

An alternative solution is to use a mutex to protect the atomic operation, but this solution loses the performance benefits of atomic variables.

Noncompliant Code Example

This noncompliant code example takes an atomic global variable `n` and computes $n + (n - 1) + (n - 2) + \dots + 1$, using the formula $n * (n + 1) / 2$:

```
#include <stdatomic.h>

atomic_int n = ATOMIC_VAR_INIT(0);

int compute_sum(void) {
    return n * (n + 1) / 2;
}
```

The value of `n` may change between the two atomic reads of `n` in the expression, yielding an incorrect result.

Compliant Solution

This compliant solution passes the atomic variable as a function argument, forcing the variable to be copied and guaranteeing a correct result. Note that the function's formal parameter need not be atomic, and the atomic variable can still be passed as an actual argument.

```
#include <stdatomic.h>

int compute_sum(int n) {
    return n * (n + 1) / 2;
}
```

Risk Assessment

When operations on atomic variables are assumed to be atomic, but are not atomic, surprising data races can occur, leading to corrupted data and invalid control flow.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON40-C	Medium	Probable	Medium	P8	L2

Related Guidelines

MITRE CWE	CWE-366, Race Condition within a Thread
	CWE-413, Improper Resource Locking
	CWE-567, Unsynchronized Access to Shared Data in a Multithreaded Context
	CWE-667, Improper Locking

Bibliography

[ISO/IEC 9899:2011]	6.5.16.2, "Compound Assignment"
	7.17, "Atomics"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/fwBnBw>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
statements_to_look_ahead	Specifies after how many statements of an atomic_load an atomic_store may appear. Default is 3.	3

Possible Messages

Name	Message
atomics_used_twice	Do not refer to an atomic variable twice in an expression.
unsafe_atomic_load_store	'atomic_store' after an 'atomic_load' is unsafe in multi-threaded environments.

CertC-MS24

Do not use deprecated or obsolescent functions.

Input: IR

Source languages: C, C++

Details

Do not use deprecated or obsolescent functions when more secure equivalent functions are available. Deprecated functions are defined by the C Standard. Obsolescent functions are defined by this recommendation.

Deprecated Functions

The `gets()` function was deprecated by Technical Corrigendum 3 to C99 and eliminated from C11.

Obsolescent Functions

Functions in the first column of the following table are hereby defined to be *obsolescent functions*. To remediate invocations of obsolescent functions, an application might use inline coding that, in all respects, conforms to this guideline, or an alternative library that, in all respects, conforms to this guideline, or alternative *non-obsolescent functions*.

Obsolescent Function	Recommended Alternative	Rationale
asctime()	asctime_s()	Non-reentrant
atof()	strtod()	No error detection
atoi()	strtol()	No error detection
atol()	strtol()	No error detection
atoll()	strtoll()	No error detection
ctime()	ctime_s()	Non-reentrant
fopen()	fopen_s()	No exclusive access to file
freopen()	freopen_s()	No exclusive access to file
rewind()	fseek()	No error detection
setbuf()	setvbuf()	No error detection

The `atof()`, `atoi()`, `atol()`, and `atoll()` functions are obsolescent because the `strtod()`, `strtof()`, `strtol()`, `strtold()`, `strtoll()`, `strtoul()`, and `strtoull()` functions can emulate their usage and have more robust error handling capabilities. See [INT05-C. Do not use input functions to convert character data if they cannot handle all possible inputs](#).

The `fopen()` and `freopen()` functions are obsolescent because the `fopen_s()` and `freopen_s()` functions can emulate their usage and improve security by protecting the file from unauthorized access by setting its file protection and opening the file with exclusive access [[ISO/IEC WG14 N1173](#)].

The `setbuf()` function is obsolescent because `setbuf()` does not return a value and can be emulated using `setvbuf()`. See [ERR07-C. Prefer functions that support error checking over equivalent functions that don't](#).

The `rewind()` function is obsolescent because `rewind()` does not return a value and can be emulated using `fseek()`. See [ERR07-C. Prefer functions that support error checking over equivalent functions that don't](#).

The `asctime()` and `ctime()` functions are obsolescent because they use non-reentrant static buffers and can be emulated using `asctime_s()` and `ctime_s()`.

Unchecked Obsolescent Functions

The following are hereby defined to be *unchecked obsolescent functions*:

	bsearch		fprintf	fscanf	fwprintf	fwscanf
getenv	gmtime	localtime	mbsrtowcs	mbstowcs	memcpy	memmove
printf	qsort	setbuf	snprintf	sprintf	sscanf	strcat
strcpy	strerror	strncat	strncpy	strtok	swprintf	swscanf
vfprintf	vfscanf	vfwprintf	vfwscanf	vprintf	vscanf	vsnprintf
vsscanf	vsscanf	vswprintf	vswscanf	vwprintf	vwscanf	wcrtomb
wcscat	wcscpy	wcsncat	wcsncpy	wcsrtombs	wcstok	wcstombs
wctomb	wmemcpy	wmemmove	wprintf	wscanf		

To remediate invocations of unchecked obsolescent functions, an application might use inline coding that, in all respects, conforms to this guideline, or an alternative library that, in all respects, conforms to this guideline, or alternative *nonobsolescent functions* from C11, Annex K:

abort_handler_s		bsearch_s		fprintf_s	freopen_s	fscanf_f_s
fwprintf_s	fwscanf_s	getenv_s	gets_s	gmtime_s	ignore_handler_s	localtime_s
mbsrtowcs_s	mbstowcs_s	memcpy_s	memmove_s	printf_s	qsort_s	scanf_s
set_constraint_handler_s	snprintf_s	snwprintf_s	sprintf_s	sscanf_s	strcat_s	strcpy_s
strerror_s	strerrorlen_s	strncat_s	strncpy_s	strnlen_s	strtok_s	swprintf_s
swscanf_s	vfprintf_s	vfscanf_s	vfwprintf_s	vfwscanf_s	vprintf_s	vscanf_s
vsnprintf_s	vsnwprintf_s	vsprintf_s	vsscanf_s	vswprintf_s	vswscanf_s	vwprintf_s
vwscanf_s	wcrtomb_s	wcrtoms_s	wcscat_s	wcscopy_s	wcsncat_s	wcsncpy_s
wcsnlen_s	wcsrtombs_s	wcstok_s	wcstombs_s	wctomb_s	wmemcpy_s	wmemmove_s
wprintf_s	wscanf_s					

or alternative *nonobsolete functions* from ISO/IEC TR 24731-2, *Extensions to the C Library-Part II: Dynamic Allocation Functions* [[ISO/IEC TR 24731-2](#)]:

asprintf	aswprintf	fmemopen	fscanf	fwscanf	getdelim	getline
getwdelim	getwline	open_memstream	open_wmemstream	strdup	strndup	

Noncompliant Code Example

In this noncompliant code example, the obsolescent functions `strcat()` and `strcpy()` are used:

```
#include <string.h>
#include <stdio.h>

enum { BUFSIZE = 32 };
void complain(const char *msg) {

    static const char prefix[] = "Error: ";
    static const char suffix[] = "\n";
    char buf[BUFSIZE];

    strcpy(buf, prefix);
    strcat(buf, msg);
    strcat(buf, suffix);
    fputs(buf, stderr);
}
```

Compliant Solution

In this compliant solution, `strcat()` and `strcpy()` are replaced by `strcat_s()` and `strcpy_s()`:

```
#define __STDC_WANT_LIB_EXT1__
#include <string.h>
#include <stdio.h>

enum { BUFFERSIZE = 256 };

void complain(const char *msg) {
    static const char prefix[] = "Error: ";
    static const char suffix[] = "\n";
    char buf[BUFFERSIZE];

    strcpy_s(buf, BUFFERSIZE, prefix);
    strcat_s(buf, BUFFERSIZE, msg);
    strcat_s(buf, BUFFERSIZE, suffix);
    fputs(buf, stderr);
}
```

Risk Assessment

The deprecated and obsolescent functions enumerated in this guideline are commonly associated with software [vulnerabilities](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC24-C	High	Probable	Medium	P12	L1

Related Guidelines

CERT C Secure Coding Standard	ERR07-C. Prefer functions that support error checking over equivalent functions that don't INT05-C. Do not use input functions to convert character data if they cannot handle all possible inputs ERR34-C. Detect errors when converting a string to a number STR06-C. Do not assume that strtok() leaves the parse string unchanged STR07-C. Use the bounds-checking interfaces for string manipulation
ISO/IEC TR 24772	Use of Libraries [TRJ]
MISRA C:2012	Rule 21.3 (required)
MITRE CWE	CWE-20 , Insufficient input validation CWE-73 , External control of file name or path CWE-79 , Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') CWE-89 , Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') CWE-91 , XML Injection (aka Blind XPath Injection) CWE-94 , Improper Control of Generation of Code ('Code Injection') CWE-114 , Process Control CWE-120 , Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') CWE-192 , Integer coercion error CWE-197 , Numeric truncation error CWE-367 , Time-of-check, time-of-use race condition CWE-464 , Addition of data structure sentinel CWE-601 , URL Redirection to Untrusted Site ('Open Redirect') CWE-676 , Use of potentially dangerous function

Bibliography

[Apple 2006]	Apple Secure Coding Guide, "Avoiding Race Conditions and Insecure File Operations"
[Burch 2006]	Specifications for Managed Strings, Second Edition
[Drepper 2006]	Section 2.2.1 "Identification When Opening"
[IEEE Std 1003.1:2013]	XSH, System Interfaces, <code>open</code>
[ISO/IEC 23360-1:2006]	
[ISO/IEC WG14 N1173]	Rationale for TR 24731 Extensions to the C Library Part I: Bounds-checking interfaces
[Klein 2002]	"Bullet Proof Integer Input Using <code>strtol()</code> "
[Linux 2008]	<code>strtok(3)</code>
[Seacord 2013]	Chapter 2, "Strings" Chapter 8, "File I/O"
[Seacord 2005b]	"Managed String Library for C, C/C++"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/LwDpAQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
blacklist	Dictionary of header globbing to (list of) function name globbing(s) of forbidden functions.	<code>dict{...}</code>
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	True
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
<code>forbidden_libfunc_call</code>	Do not use deprecated or obsolescent functions.

CertC-MS30

Do not use the `rand()` function for generating pseudorandom numbers.

Input: IR

Source languages: C, C++

Details

Pseudorandom number generators use mathematical algorithms to produce a sequence of numbers with good statistical properties, but the numbers produced are not genuinely random.

The C Standard `rand()` function makes no guarantees as to the quality of the random sequence produced. The numbers generated by some implementations of `rand()` have a comparatively short cycle and the numbers can be predictable. Applications that have strong pseudorandom number requirements must use a generator that is known to be sufficient for their needs.

Noncompliant Code Example

The following noncompliant code generates an ID with a numeric part produced by calling the `rand()` function. The IDs produced are predictable and have limited randomness.

```
#include <stdio.h>
#include <stdlib.h>

enum { len = 12 };

void func(void) {
    /*
     * id will hold the ID, starting with the characters
     * "ID" followed by a random integer.
     */
    char id[len];
    int r;
    int num;
    /* ... */
    r = rand(); /* Generate a random integer */
    num = sprintf(id, len, "ID%-d", r); /* Generate the ID */
    /* ... */
}
```

Compliant Solution (POSIX)

This compliant solution replaces the `rand()` function with the POSIX `random()` function:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

enum { len = 12 };

void func(void) {
    /*
     * id will hold the ID, starting with the characters
     * "ID" followed by a random integer.
     */
    char id[len];
    int r;
    int num;
    /* ... */
    struct timespec ts;
    if (timespec_get(&ts, TIME_UTC) == 0) {
        /* Handle error */
    }
    srand(ts.tv_nsec ^ ts.tv_sec); /* Seed the PRNG */
    /* ... */
    r = random(); /* Generate a random integer */
    num = sprintf(id, len, "ID%-d", r); /* Generate the ID */
    /* ... */
}
```

The POSIX `random()` function is a better pseudorandom number generator. Although on some platforms the low dozen bits generated by `rand()` go through a cyclic pattern, all the bits generated by `random()` are usable. The `rand48` family of functions provides another alternative for pseudorandom numbers.

Although not specified by POSIX, [arc4random\(\)](#) is another possibility for systems that support it. The `arc4random(3)` manual page [OpenBSD] states

... provides higher quality of data than those described in `rand(3)`, `random(3)`, and `drand48(3)`.

To achieve the best random numbers possible, an [implementation](#)-specific function must be used. When unpredictability is crucial and speed is not an issue, as in the creation of strong cryptographic keys, use a true entropy source, such as `/dev/random`, or a hardware device capable of generating random numbers. The `/dev/random` device can block for a long time if there are not enough events going on to generate sufficient entropy.

Compliant Solution (Windows)

On Windows platforms, the [CryptGenRandom\(\)](#) function can be used to generate cryptographically strong random numbers. The exact details of the implementation are unknown, including, for example, what source of entropy `CryptGenRandom()` uses. The Microsoft Developer Network `CryptGenRandom()` reference [MSDN] states

If an application has access to a good random source, it can fill the `pbBuffer` buffer with some random data before calling `CryptGenRandom()`. The

CSP [cryptographic service provider] then uses this data to further randomize its internal seed. It is acceptable to omit the step of initializing the pbBuffer buffer before calling `CryptGenRandom()`.

```
#include <Windows.h>
#include <wincrypt.h>
#include <stdio.h>

void func(void) {
    HCRYPTPROV prov;
    if (CryptAcquireContext(&prov, NULL, NULL,
                           PROV_RSA_FULL, 0)) {
        long int li = 0;
        if (CryptGenRandom(prov, sizeof(li), (BYTE *)&li)) {
            printf("Random number: %ld\n", li);
        } else {
            /* Handle error */
        }
        if (!CryptReleaseContext(prov, 0)) {
            /* Handle error */
        }
    } else {
        /* Handle error */
    }
}
```

Risk Assessment

The use of the `rand()` function can result in predictable random numbers.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC30-C	Medium	Unlikely	Low	P6	L2

Related Guidelines

SEI CERT C++ Coding Standard	MSC50-CPP. Do not use std::rand() for generating pseudorandom numbers
CERT Oracle Secure Coding Standard for Java	MSC02-J. Generate strong random numbers
MITRE CWE	CWE-327 , Use of a Broken or Risky Cryptographic Algorithm CWE-330 , Use of Insufficiently Random Values CWE-331 , Insufficient Entropy CWE-338 , Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)

Bibliography

[MSDN]	"CryptGenRandom Function"
[OpenBSD]	arc4random()

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki <https://www.securecoding.cert.org/confluence/x/qw4>, Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
blacklist	Dictionary of header globbing to [list of] function name globbing(s) of forbidden functions.	dict(...)
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
forbidden_libfunc_call	Do not use the <code>rand()</code> function for generating pseudorandom numbers.

CertC-MSC32

Properly seed pseudorandom number generators.

Input: IR

Source languages: C, C++

Details

A pseudorandom number generator (PRNG) is a deterministic algorithm capable of generating sequences of numbers that approximate the properties of random numbers. Each sequence is completely determined by the initial state of the PRNG and the algorithm for changing the state. Most PRNGs make it possible to set the initial state, also called the *seed state*. Setting the initial state is called *seeding* the PRNG.

Calling a PRNG in the same initial state, either without seeding it explicitly or by seeding it with the same value, results in generating the same sequence of random numbers in different runs of the program. Consider a PRNG function that is seeded with some initial seed value and is consecutively called to produce a sequence of random numbers, *s*. If the PRNG is subsequently seeded with the same initial seed value, then it will generate the same sequence *s*.

As a result, after the first run of an improperly seeded PRNG, an attacker can predict the sequence of random numbers that will be generated in the future runs. Improperly seeding or failing to seed the PRNG can lead to [vulnerabilities](#), especially in security protocols.

The solution is to ensure that the PRNG is always properly seeded. A properly seeded PRNG will generate a different sequence of random numbers each time it is run.

Not all random number generators can be seeded. True random number generators that rely on hardware to produce completely unpredictable results do not need to be and cannot be seeded. Some high-quality PRNGs, such as the `/dev/random` device on some UNIX systems, also cannot be seeded. This rule applies only to algorithmic pseudorandom number generators that can be seeded.

Noncompliant Code Example (POSIX)

This noncompliant code example generates a sequence of 10 pseudorandom numbers using the `random()` function. When `random()` is not seeded, it behaves like `rand()`, producing the same sequence of random numbers each time any program that uses it is run.

```
#include <stdio.h>
#include <stdlib.h>

void func(void) {
    for (unsigned int i = 0; i < 10; ++i) {
        /* Always generates the same sequence */
        printf("%ld, ", random());
    }
}
```

The output is as follows:

```
1st run: 1804289383, 846930886, 1681692777, 1714636915, 1957747793, 424238335, 719885386, 1649760492, 596516649,
         1189641421,
2nd run: 1804289383, 846930886, 1681692777, 1714636915, 1957747793, 424238335, 719885386, 1649760492, 596516649,
         1189641421,
...
nth run: 1804289383, 846930886, 1681692777, 1714636915, 1957747793, 424238335, 719885386, 1649760492, 596516649,
         1189641421,
```

Compliant Solution (POSIX)

Call `random()` before invoking `random()` to seed the random sequence generated by `random()`. This compliant solution produces different random number sequences each time the function is called, depending on the resolution of the system clock:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void func(void) {
    struct timespec ts;
    if (timespec_get(&ts, TIME_UTC) == 0) {
        /* Handle error */
    } else {
        random(ts.tv_nsec ^ ts.tv_sec);
        for (unsigned int i = 0; i < 10; ++i) {
            /* Generates different sequences at different runs */
            printf("%ld, ", random());
        }
    }
}
```

The output is as follows:

```
1st run: 198682410, 2076262355, 910374899, 428635843, 2084827500, 1558698420, 4459146, 733695321, 2044378618, 1649046624,
2nd run: 1127071427, 252907983, 1358798372, 2101446505, 1514711759, 229790273, 954268511, 1116446419, 368192457,
         1297948050,
3rd run: 2052868434, 1645663878, 731874735, 1624006793, 938447420, 1046134947, 1901136083, 418123888, 836428296,
         2017467418,
```

This may not be sufficiently random for concurrent execution, which may lead to correlated generated series in different threads. Depending on the application and the desired level of security, a programmer may choose alternative ways to seed PRNGs. In general, hardware is more capable than software of generating real random numbers (for example, by sampling the thermal noise of a diode).

Compliant Solution (Windows)

The [`CryptGenRandom\(\)`](#) function does not run the risk of not being properly seeded because its arguments serve as seeders:

```

#include <Windows.h>
#include <wincrypt.h>
#include <stdio.h>

void func(void) {
    HCRYPTPROV hCryptProv;
    long rand_buf;
    /* Example of instantiating the CSP */
    if (CryptAcquireContext(&hCryptProv, NULL, NULL,
                           PROV_RSA_FULL, 0)) {
        printf("CryptAcquireContext succeeded.\n");
    } else {
        printf("Error during CryptAcquireContext!\n");
    }

    for (unsigned int i = 0; i < 10; ++i) {
        if (!CryptGenRandom(hCryptProv, sizeof(rand_buf),
                            (BYTE *)&rand_buf)) {
            printf("Error\n");
        } else {
            printf("%ld, ", rand_buf);
        }
    }
}

```

The output is as follows:

```

1st run: -1597837311, 906130682, -1308031886, 1048837407, -931041900, -658114613, -1709220953, -1019697289, 1802206541,
406505841,
2nd run: 885904119, -687379556, -1782296854, 1443701916, -624291047, 2049692692, -990451563, -142307804, 1257079211,
897185104,
3rd run: 190598304, -1537409464, 1594174739, -424401916, -1975153474, 826912927, 1705549595, -1515331215, 474951399,
1982500583,

```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC32-C	Medium	Likely	Low	P18	L1

Related Guidelines

CERT C Secure Coding Standard	MSC30-C. Do not use the rand() function for generating pseudorandom numbers
SEI CERT C++ Coding Standard	MSC51-CPP. Ensure your random number generator is properly seeded
MITRE CWE	CWE-327 , Use of a Broken or Risky Cryptographic Algorithm CWE-330 , Use of Insufficiently Random Values CWE-331 , Insufficient Entropy CWE-338 , Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)

Bibliography

[MSDN]	"CryptGenRandom Function"
------------------------	---

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/hABhAQ>], Copyright [C] 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
seed_properly	Properly seed pseudorandom number generators.

CertC-MS33

Do not pass invalid data to the `asctime()` function.

Input: IR

Source languages: C, C++

Details

The C Standard, 7.27.3.1 [[ISO/IEC 9899:2011](#)], provides the following sample implementation of the `asctime()` function:

```
char *asctime(const struct tm *timeptr) {
    static const char wday_name[7][3] = {
        "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
    };
    static const char mon_name[12][3] = {
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
    };
    static char result[26];
    sprintf(
        result,
        "%3s %3s%3d %2d:%2d:%2d %d\n",
        wday_name[timeptr->tm_wday],
        mon_name[timeptr->tm_mon],
        timeptr->tm_mday, timeptr->tm_hour,
        timeptr->tm_min, timeptr->tm_sec,
        1900 + timeptr->tm_year
    );
    return result;
}
```

This function is supposed to output a character string of 26 characters at most, including the terminating null character. If we count the length indicated by the format directives, we arrive at 25. Taking into account the terminating null character, the array size of the string appears sufficient.

However, this [implementation](#) assumes that the values of the `struct tm` data are within normal ranges and does nothing to enforce the range limit. If any of the values print more characters than expected, the `sprintf()` function may overflow the `result` array. For example, if `tm_year` has the value 12345, then 27 characters (including the terminating null character) are printed, resulting in a buffer overflow.

The *POSIX® Base Specifications* [[IEEE Std 1003.1:2013](#)] says the following about the `asctime()` and `asctime_r()` functions:

These functions are included only for compatibility with older implementations. They have undefined behavior if the resulting string would be too long, so the use of these functions should be discouraged. On implementations that do not detect output string length overflow, it is possible to overflow the output buffers in such a way as to cause applications to fail, or possible system security violations. Also, these functions do not support localized date and time formats. To avoid these problems, applications should use `strftime()` to generate strings from broken-down times.

The C Standard, Annex K, also defines `asctime_s()`, which can be used as a secure substitute for `asctime()`.

The `asctime()` function appears in the list of obsolescent functions in [MSC24-C. Do not use deprecated or obsolescent functions](#).

Noncompliant Code Example

This noncompliant code example invokes the `asctime()` function with potentially unsanitized data:

```
#include <time.h>

void func(struct tm *time_tm) {
    char *time = asctime(time_tm);
    /* ... */
}
```

Compliant Solution (`strftime()`)

The `strftime()` function allows the programmer to specify a more rigorous format and also to specify the maximum size of the resulting time string:

```
#include <time.h>

enum { maxsize = 26 };

void func(struct tm *time) {
    char s[maxsize];
    /* Current time representation for locale */
    const char *format = "%c";

    size_t size = strftime(s, maxsize, format, time);
}
```

This call has the same effects as `asctime()` but also ensures that no more than `maxsize` characters are printed, preventing buffer overflow.

Compliant Solution (`asctime_s()`)

The C Standard, Annex K, defines the `asctime_s()` function, which serves as a close replacement for the `asctime()` function but requires an additional argument that specifies the maximum size of the resulting time string:

```
#define __STDC_WANT_LIB_EXT1__ 1
```

```

#include <time.h>
enum { maxsize = 26 };
void func(struct tm *time_tm) {
    char buffer[maxsize];
    if (asctime_s(buffer, maxsize, &time_tm)) {
        /* Handle error */
    }
}

```

Risk Assessment

On [implementations](#) that do not detect output-string-length overflow, it is possible to overflow the output buffers.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC33-C	High	Likely	Low	P27	L1

Related Guidelines

CERT C Secure Coding Standard	MSC24-C. Do not use deprecated or obsolescent functions
---	---

Bibliography

[IEEE Std 1003.1:2013]	XSH, System Interfaces, <code>asctime</code>
[ISO/IEC 9899:2011]	7.27.3.1, "The <code>asctime</code> Function"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/CgCuAQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
blacklist	Dictionary of header globbing to (list of) function name globbing(s) of forbidden functions.	dict{...}
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	27
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
forbidden_libfunc_call	Potentially unsanitized data passed to <code>asctime</code> . Use <code>asctime_s</code> or <code>strftime</code> instead.

CertC-MSC37

Ensure that control never reaches the end of a non-void function.

Input: IR

Source languages: C, C++

Details

If control reaches the closing curly brace {} of a non-void function without evaluating a `return` statement, using the return value of the function call is [undefined behavior](#). (See [undefined behavior 88](#).)

Noncompliant Code Example

In this noncompliant code example, control reaches the end of the `checkpass()` function when the two strings passed to `strcmp()` are not equal, resulting in undefined behavior. Many compilers will generate code for the `checkpass()` function, returning various values along the execution path where no `return` statement is defined.

```
#include <string.h>
#include <stdio.h>
```

```

int checkpass(const char *password) {
    if (strcmp(password, "pass") == 0) {
        return 1;
    }
}

void func(const char *userinput) {
    if (checkpass(userinput)) {
        printf("Success\n");
    }
}

```

This error is frequently diagnosed by compilers. (See [MSC00-C. Compile cleanly at high warning levels.](#))

Compliant Solution

This compliant solution ensures that the `checkpass()` function always returns a value:

```

#include <string.h>
#include <stdio.h>

int checkpass(const char *password) {
    if (strcmp(password, "pass") == 0) {
        return 1;
    }
    return 0;
}

void func(const char *userinput) {
    if (checkpass(userinput)) {
        printf("Success!\n");
    }
}

```

Noncompliant Code Example

In this noncompliant code example, control reaches the end of the `getlen()` function when `input` does not contain the integer `delim`. Because the potentially undefined return value of `getlen()` is later used as an index into an array, a buffer overflow may occur.

```

#include <stddef.h>

size_t getlen(const int *input, size_t maxlen, int delim) {
    for (size_t i = 0; i < maxlen; ++i) {
        if (input[i] == delim) {
            return i;
        }
    }
}

void func(int userdata) {
    size_t i;
    int data[] = { 1, 1, 1 };
    i = getlen(data, sizeof(data), 0);
    data[i] = userdata;
}

```

Implementation Details (GCC)

Violating this rule can have unexpected consequences, as in the following example:

```

#include <stdio.h>

size_t getlen(const int *input, size_t maxlen, int delim) {
    for (size_t i = 0; i < maxlen; ++i) {
        if (input[i] == delim) {
            return i;
        }
    }
}

int main(int argc, char **argv) {
    size_t i;
    int data[] = { 1, 1, 1 };

    i = getlen(data, sizeof(data), 0);
    printf("Returned: %zu\n", i);
    data[i] = 0;

    return 0;
}

```

When this program is compiled with `-Wall` on most versions of the GCC compiler, the following warning is generated:

```

example.c: In function 'getlen':
example.c:12: warning: control reaches end of non-void function

```

None of the inputs to the function equal the delimiter, so when run with GCC 5.3 on Linux, control reaches the end of the `getlen()` function, which is undefined behavior and in this test returns 3, causing an out-of-bounds write to the `data` array.

Compliant Solution

This compliant solution changes the interface of `getlen()` to store the result in a user-provided pointer and returns a status indicator to report success or failure. The best method for handling this type of error is specific to the application and the type of error. (See [ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy](#) for more on error handling.)

```
#include <stddef.h>

int getlen(const int *input, size_t maxlen, int delim,
           size_t *result) {
    if (result == NULL) {
        return -1;
    }
    for (size_t i = 0; i < maxlen; ++i) {
        if (input[i] == delim) {
            *result = i;
            return 0;
        }
    }
    return -1;
}

void func(int userdata) {
    size_t i;
    int data[] = {1, 1, 1};
    if (getlen(data, sizeof(data), 0, &i) != 0) {
        /* Handle error */
    } else {
        data[i] = userdata;
    }
}
```

Exceptions

MSC37-C-EX1: According to the C Standard, 5.1.2.2.3, paragraph 1 [[ISO/IEC 9899:2011](#)], "Reaching the `}` that terminates the main function returns a value of 0." As a result, it is permissible for control to reach the end of the `main()` function without executing a `return` statement.

MSC37-C-EX2: It is permissible for a control path to not return a value if that code path is never taken and a function marked `_Noreturn` is called as part of that code path. For example:

```
#include <stdio.h>
#include <stdlib.h>

_Noreturn void unreachable(const char *msg) {
    printf("Unreachable code reached: %s\n", msg);
    exit(1);
}

enum E {
    One,
    Two,
    Three
};

int f(enum E e) {
    switch (e) {
        case One: return 1;
        case Two: return 2;
        case Three: return 3;
    }
    unreachable("Can never get here");
}
```

Risk Assessment

Using the return value from a non-`void` function where control reaches the end of the function without evaluating a `return` statement can lead to buffer overflow [vulnerabilities](#) as well as other [unexpected program behaviors](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC37-C	High	Unlikely	Low	P9	L2

Related Guidelines

CERT C Secure Coding Standard	MSC01-C. Strive for logical completeness
---	--

Bibliography

[ISO/IEC 9899:2011]	5.1.2.2.3, "Program Termination"
Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [https://www.securecoding.cert.org/confluence/x/goCGAg], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.	

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	9
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
lambda_return_missing_value	Return without value in non-void lambda expression
missing_return	Ensure that control never reaches the end of a non-void function.
missing_return_in_lambda	Non-void lambda expression needs return with value at end.
return_missing_value	Return without value in non-void function

CertC-POS30

Use the `readlink()` function properly.

Input: IR

Source languages: C, C++

Details

The `readlink()` function reads where a link points to. It makes no effort to null-terminate its second argument, `buffer`. Instead, it just returns the number of characters it has written.

Noncompliant Code Example

If `len` is equal to `sizeof(buf)`, the null terminator is written 1 byte past the end of `buf`:

```
char buf[1024];
ssize_t len = readlink("/usr/bin/perl", buf, sizeof(buf));
buf[len] = '\0';
```

An incorrect solution to this problem is to try to make `buf` large enough that it can always hold the result:

```
long symlink_max;
size_t bufsize;
char *buf;
ssize_t len;

errno = 0;
symlink_max = pathconf("/usr/bin/", _PC_SYMLINK_MAX);
if (symlink_max == -1) {
    if (errno != 0) {
        /* handle error condition */
    }
    bufsize = 10000;
}
else {
    bufsize = symlink_max+1;
}

buf = (char *)malloc(bufsize);
if (buf == NULL) {
    /* handle error condition */
}

len = readlink("/usr/bin/perl", buf, bufsize);
buf[len] = '\0';
```

This modification incorrectly assumes that the symbolic link cannot be longer than the value of `SYMLINK_MAX` returned by `pathconf()`. However, the value returned by `pathconf()` is out of date by the time `readlink()` is called, so the off-by-one buffer-overflow risk is still present because, between the two calls, the location of `/usr/bin/perl` can change to a file system with a larger `SYMLINK_MAX` value. Also, if `SYMLINK_MAX` is indeterminate (that is, if `pathconf()` returned `-1` without setting `errno`), the code uses an arbitrary large buffer size (10,000) that it hopes will be sufficient, but there is a small chance that `readlink()` can return exactly this size.

An additional issue is that `readlink()` can return `-1` if it fails, causing an off-by-one underflow.

Compliant Solution

This compliant solution ensures there is no overflow by reading in only `sizeof(buf) - 1` characters. It also properly checks to see if an error has occurred:

```
enum { BUFSIZE = 1024 };
char buf[BUFSIZE];
ssize_t len = readlink("/usr/bin/perl", buf, sizeof(buf)-1);

if (len != -1) {
    buf[len] = '\0';
}
else {
    /* handle error condition */
}
```

Risk Assessment

Failing to properly null-terminate the result of `readlink()` can result in abnormal program termination and buffer-overflow vulnerabilities.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
POS30-C	high	probable	medium	P12	L1

Related Guidelines

MITRE CWE	CWE-170 , Improper null termination
---------------------------	---

Bibliography

[Ilja 2006]
[Open Group 1997a]
[Open Group 2004]

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki <https://www.securecoding.cert.org/confluence/x/OQU>, Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
errno_headers	Header names were 'errno', 'EILSEQ' and 'ERANGE' might be located.	['errno.h', 'errno-base.h']
functions	Allows to declare function names for which a check must exist. The check is expressed as an IR pattern.	dict{...}
functions_under_test	Readlink functions which should be used properly.	['readlink']
known_check_functions	Collection of functions which are known to test return values of functions under test.	[]
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
stdio_headers	Header names were 'EOF' might be located.	['stdio.h', 'libio.h']
threads_headers	Header names were thread related macros and functions might be located.	['threads.h']
wchar_headers	Header names were 'WEOF' might be located.	['wchar.h', 'corecrt_wctype.h', '_wctype.h', '_types.h']

Possible Messages

Name	Message
unhandled_return_value	Use the <code>readlink()</code> function properly.
wrong_usage	Use the <code>readlink()</code> function properly.

CertC-POS33

Do not use `vfork()`.

Input: IR

Source languages: C, C++

Details

Using the `vfork` function introduces many portability and security issues. There are many cases in which [undefined](#) and implementation-specific behavior can occur, leading to a denial-of-service [vulnerability](#).

According to the `vfork` man page,

The `vfork()` function has the same effect as `fork()`, except that the behavior is undefined if the process created by `vfork()` either modifies any data other than a variable of type `pid_t` used to store the return value from `vfork()`, or returns from the function in which `vfork()` was called, or calls any other function before successfully calling `_exit()` or one of the `exec` family of functions.

Furthermore, older versions of Linux are vulnerable to a race condition, occurring when a privileged process calls `vfork()`, and then the child process lowers its privileges and calls `execve()`. The child process is executed with the unprivileged user's UID before it calls `execve()`.

Because of the implementation of the `vfork()` function, the parent process is suspended while the child process executes. If a user sends a signal to the child process, delaying its execution, the parent process (which is privileged) is also blocked. This means that an unprivileged process can cause a privileged process to halt, which is a privilege inversion resulting in a denial of service.

This code example shows how difficult it is to use `vfork()` without triggering undefined behavior. The lowering of privileges in this case requires a call to `setuid()`, the behavior of which is undefined because it occurs between the `vfork()` and the `execve()`.

```
pid_t pid = vfork();
if (pid == 0) /* child */ {
    setuid(unprivileged_user); /* undefined behavior */
    /*
     * Window of vulnerability to privilege inversion on
     * older versions of Linux
     */
    if (execve(filename, NULL, NULL) == -1) {
        /* Handle error */
    }
    /*
     * In normal operations, execve() might fail; if it does,
     * vfork() behavior is undefined.
     */
    _exit(1); /* in case execve() fails */
}
```

Use `fork()` instead of `vfork()` in all circumstances.

Noncompliant Code Example

This noncompliant code example calls `vfork()` and then `execve()`. As previously discussed, a `vfork()`/`execve()` pair contains an inherent race window on some [implementations](#).

```
char *filename = /* something */;

pid_t pid = vfork();
if (pid == 0) /* child */ {
    if (execve(filename, NULL, NULL) == -1) {
        /* Handle error */
    }
    _exit(1); /* in case execve() fails */
}
```

Compliant Solution

This compliant solution replaces the call to `vfork()` with a call to `fork()`, which does not contain a race condition, and eliminates the denial-of-service vulnerability:

```
char *filename = /* something */;

pid_t pid = fork();
if (pid == 0) /* child */ {
    if (execve(filename, NULL, NULL) == -1) {
        /* Handle error */
    }
    _exit(1); /* in case execve() fails */
}
```

Risk Assessment

Using the `vfork` function can result in a denial-of-service vulnerability.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
POS33-C	low	probable	low	P6	L2

Related Guidelines

MITRE CWE	CWE-242 , Use of inherently dangerous function
---------------------------	--

Bibliography

[Wheeler 2003]	Section 8.6
--------------------------------	-----------------------------

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/EgAa>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
blacklist	Dictionary of header globbing to (list of) function name globbing(s) of forbidden functions.	dict(...)
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
forbidden_libfunc_call	Do not use vfork().

CertC-POS34

Do not call `putenv()` with a pointer to an automatic variable as the argument.

Input: IR

Source languages: C, C++

Details

The POSIX function `putenv()` is used to set environment variable values. The `putenv()` function does not create a copy of the string supplied to it as an argument; rather, it inserts a pointer to the string into the environment array. If a pointer to a buffer of automatic storage duration is supplied as an argument to `putenv()`, the memory allocated for that buffer may be overwritten when the containing function returns and stack memory is recycled. This behavior is noted in the Open Group Base Specifications, Issue 6 [[Open Group 2004](#)]:

A potential error is to call `putenv()` with an automatic variable as the argument, then return from the calling function while `string` is still part of the environment.

The actual problem occurs when passing a *pointer* to an automatic variable to `putenv()`. An automatic pointer to a static buffer would work as intended.

Noncompliant Code Example

In this noncompliant code example, a pointer to a buffer of automatic storage duration is used as an argument to `putenv()` [[Dowd 2006](#)]. The `TEST` environment variable may take on an unintended value if it is accessed after `func()` has returned and the stack frame containing `env` has been recycled.

Note that this example also violates [DCL30-C. Declare objects with appropriate storage durations](#).

```
int func(const char *var) {
    char env[1024];
    int retval = snprintf(env, sizeof(env), "TEST=%s", var);
    if (retval < 0 || (size_t)retval >= sizeof(env)) {
        /* Handle error */
    }
    return putenv(env);
}
```

Compliant Solution (`static`)

This compliant solution uses a static array for the argument to `putenv()`.

```
int func(const char *var) {
    static char env[1024];

    int retval = sprintf(env, sizeof(env), "TEST=%s", var);
    if (retval < 0 || (size_t)retval >= sizeof(env)) {
        /* Handle error */
    }

    return putenv(env);
}
```

Compliant Solution (Heap Memory)

This compliant solution dynamically allocates memory for the argument to `putenv()`:

```
int func(const char *var) {
    static char *oldenv;
    const char *env_format = "TEST=%s";
    const size_t len = strlen(var) + strlen(env_format);
    char *env = (char *) malloc(len);
    if (env == NULL) {
        return -1;
    }
    int retval = sprintf(env, len, env_format, var);
    if (retval < 0 || (size_t)retval >= len) {
        /* Handle error */
    }
    if (putenv(env) != 0) {
        free(env);
        return -1;
    }
    if (oldenv != NULL) {
        free(oldenv); /* avoid memory leak */
    }
    oldenv = env;
    return 0;
}
```

The POSIX `setenv()` function is preferred over this function [[Open Group 2004](#)].

Compliant Solution (`setenv()`)

The `setenv()` function allocates heap memory for environment variables, which eliminates the possibility of accessing volatile stack memory:

```
int func(const char *var) {
    return setenv("TEST", var, 1);
}
```

Using `setenv()` is easier and consequently less error prone than using `putenv()`.

Risk Assessment

Providing a pointer to a buffer of automatic storage duration as an argument to `putenv()` may cause that buffer to take on an unintended value. Depending on how and when the buffer is used, it can cause unexpected program behavior or possibly allow an attacker to run arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
POS34-C	high	unlikely	medium	P6	L2

Related Guidelines

MITRE CWE	CWE-686 , Function call with incorrect argument type CWE-562 , Return of stack variable address
---------------------------	--

Bibliography

[Dowd 2006]	Chapter 10, "UNIX Processes"
[ISO/IEC 9899:2011]	Section 6.2.4, "Storage Durations of Objects" Section 7.22.3, "Memory Management Functions"
[Open Group 2004]	putenv() setenv()

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/HoAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
putenv_arg_pointing_to_local	Do not call putenv() with a pointer to an automatic variable as the argument.

CertC-POS35

Avoid race conditions while checking for the existence of a symbolic link.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Many common operating systems, such as Windows and UNIX, support symbolic (soft) links. Symbolic links can be created in UNIX using the `ln -s` command or in Windows by using directory junctions in NTFS or the Linkd.exe (Win 2K resource kit) or "junction" freeware.

If not properly performed, checking for the existence of symbolic links can lead to race conditions.

Noncompliant Code Example

The POSIX `lstat()` function collects information about a symbolic link rather than its target. This noncompliant code example uses the `lstat()` function to collect information about the file, checks the `st_mode` field to determine if the file is a symbolic link, and then opens the file if it is not a symbolic link:

```
char *filename = /* file name */;
char *userbuf = /* user data */;
unsigned int userlen = /* length of userbuf string */;

struct stat lstat_info;
int fd;
/* ... */
if (lstat(filename, &lstat_info) == -1) {
    /* Handle error */
}

if (!S_ISLNK(lstat_info.st_mode)) {
    fd = open(filename, O_RDWR);
    if (fd == -1) {
        /* Handle error */
    }
}
if (write(fd, userbuf, userlen) < userlen) {
    /* Handle error */
}
```

This code contains a time-of-check, time-of-use (TOCTOU) race condition between the call to `lstat()` and the subsequent call to `open()` because both functions operate on a file name that can be manipulated asynchronously to the execution of the program. (See [F1001-C. Be careful using functions that use file names for identification](#).)

Compliant Solution

This compliant solution eliminates the race condition by

1. Calling `lstat()` on the file name.
2. Calling `open()` to open the file.
3. Calling `fstat()` on the file descriptor returned by `open()`.
4. Comparing the file information returned by the calls to `lstat()` and `fstat()` to ensure that the files are the same.

```
char *filename = /* file name */;
char *userbuf = /* user data */;
unsigned int userlen = /* length of userbuf string */;

struct stat lstat_info;
struct stat fstat_info;
int fd;
/* ... */
if (lstat(filename, &lstat_info) == -1) {
    /* handle error */
}
```

```

}

fd = open(filename, O_RDWR);
if (fd == -1) {
    /* handle error */
}

if (fstat(fd, &fstat_info) == -1) {
    /* handle error */
}

if (lstat_info.st_mode == fstat_info.st_mode &&
    lstat_info.st_ino == fstat_info.st_ino &&
    lstat_info.st_dev == fstat_info.st_dev) {
    if (write(fd, userbuf, userlen) < userlen) {
        /* Handle Error */
    }
}

```

This code eliminates the TOCTOU condition because `fstat()` is applied to file descriptors, not file names, so the file passed to `fstat()` must be identical to the file that was opened. The `lstat()` function does not follow symbolic links, but `open()` does. Comparing modes using the `st_mode` field is sufficient to check for a symbolic link.

Comparing i-nodes, using the `st_ino` fields, and devices, using the `st_dev` fields, ensures that the file passed to `lstat()` is the same as the file passed to `fstat()`. (See [FI005-C. Identify files using multiple file attributes.](#))

Risk Assessment

TOCTOU race condition vulnerabilities can be exploited to gain elevated privileges.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
POS35-C	high	likely	medium	P18	L1

Related Guidelines

MITRE CWE	CWE-363 , Race condition enabling link following CWE-365 , Race condition in switch
---------------------------	--

Bibliography

[Dowd 2006]	Chapter 9, "UNIX 1: Privileges and Files"
[ISO/IEC 9899:2011]	Section 7.21, "Input/output <stdio.h>"
[Open Group 2004]	lstat() fstat() open()
[Seacord 2013]	Chapter 8, "File I/O"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/ZgAI>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
no_fstat_after_lstat_open	A <code>fstat()</code> should follow an <code>open()</code> that is preceded by a <code>lstat()</code> call.

CertC-POS36

Observe correct revocation order while relinquishing privileges.

Input: IR

Source languages: C, C++

Details

In case of set-user-ID and set-group-ID programs, when the effective user ID and group ID are different from those of the real user, it is important to drop not only the user-level privileges but also the group privileges. While doing so, the order of revocation must be correct.

POSIX defines `setgid()` to have the following behavior [[Open Group 2004](#)]:

If the process has appropriate privileges, `setgid()` shall set the real group ID, effective group ID, and the saved set-group-ID of the calling process to `gid`.

If the process does not have appropriate privileges, but `gid` is equal to the real group ID or the saved set-group-ID, `setgid()` shall set the effective group ID to `gid`; the real group ID and saved set-group-ID shall remain unchanged.

Noncompliant Code Example

This noncompliant code example drops privileges to those of the real user and similarly drops the group privileges. However, the order is incorrect because the `setgid()` function must be run with superuser privileges, but the call to `setuid()` leaves the effective user ID as nonzero. As a result, if a vulnerability is discovered in the program that allows for the execution of arbitrary code, an attacker can regain the original group privileges.

```
/* Drop superuser privileges in incorrect order */

if (setuid(getuid()) == -1) {
    /* handle error condition */
}
if (setgid(getgid()) == -1) {
    /* handle error condition */
}

/* It is still possible to regain group privileges due to
 * incorrect relinquishment order */
```

Compliant Solution

This compliant solution relinquishes group privileges before taking away the user-level privileges so that both operations execute as intended.

```
/* Drop superuser privileges in correct order */

if (setgid(getgid()) == -1) {
    /* handle error condition */
}
if (setuid(getuid()) == -1) {
    /* handle error condition */
}

/*
 * Not possible to regain group privileges due to correct relinquishment order
 */
```

Supplementary Group IDs

A process may have a number of supplementary group IDs in addition to its effective group ID, and the supplementary groups can allow privileged access to files. The `getgroups()` function returns an array that contains the supplementary group IDs and may also contain the effective group ID. The `setgroups()` function can set the supplementary group IDs and may also set the effective group ID on some systems. Using `setgroups()` usually requires privileges. Although POSIX defines the `getgroups()` function, it does not define `setgroups()`.

Under normal circumstances, `setuid()` and related calls do not alter the supplementary group IDs. However, a setuid-root program can alter its supplementary group IDs and then relinquish root privileges, in which case it maintains the supplementary group IDs but lacks the privilege necessary to relinquish them. Consequently, it is recommended that a program relinquish supplementary group IDs immediately before relinquishing root privileges. The following code defines a `set_sups()` function that will set the supplementary group IDs to a specific array on systems that support the `setgroups()` function.

```
/* Returns nonzero if the two group lists are equivalent (taking into
   account that the lists may differ wrt the egid *)
int eql_sups(const int cursups_size, const gid_t* const cursups_list,
             const int targetsups_size, const gid_t* const targetsups_list) {
    int i;
    int j;
    const int n = targetsups_size;
    const int diff = cursups_size - targetsups_size;
    const gid_t egid = getegid();
    if (diff > 1 || diff < 0 ) {
        return 0;
    }
    for (i=0, j=0; i < n; i++, j++) {
        if (cursups_list[j] != targetsups_list[i]) {
            if (cursups_list[j] == egid) {
                i--; /* skipping j */
            } else {
                return 0;
            }
        }
    }
    /* If reached here, we're sure i==targetsups_size. Now, either
       j==cursups_size (skipped the egid or it wasn't there), or we didn't
       get to the egid yet because it's the last entry in cursups */
    return j == cursups_size ||
           (j+1 == cursups_size && cursups_list[j] == egid);
```

```

}

/* Sets the supplementary group list, returns 0 if successful */
int set_sups(const int target_sups_size,const gid_t* const target_sups_list) {
#ifdef __FreeBSD__
    const int targetsups_size = target_sups_size + 1;
    gid_t* const targetsups_list = (gid_t*) malloc(sizeof(gid_t) * targetsups_size);
    if (targetsups_list == NULL) {
        /* handle error */
    }
    memcpy(targetsups_list+1, target_sups_list, target_sups_size * sizeof(gid_t));
    targetsups_list[0] = getegid();
#else
    const int targetsups_size = target_sups_size;
    const gid_t* const targetsups_list = target_sups_list;
#endif
    if (geteuid() == 0) { /* allowed to setgroups, let's not take any chances */
        if (-1 == setgroups(targetsups_size, targetsups_list)) {
            /* handle error */
        }
    } else {
        int cursups_size = getgroups(0, NULL);
        gid_t* cursups_list = (gid_t*) malloc(sizeof(gid_t) * cursups_size);
        if (cursups_list == NULL) {
            /* handle error */
        }
        if (-1 == getgroups(cursups_size, cursups_list)) {
            /* handle error */
        }
        if (!eq1_sups(cursups_size, cursups_list, targetsups_size, targetsups_list)) {
            if (-1 == setgroups(targetsups_size, targetsups_list)) { /* will probably fail... :( */
                /* handle error */
            }
        }
        free(cursups_list);
    }
#endif
    free(targetsups_list);
#endif
    return 0;
}

```

Risk Assessment

Failing to observe the correct revocation order while relinquishing privileges allows an attacker to regain elevated privileges.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
POS36-C	high	probable	medium	P12	L1

Related Guidelines

ISO/IEC TR 24772	Privilege Sandbox Issues [XY0]
MITRE CWE	CWE-250 , Execution with unnecessary privileges CWE-696 , Incorrect behavior order

Bibliography

[Chen 2002]	"Setuid Demystified"
[Dowd 2006]	Chapter 9, "UNIX I: Privileges and Files"
[Open Group 2004]	setuid() setgid()
[Tsafrir 2008]	"The Murky Issue of Changing Process Identity: Revising 'Setuid Demystified'"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/dgL7>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
check_privilege_drop	setgid(getgid()) call must be used before a setuid(getuid()) call

CertC-POS37

Ensure that privilege relinquishment is successful.

Input: IR

Source languages: C, C++

Details

The POSIX `setuid()` function has complex semantics and platform-specific behavior [[Open Group 2004](#)].

If the process has appropriate privileges, `setuid()` shall set the real user ID, effective user ID, and the saved set-user-ID of the calling process to `uid`.

If the process does not have appropriate privileges, but `uid` is equal to the real user ID or the saved set-user-ID, `setuid()` shall set the effective user ID to `uid`; the real user ID and saved set-user-ID shall remain unchanged.

The meaning of "appropriate privileges" varies from platform to platform. For example, on Solaris, appropriate privileges for `setuid()` means that the `PRIV_PROC_SETID` privilege is in the effective privilege set of the process. On BSD, it means that the effective user ID (EUID) is zero (that is, the process is running as root) or that `uid==geteuid()`. On Linux, it means that the process has `CAP_SETUID` capability and that `setuid(geteuid())` will fail if the EUID is not equal to 0, the real user ID (RUID), or the saved set-user ID (SSUID).

Because of this complex behavior, desired privilege drops sometimes may fail. For example, the range of Linux Kernel versions [2.2.0-2.2.15] is vulnerable to an insufficient privilege attack wherein `setuid(getuid())` did not drop privileges as expected when the capability bits were set to zero. As a precautionary measure, subtle behavior and error conditions for the targeted implementation must be carefully noted.

Noncompliant Code Example

This noncompliant code example compiles cleanly on most POSIX systems, but no explicit checks are made to ensure that privilege relinquishment has succeeded. This may be dangerous depending on the sequence of the preceding privilege changes.

```
/* Code intended to run with elevated privileges */

/* Temporarily drop privileges */
if (seteuid(getuid()) != 0) {
    /* Handle error */
}

/* Code intended to run with lower privileges */

if (need_more_privileges) {
    /* Restore privileges */
    if (seteuid(0) != 0) {
        /* Handle error */
    }
}

/* Code intended to run with elevated privileges */
}

/* ... */

/* Permanently drop privileges */
if (setuid(getuid()) != 0) {
    /* Handle error */
}

/*
 * Code intended to run with lower privileges,
 * but if privilege relinquishment failed,
 * attacker can regain elevated privileges!
*/
```

If the program is run as a setuid root program, over time, the state of the UIDs might look like the following:

Description	Code	EUID	RUID	SSUID
Program startup		0	User	0
Temporary drop	<code>seteuid(getuid())</code>	User	User	0
Restore	<code>seteuid(0)</code>	0	User	0
Permanent drop	<code>setuid(getuid())</code>	User	User	User
Restore (attacker)	<code>setuid(0) {fails}</code>	User	User	User

If the program fails to restore privileges, it will be unable to permanently drop them later:

Description	Code	EUID	RUID	SSUID
program startup		0	User	0
Temporary drop	<code>seteuid(getuid())</code>	User	User	0
Restore	<code>seteuid(0)</code>	User	User	0
Permanent drop	<code>setuid(getuid())</code>	User	User	0
Restore (attacker)	<code>setuid(0)</code>	0	0	0

Compliant Solution

This compliant solution was implemented in sendmail, a popular mail transfer agent, to determine if superuser privileges were successfully dropped [Wheeler 2003]. If the `setuid()` call succeeds after (supposedly) dropping privileges permanently, then the privileges were not dropped as intended.

```
/* Code intended to run with elevated privileges */

/* Temporarily drop privileges */
if (seteuid(getuid()) != 0) {
    /* Handle error */
}

/* Code intended to run with lower privileges */

if (need_more_privileges) {
    /* Restore Privileges */
    if (seteuid(0) != 0) {
        /* Handle error */
    }
}

/* Code intended to run with elevated privileges */
}

/* ... */

/* Permanently drop privileges */
if (setuid(getuid()) != 0) {
    /* Handle error */
}

if (setuid(0) != -1) {
    /* Privileges can be restored, handle error */
}

/*
 * Code intended to run with lower privileges;
 * attacker cannot regain elevated privileges
*/
```

Compliant Solution

A better solution is to ensure that proper privileges exist before attempting to perform a permanent drop:

```
/* Store the privileged ID for later verification */
uid_t privid = geteuid();

/* Code intended to run with elevated privileges */

/* Temporarily drop privileges */
if (seteuid(getuid()) != 0) {
    /* Handle error */
}

/* Code intended to run with lower privileges */

if (need_more_privileges) {
    /* Restore Privileges */
    if (seteuid(privid) != 0) {
        /* Handle error */
    }
}

/* Code intended to run with elevated privileges */
```

```

}

/* ... */

/* Restore privileges if needed */
if (geteuid() != privid) {
    if (seteuid(privid) != 0) {
        /* Handle error */
    }
}

/* Permanently drop privileges */
if (setuid(getuid()) != 0) {
    /* Handle error */
}

if (setuid(0) != -1) {
    /* Privileges can be restored, handle error */
}

/*
 * Code intended to run with lower privileges;
 * attacker cannot regain elevated privileges
*/

```

Supplementary Group IDs

A process may have a number of supplementary group IDs, in addition to its effective group ID, and the supplementary groups can allow privileged access to files. The `getgroups()` function returns an array that contains the supplementary group IDs and can also contain the effective group ID. The `setgroups()` function can set the supplementary group IDs and can also set the effective group ID on some systems. Using `setgroups()` usually requires privileges. Although POSIX defines the `getgroups()` function, it does not define `setgroups()`.

Under normal circumstances, `setuid()` and related calls do not alter the supplementary group IDs. However, a setuid-root program can alter its supplementary group IDs and then relinquish root privileges, in which case, it maintains the supplementary group IDs but lacks the privilege necessary to relinquish them. Consequently, it is recommended that a program immediately relinquish supplementary group IDs before relinquishing root privileges.

[POS36-C. Observe correct revocation order while relinquishing privileges](#) discusses how to drop supplementary group IDs. To ensure that supplementary group IDs are indeed relinquished, you can use the following `eql_sups` function:

```

/* Returns nonzero if the two group lists are equivalent (taking into
   account that the lists may differ wrt the egid *)
int eql_sups(const int cursups_size, const gid_t* const cursups_list,
             const int targetsups_size, const gid_t* const targetsups_list) {
    int i;
    int j;
    const int n = targetsups_size;
    const int diff = cursups_size - targetsups_size;
    const gid_t egid = getegid();
    if (diff > 1 || diff < 0 ) {
        return 0;
    }
    for (i=0, j=0; i < n; i++, j++) {
        if (cursups_list[j] != targetsups_list[i]) {
            if (cursups_list[j] == egid) {
                i--; /* skipping j */
            } else {
                return 0;
            }
        }
    }
    /* If reached here, we're sure i==targetsups_size. Now, either
       j==cursups_size (skipped the egid or it wasn't there), or we didn't
       get to the egid yet because it's the last entry in cursups */
    return j == cursups_size ||
           (j+1 == cursups_size && cursups_list[j] == egid);
}

```

System-Specific Capabilities

Many systems have nonportable privilege capabilities that, if unchecked, can yield privilege escalation vulnerabilities. The following section describes one such capability.

File System Access Privileges [Linux]

Processes on Linux have two additional values called `fsuid` and `fsgid`. These values indicate the privileges used when accessing files on the file system. They normally shadow the effective user ID and effective group ID, but the `setfsuid()` and `setfsgid()` functions allow them to be changed. Because changes to the `euid` and `egid` normally also apply to `fsuid` and `fsgid`, a program relinquishing root privileges need not be concerned with setting `fsuid` or `fsgid` to safe values. However, there has been at least one kernel bug that violated this invariant ([[Chen 2002](#)] and [[Tsafrir 2008](#)]). Consequently, a prudent program checks that `fsuid` and `fsgid` have harmless values after relinquishing privileges.

Risk Assessment

If privilege relinquishment conditions are left unchecked, any flaw in the program may lead to unintended system compromise corresponding to the more privileged user or group account.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
POS37-C	high	probable	low	P18	L1

Related Guidelines

ISO/IEC TR 24772	Privilege Sandbox Issues [XYO]
MITRE CWE	CWE-250 , Execution with unnecessary privileges CWE-273 , Failure to check whether privileges were dropped successfully

Bibliography

[Chen 2002]	"Setuid Demystified"
[Dowd 2006]	Chapter 9, "Unix I: Privileges and Files"
[Open Group 2004]	setuid() getuid() seteuid()
[Tsafrir 2008]	"The Murky Issue of Changing Process Identity: Revising 'Setuid Demystified'"
[Wheeler 2003]	Section 7.4, "Minimize Privileges"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/WIAAAQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
check_privilege_drop	setuid(getuid()) call must be followed by a setuid(0) != -1 check.

CertC-POS39

Use the correct byte ordering when transferring data between systems.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Different system architectures use different byte ordering, either little endian (least significant byte first) or big endian (most significant byte first). IA-32 is an example of an architecture that implements little endian byte ordering. In contrast, PowerPC and most Network Protocols (including TCP and IP) use big endian.

When transferring data between systems of different endianness, the programmer must take care to reverse the byte ordering before interpreting the data.

The functions `htonl()`, `htons()`, `ntohl()`, and `ntohs()` can be used to transfer between network byte ordering (big endian) and the host's byte ordering. On big endian systems, these functions do nothing. They may also be implemented as macros rather than functions.

Noncompliant Code Example

In this noncompliant code example, the programmer tries to read an unsigned 32-bit integer off a previously connected network socket.

It is important to know the sizes of your data types lest they be different on architectures that are accessible over the network. Hence, we transfer a `uint32_t` rather than an `int`. For more information, see [FI009-C. Be careful with binary data when transferring data across systems](#).

```
/* sock is a connected TCP socket */  
uint32_t num;  
if (recv(sock, (void *)&num, sizeof(uint32_t), 0) < (int)sizeof(uint32_t)) {
```

```

    /* Handle error */
}

printf("We received %u from the network!\n", (unsigned int)num);

```

This program prints out the number received from the socket using an incorrect byte ordering. For example, if the value 4 is sent from a big endian machine, and the receiving system is little endian, the value 536,870,912 is read. This problem can be corrected by sending and receiving using network byte ordering.

Compliant Solution

In this compliant solution, the programmer uses the `ntohl()` function to convert the integer from network byte order to host byte ordering:

```

/* sock is a connected TCP socket */

uint32_t num;

if (recv(sock, (void *)&num, sizeof(uint32_t), 0) < (int)sizeof(uint32_t)) {
    /* Handle error */
}

num = htonl(num);
printf("We received %u from the network!\n", (unsigned int)num);

```

The `htonl()` function [network to host long] translates a `uint32_t` value into the host byte ordering from the network byte ordering. This function is always appropriate to use because its implementation depends on the specific system's byte ordering. Consequently, on a big endian architecture, `htonl()` does nothing.

The reciprocal function `htons()` [host to network long] should be used before sending any data to another system over network protocols.

Portability Details

- `ntohs()`, `ntohl()`, `htons()`, and `htonl()` are not part of the C Standard and are consequently not guaranteed to be portable to non-POSIX systems.
- The POSIX implementations of `ntohs()`, `ntohl()`, `htons()`, and `htonl()` take arguments of types `uint16_t` and `uint32_t` and can be found in the header file `<arpa/inet.h>`.
- The Windows implementations use `unsigned short` and `unsigned long` and can be found in the header file `<winsock2.h>`.
- Other variants of `ntoh()` and `hton()`, such as `ntohi()`/`hton()` or `ntohll()`/`htonll()`, may exist on some systems.

Risk Assessment

If the programmer is careless, this bug is likely. However, it will immediately break the program by printing the incorrect result and therefore should be caught by the programmer during the early stages of debugging and testing. Recognizing a value as in reversed byte ordering, however, can be difficult depending on the type and magnitude of the data.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
POS39-C	Medium	Likely	Low	P18	L1

Bibliography

[MSDN]	"Winsock Functions"
[Open Group 2004]	htonl, htons, ntohs, ntohs-Convert Values between Host and Network Byte Order

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/IgDAAQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
functions_to_follow_receive	Function which should be used after receiving data from a network.	['ntohl', ' ntohs']
functions_to_precede_send	Function which should be used before sending data to a network.	['htonl', ' htons']
functions_to_receive	Functions which are used to receive data from a network.	['recv', 'recvfrom', 'recvmsg']
functions_to_send	Functions which are used to send data to a network.	['send', 'sendmsg', 'sendto']
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
correct_byte_order_receive	Use e.g. 'ntohl' after receiving data from a network by e.g. 'recv'.
correct_byte_order_send	Use e.g. 'htonl' before sending data from a network by e.g. 'send'.

CertC-POS47

Do not use threads that can be canceled asynchronously.

Input: IR

Source languages: C, C++

Details

In threading, `pthread`s can optionally be set to cancel immediately or defer until a specific cancellation point. Canceling asynchronously (immediately) is dangerous, however, because most threads are in fact not safe to cancel immediately.

The [IEEE standards page](#) states that

only functions that are cancel-safe may be called from a thread that is asynchronously cancelable.

Canceling asynchronously would follow the same route as passing a signal into the thread to kill it, posing problems similar to those in [CON37-C. Do not call signal\(\) in a multithreaded program](#), which is strongly related to [SIG02-C. Avoid using signals to implement normal functionality](#). POS44-C and SIG02-C expand on the dangers of canceling a thread suddenly, which can create a [data race condition](#).

Noncompliant Code Example

In this noncompliant code example, the worker thread is doing something as simple as swapping `a` and `b` repeatedly.

This code uses one lock. The `global_lock` mutex ensures that the worker thread and main thread do not collide in accessing the `a` and `b` variables.

The worker thread repeatedly exchanges the values of `a` and `b` until it is canceled by the main thread. The main thread then prints out the current values of `a` and `b`. Ideally, one should be 5, and the other should be 10.

```

volatile int a = 5;
volatile int b = 10;

/* Lock to enable threads to access a and b safely */
pthread_mutex_t global_lock = PTHREAD_MUTEX_INITIALIZER;

void* worker_thread(void* dummy) {
    int i;
    int c;
    int result;

    if ((result = pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,&i)) != 0) {
        /* handle error */
    }

    while (1) {
        if ((result = pthread_mutex_lock(&global_lock)) != 0) {
            /* handle error */
        }
        c = b;
        b = a;
        a = c;
    }
}
```

```

    if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
        /* handle error */
    }
    return NULL;
}

int main(void) {
    int result;
    pthread_t worker;

    if ((result = pthread_create(&worker, NULL, worker_thread, NULL)) != 0) {
        /* handle error */
    }

    /* .. Do stuff...meanwhile worker thread runs for some time */

    /* since we don't know when the character is read in, the program could continue at any time */
    if ((result = pthread_cancel(worker)) != 0) {
        /* handle error */
    }

    /* pthread_join waits for the thread to finish up before continuing */
    if ((result = pthread_join(worker, 0)) != 0) {
        /* handle error */
    }

    if ((result = pthread_mutex_lock(&global_lock)) != 0) {
        /* handle error */
    }
    printf("a: %i | b: %i", a, b);
    if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
        /* handle error */
    }

    return 0;
}

```

However, this program is subject to a race condition because an asynchronous cancel can happen at any time. If the worker thread is canceled while the `global_lock` mutex is held, it is never actually released. In this case, the main thread will wait forever trying to acquire the `global_lock`, and the program will deadlock.

It is also possible that the main thread cancels the worker thread before it has invoked `pthread_setcanceltype()`. If this happens, the cancellation will be delayed until the worker thread calls `pthread_setcanceltype()`.

Noncompliant Code Example

In this example, the worker thread arranges to release the `global_lock` mutex if it gets interrupted:

```

void release_global_lock(void* dummy) {
    int result;
    if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
        /* handle error */
    }
}

void* worker_thread(void* dummy) {
    int i;
    int c;
    int result;

    if ((result = pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,&i)) != 0) {
        /* handle error */
    }

    while (1) {
        if ((result = pthread_mutex_lock(&global_lock)) != 0) {
            /* handle error */
        }
        pthread_cleanup_push( release_global_lock, NULL);
        c = b;
        b = a;
        a = c;
        pthread_cleanup_pop(1);
    }
    return NULL;
}

```

The global variables are still subject to a race condition because an asynchronous cancel can happen at any time. For instance, the worker thread could be canceled just before the last line (`a = c`) and thereby lose the old value of `b`. Consequently, the main thread might print that `a` and `b` have the same value.

The program is still subject to the race condition where the main thread cancels the worker thread before it has invoked `pthread_setcanceltype()`. If this happens, the cancellation will be delayed until the worker thread calls `pthread_setcanceltype()`.

Furthermore, though less likely, the program can still deadlock if the worker thread gets canceled after the `global_lock` is acquired but before `pthread_cleanup_push()` is invoked. In this case, the worker thread is canceled while holding `global_lock`, and the program will deadlock.

Compliant Solution

From [IEEE standards page](#):

The cancelability state and type of any newly created threads, including the thread in which `main()` was first invoked, shall be `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DEFERRED` respectively.

Because the default condition for POSIX, according to the IEEE standards, is `PTHREAD_CANCEL_DEFERRED`, it is not necessary to invoke `pthread_setcanceltype()` in the compliant solution:

```
void* worker_thread(void* dummy) {
    int c;
    int result;

    while (1) {
        if ((result = pthread_mutex_lock(&global_lock)) != 0) {
            /* handle error */
        }
        c = b;
        b = a;
        a = c;
        if ((result = pthread_mutex_unlock(&global_lock)) != 0) {
            /* handle error */
        }

        /* now we're safe to cancel, creating cancel point */
        pthread_testcancel();
    }
    return NULL;
}
```

Because this code limits cancellation of the worker thread to the end of the `while` loop, the worker thread can preserve the data invariant that `a != b`. Consequently, the program might print that `a` is 5 and `b` is 10 or that `a` is 10 and `b` is 5, but they will always be revealed to have different values when the worker thread is canceled.

The other race conditions that plague the noncompliant code examples are not possible here. Because the worker thread does not modify its cancel type, it cannot be canceled before being improperly initialized. And because it cannot be canceled while the `global_lock` mutex is held, there is no possibility of deadlock, and the worker thread does not need to register any cleanup handlers.

Risk Assessment

Incorrectly using threads that asynchronously cancel may result in silent corruption, resource leaks, and, in the worst case, unpredictable interactions.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
POS47-C	Medium	Probable	Low	P12	L1

Related Guidelines

[SEI CERT Oracle Coding Standard for Java: THI05-J. Do not use Thread.stop\(\) to terminate threads](#)

In Java, similar reasoning resulted in the deprecation of `Thread.stop()`.

Bibliography

[MKS]	pthread_cancel() Man Page
[Open Group 2004]	Threads Overview

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/UICjAg>], Copyright [C] 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	12
pthread_functions	Function names which use the forbidden 'PTHREAD_CANCEL_ASYNCHRONOUS' as first parameter.	['pthread_setcanceltype', 'pthread_setcancelstate']
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
pthread_cancel_async	Do not use threads that can be canceled asynchronously.

CertC-POS49

When data must be accessed by multiple threads, provide a mutex and guarantee no adjacent data is also accessed.

Input: IR

Source languages: C, C++

Details

When multiple threads must access or make modifications to a common variable, they may also inadvertently access other variables adjacent in memory. This is an artifact of variables being stored compactly, with one byte possibly holding multiple variables, and is a common optimization on word-addressed machines. Bit-fields are especially prone to this behavior because compilers are allowed to store multiple bit-fields in one addressable byte or word. This implies that race conditions may exist not just on a variable accessed by multiple threads but also on other variables sharing the same byte or word address. This recommendation is a specific instance of [CON32-C. Prevent data races when accessing bit-fields from multiple threads](#) using POSIX threads.

A common tool for preventing race conditions in concurrent programming is the mutex. When properly observed by all threads, a mutex can provide safe and secure access to a common variable; however, it guarantees nothing with regard to other variables that might be accessed when a common variable is accessed.

Unfortunately, there is no portable way to determine which adjacent variables may be stored along with a certain variable.

A better approach is to embed a concurrently accessed variable inside a union, along with a `long` variable, or at least some padding to ensure that the concurrent variable is the only element to be accessed at that address. This technique would effectively guarantee that no other variables are accessed or modified when the concurrent variable is accessed or modified.

Noncompliant Code Example (Bit-field)

In this noncompliant code example, two executing threads simultaneously access two separate members of a global `struct`:

```
struct multi_threaded_flags {
    unsigned int flag1 : 2;
    unsigned int flag2 : 2;
};

struct multi_threaded_flags flags;

void thread1(void) {
    flags.flag1 = 1;
}

void thread2(void) {
    flags.flag2 = 2;
}
```

Although this code appears to be harmless, it is likely that `flag1` and `flag2` are stored in the same byte. If both assignments occur on a thread-scheduling interleaving that ends with both stores occurring after one another, it is possible that only one of the flags will be set as intended, and the other flag will equal its previous value, because both bit-fields are represented by the same byte, which is the smallest unit the processor can work on.

For example, the following sequence of events can occur:

```
Thread 1: register 0 = flags
Thread 1: register 0 &= ~mask(flag1)
Thread 2: register 0 = flags
Thread 2: register 0 &= ~mask(flag2)
Thread 1: register 0 |= 1 << shift(flag1)
Thread 1: flags = register 0
Thread 2: register 0 |= 2 << shift(flag2)
Thread 2: flags = register 0
```

Even though each thread is modifying a separate bit-field, they are both modifying the same location in memory. This is the same problem discussed in [CON43-C. Do not allow data races in multithreaded code](#) but is harder to diagnose because it is not immediately obvious that the same memory location is being modified.

Compliant Solution (Bit-field)

This compliant solution protects all accesses of the flags with a mutex, thereby preventing any thread-scheduling interleaving from occurring. In addition, the flags are declared `volatile` to ensure that the compiler will not attempt to move operations on them outside the mutex. Finally, the flags are embedded in a union alongside a `long`, and a static assertion guarantees that the flags do not occupy more space than the `long`. This technique prevents any data not checked by the mutex from being accessed or modified with the bit-fields.

```
struct multi_threaded_flags {
    volatile unsigned int flag1 : 2;
    volatile unsigned int flag2 : 2;
};

union mtf_protect {
    struct multi_threaded_flags s;
    long padding;
};

static_assert(sizeof(long) >= sizeof(struct multi_threaded_flags));

struct mtf_mutex {
    union mtf_protect u;
    pthread_mutex_t mutex;
};
```

```

struct mtf_mutex flags;

void thread1(void) {
    int result;
    if ((result = pthread_mutex_lock(&flags.mutex)) != 0) {
        /* Handle error */
    }
    flags.u.s.flag1 = 1;
    if ((result = pthread_mutex_unlock(&flags.mutex)) != 0) {
        /* Handle error */
    }
}

void thread2(void) {
    int result;
    if ((result = pthread_mutex_lock(&flags.mutex)) != 0) {
        /* Handle error */
    }
    flags.u.s.flag2 = 2;
    if ((result = pthread_mutex_unlock(&flags.mutex)) != 0) {
        /* Handle error */
    }
}

```

Static assertions are discussed in detail in [DCL03-C. Use a static assertion to test the value of a constant expression](#).

Risk Assessment

Although the race window is narrow, having an assignment or an expression evaluate improperly because of misinterpreted data can result in a corrupted running state or unintended information disclosure.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
POS49-C	Medium	Probable	Medium	P8	L2

Bibliography

[ISO/IEC 9899:2011]	Subclause 6.7.2.1, "Structure and Union Specifiers"
---------------------	---

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/eoBcBQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
access_kinds	Access kinds (e.g. Reading_Operand_Interface, Writing_Operand_Interface, Address_Operand_Interface).	['Reading_Operand_Interface', 'Writing_Operand_Interface']
allow_c11_atomics	If set, don't report races on C11 atomic variables.	True
allow_volatile_sig_atomic_t	If set, don't report races on variables of type "volatile sig_atomic_t".	False
debug_output	Option to provide diagnostic output.	False
enter_critical_functions	List of function names to enter a critical region.	[]
enter_critical_macros	List of macro names to enter a critical region (macros must expand to asm() statement).	[]
excluded_routines	List of functions that should be excluded from check.	[]
excluded_subgraphs	List of entry functions to subgraphs that should be excluded as subgraph from check.	[]
exit_critical_functions	List of function names to exit a critical region.	[]
exit_critical_macros	List of macro names to exit a critical region (macros must expand to asm() statement).	[]
inspect_pointers	Whether pointer targets should be inspected to detect more global variable uses.	False
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
nested_critical_regions	If set to True, critical regions nest; if set to False, a single exit-critical-region terminates all open critical regions.	True
output_safe_accesses	When enabled, outputs not only unsafe variable accesses, but also the safe ones.	False
partitions	Dict with partition name as key and dict as value which may contain keys 'entries' and/or 'vectors' with lists of entry points or vector table variables respectively. If special partition '*IRQ*' to configure interrupt handlers is missing, all functions not reached by any of the other options are treated as interrupt handlers.	dict(...)
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
report_cfg_based_critical_region_issues	Report unbalanced lock/unlock pairs within a routine. This has the same intention, but is slightly less strict than the purely syntactic check performed by the rule Parallelism-IncorrectCriticalSection.	False
show_identical_access	When enabled, outputs variable accesses of same kind (i.e., R/R and W/W).	True
show_object_number	Option for debugging (shows internal node numbers).	False
treat_types_as_atomic	List of type-patterns. A type-pattern is either a regular expression of a type name, or a triple of {min. alignment, max. size, type name-regex}. Each of the triple's components may be None. None is interpreted as general wildcard.	[]

Possible Messages

Name	Message
multiple_lock_add	Lock is acquired while it is already locked.
prevent-data-races	When data must be accessed by multiple threads, provide a mutex and guarantee no adjacent data is also accessed.
removed_nonexisting_lock	Lock is released, although it is not currently locked.
unbalanced_locks_path	Different control flow paths have different sets of locks.
unbalanced_locks_routine	Routine may return with different lock set than it is entered with {{in_set} vs {out_set}}.

CertC-POS54

Detect and handle POSIX library errors.

Input: IR

Source languages: C, C++

Details

All standard library functions, including I/O functions and memory allocation functions, return either a valid value or a value of the correct return type that indicates an error (for example, -1 or a null pointer). Assuming that all calls to such functions will succeed and failing to check the return value for an indication of an error is a dangerous practice that may lead to [unexpected](#) or [undefined behavior](#) when an error occurs. It is essential that programs detect and appropriately handle all errors in accordance with an error-handling policy, as discussed in [ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy](#). In addition to the C standard library functions mentioned in [ERR33-C. Detect and handle standard library errors](#), the following functions defined in POSIX require error checking (list is not all-inclusive).

The successful completion or failure of each of the standard library functions listed in the following table shall be determined either by comparing the function's return value with the value listed in the column labeled "Error Return" or by calling one of the library functions mentioned in the footnotes to the same column.

Function	Successful Return	Error Return	errno
fmemopen()	Pointer to a FILE object	NULL	ENOMEM
open_memstream()	Pointer to a FILE object	NULL	ENOMEM
posix_memalign()	0	Nonzero	Unchanged

Setting `errno` is a POSIX [[ISO/IEC 9945:2008](#)] extension to the C Standard. On error, `posix_memalign()` returns a value that corresponds to one of the constants defined in the `<errno.h>` header. The function does not set `errno`. The `posix_memalign()` function is optional and is not required to be provided by POSIX-conforming implementations.

Noncompliant Code Example (POSIX)

In this noncompliant code example, `fmemopen()` and `open_memstream()` are assumed to succeed. However, if the calls fail, the two file pointers `in` and `out` will be null and the program will have [undefined behavior](#).

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    FILE *out;
    FILE *in;
    size_t size;
    char *ptr;

    if (argc != 2) {
        /* Handle error */
    }

    in = fmemopen(argv[1], strlen(argv[1]), "r");
    /* Use in */

    out = open_memstream(&ptr, &size);
    /* Use out */

    return 0;
}
```

Compliant Solution (POSIX)

A compliant solution avoids assuming that `fmemopen()` and `open_memstream()` succeed regardless of its arguments and tests the return value of the function before using the file pointers `in` and `out`:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
```

```

FILE *out;
FILE *in;
size_t size;
char *ptr;

if (argc != 2) {
    /* Handle error */
}

in = fmemopen(argv[1], strlen(argv[1]), "r");

if (in == NULL) {
    /* Handle error */
}
/* Use in */

out = open_memstream(&ptr, &size);

if (out == NULL) {
    /* Handle error */
}
/* Use out */
return 0;
}

```

Exceptions

ERR33-C-EX1: The exception from [EXP12-C. Do not ignore values returned by functions](#) still applies. If the return value is inconsequential or if any errors can be safely ignored, such as for functions called because of their [side effects](#), the function should be explicitly cast to `void` to signify programmer intent.

ERR33-C-EX2: Ignore the return value of a function that cannot fail or whose return value cannot signify that an error condition need not be diagnosed. For example, `strcpy()` is one such function.

Return values from the following functions do not need to be checked because their historical use has overwhelmingly omitted error checking, and the consequences are not relevant to security.

Function	Successful Return	Error Return
<code>printf()</code>	Number of characters (nonnegative)	Negative
<code>putchar()</code>	Character written	<code>EOF</code>
<code>puts()</code>	Nonnegative	<code>EOF</code> (negative)
<code>putwchar()</code>	Wide character written	<code>WEOF</code>
<code>vprintf()</code>	Number of characters (nonnegative)	Negative
<code>vwprintf()</code>	Number of wide characters (nonnegative)	Negative
<code>wprintf()</code>	Number of wide characters (nonnegative)	Negative

Risk Assessment

Failing to detect error conditions can lead to unpredictable results, including [abnormal program termination](#) and [denial-of-service attacks](#) or, in some situations, could even allow an attacker to run arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
POS54-C	High	Likely	Medium	P18	L1

Related Guidelines

CERT C Secure Coding Standard	API04-C. Provide a consistent and usable error-checking mechanism ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy ERR02-C. Avoid in-band error indicators ERR05-C. Application-independent code should provide error detection without dictating error handling EXP12-C. Do not ignore values returned by functions EXP34-C. Do not dereference null pointers FI010-C. Take care when using the rename() function FI013-C. Never push back anything other than one read character FI033-C. Detect and handle input output errors resulting in undefined behavior FI034-C. Distinguish between characters read from a file and EOF or WEOF FLP03-C. Detect and handle floating-point errors FLP32-C. Prevent or detect domain and range errors in math functions MEM04-C. Do not perform zero-length allocations MEM12-C. Consider using a goto chain when leaving a function on error when using and releasing resources
SEI CERT C++ Coding Standard	ERR10-CPP. Check for error conditions FI004-CPP. Detect and handle input and output errors
ISO/IEC TS 17961	Failing to detect and handle standard library errors [liberr]
MITRE CWE	CWE-252 , Unchecked return value CWE-253 , Incorrect check of function return value CWE-390 , Detection of error condition without action CWE-391 , Unchecked error condition

Bibliography

[DHS 2006]	Handle All Errors Safely
[Henricson 1997]	Recommendation 12.1, "Check for All Errors Reported from Functions"
[ISO/IEC 9899:2011]	Subclause 7.21.7.10, "The <code>ungetc</code> Function"

Excerpt from SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition) and SEI CERT C Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/iLBfBw>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
allowed_functions		frozenset(['strncat', 'memmove', 'memset', 'strcat', 'strcpy', 'memcpy', 'strncpy'])
inspect_template_instances	Whether calls in template instances should be reported.	False
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
null_check_macro	Name of macro used to represent check for NULL	
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
discarded_return_with_entity	Return value of function discarded.
discarded_return_without_entity	Return value of function discarded.
handle_posix_error	Detect and handle POSIX library errors.

Rules in Group CertC++

CertC++-DCL30

Declare objects with appropriate storage durations.

Input: IR
Source languages: C++

Details

Every object has a storage duration that determines its lifetime: *static*, *thread*, *automatic*, or *allocated*.

According to the C Standard, 6.2.4, paragraph 2 [[ISO/IEC 9899:2011](#)],

The lifetime of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address, and retains its last-stored value throughout its lifetime. If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime.

Do not attempt to access an object outside of its lifetime. Attempting to do so is [undefined behavior](#) and can lead to an exploitable [vulnerability](#). (See also [undefined behavior 9](#) in the C Standard, Annex J.)

Noncompliant Code Example (Differing Storage Durations)

In this noncompliant code example, the address of the variable `c_str` with automatic storage duration is assigned to the variable `p`, which has static storage duration. The assignment itself is valid, but it is invalid for `c_str` to go out of scope while `p` holds its address, as happens at the end of `dont_do_this()`.

```
#include <stdio.h>

const char *p;
void dont_do_this(void) {
    const char c_str[] = "This will change";
    p = c_str; /* Dangerous */
}

void innocuous(void) {
    printf("%s\n", p);
}

int main(void) {
    dont_do_this();
    innocuous();
    return 0;
}
```

Compliant Solution (Same Storage Durations)

In this compliant solution, `p` is declared with the same storage duration as `c_str`, preventing `p` from taking on an [indeterminate value](#) outside of `this_is_OK()`:

```
void this_is_OK(void) {
    const char c_str[] = "Everything OK";
    const char *p = c_str;
    /* ... */
} /* p is inaccessible outside the scope of string c_str */
```

Alternatively, both `p` and `c_str` could be declared with static storage duration.

Compliant Solution (Differing Storage Durations)

If it is necessary for `p` to be defined with static storage duration but `c_str` with a more limited duration, then `p` can be set to `NULL` before `c_str` is destroyed. This practice prevents `p` from taking on an [indeterminate value](#), although any references to `p` must check for `NULL`.

```
const char *p;
void is_this_OK(void) {
    const char c_str[] = "Everything OK?";
    p = c_str;
    /* ... */
    p = NULL;
}
```

Noncompliant Code Example (Return Values)

In this noncompliant code sample, the function `init_array()` returns a pointer to a character array with automatic storage duration, which is accessible to the caller:

```
char *init_array(void) {
    char array[10];
    /* Initialize array */
    return array;
}
```

Some compilers generate a diagnostic message when a pointer to an object with automatic storage duration is returned from a function, as in this example. Programmers should compile code at high warning levels and resolve any diagnostic messages. (See [MSC00-C. Compile cleanly at high warning levels](#).)

Compliant Solution (Return Values)

The solution, in this case, depends on the intent of the programmer. If the intent is to modify the value of `array` and have that modification persist outside the scope of `init_array()`, the desired behavior can be achieved by declaring `array` elsewhere and passing it as an argument to `init_array()`:

```
#include <stddef.h>
void init_array(char *array, size_t len) {
    /* Initialize array */
}
```

```

    return;
}

int main(void) {
    char array[10];
    init_array(array, sizeof(array) / sizeof(array[0]));
    /* ... */
    return 0;
}

```

Noncompliant Code Example (Output Parameter)

In this noncompliant code example, the function `squirrel_away()` stores a pointer to local variable `local` into a location pointed to by function parameter `ptr_param`. Upon the return of `squirrel_away()`, the pointer `ptr_param` points to a variable that has an expired lifetime.

```

void squirrel_away(char **ptr_param) {
    char local[10];
    /* Initialize array */
    *ptr_param = local;
}

void rodent(void) {
    char *ptr;
    squirrel_away(&ptr);
    /* ptr is live but invalid here */
}

```

Compliant Solution (Output Parameter)

In this compliant solution, the variable `local` has static storage duration; consequently, `ptr` can be used to reference the `local` array within the `rodent()` function:

```

char local[10];

void squirrel_away(char **ptr_param) {
    /* Initialize array */
    *ptr_param = local;
}

void rodent(void) {
    char *ptr;
    squirrel_away(&ptr);
    /* ptr is valid in this scope */
}

```

Risk Assessment

Referencing an object outside of its lifetime can result in an attacker being able to execute arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL30-C	High	Probable	High	P6	L2

Related Guidelines

CERT C Secure Coding Standard	MSC00-C. Compile cleanly at high warning levels
SEI CERT C++ Coding Standard	EXP54-CPP. Do not access an object outside of its lifetime
ISO/IEC TR 24772:2013	Dangling References to Stack Frames [DCM]
ISO/IEC TS 17961	Escaping of the address of an automatic object [addrescape]
MISRA C:2012	Rule 18.6 (required)

Bibliography

[Coverity 2007]	
[ISO/IEC 9899:2011]	6.2.4, "Storage Durations of Objects"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/bQ4>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
global_reference_to_local_var	Address of local variable escapes via global variable.
parameter_reference_to_local_var	Address of local variable escapes via parameter.
returning_reference_to_local_var	Return of reference(pointer to local variable.

CertC++-DCL39

Avoid information leakage when passing a structure across a trust boundary.

Input: IR

Source languages: C++

Details

The C Standard, 6.7.2.1, discusses the layout of structure fields. It specifies that non-bit-field members are aligned in an [implementation-defined](#) manner and that there may be padding within or at the end of a structure. Furthermore, initializing the members of the structure does not guarantee initialization of the padding bytes. The C Standard, 6.2.6.1, paragraph 6 [[ISO/IEC 9899:2011](#)], states

When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values.

Additionally, the storage units in which a bit-field resides may also have padding bits. For an object with automatic storage duration, these padding bits do not take on specific values and can contribute to leaking sensitive information.

When passing a pointer to a structure across a trust boundary to a different trusted domain, the programmer must ensure that the padding bytes and bit-field storage unit padding bits of such a structure do not contain sensitive information.

Noncompliant Code Example

This noncompliant code example runs in kernel space and copies data from `arg` to user space. However, padding bytes may be used within the structure, for example, to ensure the proper alignment of the structure members. These padding bytes may contain sensitive information, which may then be leaked when the data is copied to user space.

```
#include <stddef.h>

struct test {
    int a;
    char b;
    int c;
};

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
    struct test arg = {.a = 1, .b = 2, .c = 3};
    copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

Noncompliant Code Example (`memset()`)

The padding bytes can be explicitly initialized by calling `memset()`:

```
#include <string.h>

struct test {
    int a;
    char b;
    int c;
};

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
    struct test arg;
    memset(&arg, 0, sizeof(arg));
    copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

```

/* Set all bytes (including padding bytes) to zero */
memset(&arg, 0, sizeof(arg));

arg.a = 1;
arg.b = 2;
arg.c = 3;

copy_to_user(usr_buf, &arg, sizeof(arg));
}

```

However, compilers are free to implement `arg.b = 2` by setting the low byte of a 32-bit register to 2, leaving the high bytes unchanged and storing all 32 bits of the register into memory. This implementation could leak the high-order bytes resident in the register to a user.

Compliant Solution

This compliant solution serializes the structure data before copying it to an untrusted context:

```

#include <stddef.h>
#include <string.h>

struct test {
    int a;
    char b;
    int c;
};

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
    struct test arg = {.a = 1, .b = 2, .c = 3};
    /* May be larger than strictly needed */
    unsigned char buf[sizeof(arg)];
    size_t offset = 0;

    memcpy(buf + offset, &arg.a, sizeof(arg.a));
    offset += sizeof(arg.a);
    memcpy(buf + offset, &arg.b, sizeof(arg.b));
    offset += sizeof(arg.b);
    memcpy(buf + offset, &arg.c, sizeof(arg.c));
    offset += sizeof(arg.c);

    copy_to_user(usr_buf, buf, offset /* size of info copied */);
}

```

This code ensures that no uninitialized padding bytes are copied to unprivileged users. The structure copied to user space is now a packed structure and the `copy_to_user()` function would need to unpack it to recreate the original padded structure.

Compliant Solution [Padding Bytes]

Padding bytes can be explicitly declared as fields within the structure. This solution is not portable, however, because it depends on the [implementation](#) and target memory architecture. The following solution is specific to the x86-32 architecture:

```

#include <assert.h>
#include <stddef.h>

struct test {
    int a;
    char b;
    char padding_1, padding_2, padding_3;
    int c;
};

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
    /* Ensure c is the next byte after the last padding byte */
    static_assert(offsetof(struct test, c) ==
                 offsetof(struct test, padding_3) + 1,
                 "Structure contains intermediate padding");
    /* Ensure there is no trailing padding */
    static_assert(sizeof(struct test) ==
                 offsetof(struct test, c) + sizeof(int),
                 "Structure contains trailing padding");
    struct test arg = {.a = 1, .b = 2, .c = 3};
    arg.padding_1 = 0;
    arg.padding_2 = 0;
    arg.padding_3 = 0;
    copy_to_user(usr_buf, &arg, sizeof(arg));
}

```

The C Standard `static_assert()` macro accepts a constant expression and an [error message](#). The expression is evaluated at compile time and, if false, the compilation is terminated and the error message is output. (See [DCL03-C. Use a static assertion to test the value of a constant expression](#) for more details.) The explicit insertion of the padding bytes into the `struct` should ensure that no additional padding bytes are added by the compiler and consequently both static assertions should be true. However, it is necessary to validate these assumptions to ensure that the solution is correct for a particular implementation.

Compliant Solution [Structure Packing-GCC]

GCC allows specifying declaration attributes using the keyword `__attribute__((__packed__))`. When this attribute is present, the compiler will not add padding bytes for memory alignment unless otherwise required by the `_Alignas` alignment specifier, and it will attempt to place fields at adjacent memory offsets when possible.

```
#include <stddef.h>

struct test {
    int a;
    char b;
    int c;
} __attribute__((__packed__));

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
    struct test arg = { .a = 1, .b = 2, .c = 3 };
    copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

Compliant Solution [Structure Packing-Microsoft Visual Studio]

Microsoft Visual Studio supports `#pragma pack()` to suppress padding bytes [[MSDN](#)]. The compiler adds padding bytes for memory alignment, depending on the current packing mode, but still honors the alignment specified by `__declspec(align())`. In this compliant solution, the packing mode is set to 1 in an attempt to ensure all fields are given adjacent offsets:

```
#include <stddef.h>

#pragma pack(push, 1) /* 1 byte */
struct test {
    int a;
    char b;
    int c;
};
#pragma pack(pop)

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
    struct test arg = {1, 2, 3};
    copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

The `pack` pragma takes effect at the first `struct` declaration after the pragma is seen.

Noncompliant Code Example

This noncompliant code example also runs in kernel space and copies data from `struct test` to user space. However, padding bits will be used within the structure due to the bit-field member lengths not adding up to the number of bits in an `unsigned` object. Further, there is an unnamed bit-field that causes no further bit-fields to be packed into the same storage unit. These padding bits may contain sensitive information, which may then be leaked when the data is copied to user space. For instance, the uninitialized bits may contain a sensitive kernel space pointer value that can be trivially reconstructed by an attacker in user space.

```
#include <stddef.h>

struct test {
    unsigned a : 1;
    unsigned : 0;
    unsigned b : 4;
};

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
    struct test arg = { .a = 1, .b = 10 };
    copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

However, compilers are free to implement the initialization of `arg.a` and `arg.b` by setting the low byte of a 32-bit register to the value specified, leaving the high bytes unchanged and storing all 32 bits of the register into memory. This implementation could leak the high-order bytes resident in the register to a user.

Compliant Solution

Padding bits can be explicitly declared, allowing the programmer to specify the value of those bits. When explicitly declaring all of the padding bits, any unnamed bit-fields of length 0 must be removed from the structure because the explicit padding bits ensure that no further bit-fields will be packed into the same storage unit.

```
#include <assert.h>
#include <limits.h>
#include <stddef.h>

struct test {
    unsigned a : 1;
    unsigned padding1 : sizeof(unsigned) * CHAR_BIT - 1;
    unsigned b : 4;
    unsigned padding2 : sizeof(unsigned) * CHAR_BIT - 4;
};
/* Ensure that we have added the correct number of padding bits. */
static_assert(sizeof(struct test) == sizeof(unsigned) * 2,
             "Incorrect number of padding bits for type: unsigned");

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);
```

```

void do_stuff(void *usr_buf) {
    struct test_arg = { .a = 1, .padding1 = 0, .b = 10, .padding2 = 0 };
    copy_to_user(usr_buf, &arg, sizeof(arg));
}

```

This solution is not portable, however, because it depends on the [implementation](#) and target memory architecture. The explicit insertion of padding bits into the `struct` should ensure that no additional padding bits are added by the compiler. However, it is still necessary to validate these assumptions to ensure that the solution is correct for a particular implementation. For instance, the DEC Alpha is an example of a 64-bit architecture with 32-bit integers that allocates 64 bits to a storage unit.

In addition, this solution assumes that there are no integer padding bits in an `unsigned int`. The portable version of the width calculation from [INT35-C. Use correct integer precisions](#) cannot be used because the bit-field width must be an integer constant expression.

From this situation, it can be seen that special care must be taken because no solution to the bit-field padding issue will be 100% portable.

Risk Assessment

Padding units might contain sensitive data because the C Standard allows any padding to take [unspecified values](#). A pointer to such a structure could be passed to other functions, causing information leakage.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL39-C	Low	Unlikely	High	P1	L3

Related Guidelines

CERT C Secure Coding Standard	DCL03-C. Use a static assertion to test the value of a constant expression
---	--

Bibliography

[ISO/IEC 9899:2011]	6.2.6.1, "General" 6.7.2.1, "Structure and Union Specifiers"
[Graff 2003]	
[Sun 1993]	

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/IABLAW>], Copyright [C] 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	1
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high
trust_boundary_functions	When names of functions are provided in this set, only calls to these functions are reported when a pointer to some struct/class with padding is passed as argument	set([])

Possible Messages

Name	Message
composite_with_padding	Composite type has padding after field '{}'.
pointer_to_padded_composite_as_argument	Pointer to composite type '{}' with padding after field '{}' is passed as argument: potential information leak.

CertC++-DCL40

Do not create incompatible declarations of the same function or object.

Input: IR

Source languages: C++

Details

Two or more incompatible declarations of the same function or object must not appear in the same program because they result in [undefined behavior](#). The C Standard, 6.2.7, mentions that two types may be distinct yet compatible and addresses precisely when two distinct types are compatible.

The C Standard identifies four situations in which [undefined behavior \[UB\]](#) may arise as a result of incompatible declarations of the same function or object:

UB	Description	Code
15	<i>Two declarations of the same object or function specify types that are not compatible (6.2.7).</i>	All noncompliant code in this guideline
31	<i>Two identifiers differ only in nonsignificant characters (6.4.2.1).</i>	Excessively Long Identifiers
37	<i>An object has its stored value accessed other than by an lvalue of an allowable type (6.5).</i>	Incompatible Object Declarations Incompatible Array Declarations
41	<i>A function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function (6.5.2.2).</i>	Incompatible Function Declarations Excessively Long Identifiers

Although the effect of two incompatible declarations simply appearing in the same program may be benign on most [implementations](#), the effects of invoking a function through an expression whose type is incompatible with the function definition are typically catastrophic. Similarly, the effects of accessing an object using an [lvalue](#) of a type that is incompatible with the object definition may range from unintended information exposure to memory overwrite to a hardware trap.

Noncompliant Code Example {Incompatible Object Declarations}

In this noncompliant code example, the variable `i` is declared to have type `int` in file `a.c` but defined to be of type `short` in file `b.c`. The declarations are incompatible, resulting in [undefined behavior 15](#). Furthermore, accessing the object using an [lvalue](#) of an incompatible type, as shown in function `f()`, is [undefined behavior 37](#) with possible observable results ranging from unintended information exposure to memory overwrite to a hardware trap.

```
/* In a.c */
extern int i; /* UB 15 */

int f(void) {
    return ++i; /* UB 37 */
}

/* In b.c */
short i; /* UB 15 */
```

Compliant Solution {Incompatible Object Declarations}

This compliant solution has compatible declarations of the variable `i`:

```
/* In a.c */
extern int i;

int f(void) {
    return ++i;
}

/* In b.c */
int i;
```

Noncompliant Code Example {Incompatible Array Declarations}

In this noncompliant code example, the variable `a` is declared to have a pointer type in file `a.c` but defined to have an array type in file `b.c`. The two declarations are incompatible, resulting in [undefined behavior 15](#). As before, accessing the object in function `f()` is [undefined behavior 37](#) with the typical effect of triggering a hardware trap.

```
/* In a.c */
extern int *a; /* UB 15 */

int f(unsigned int i, int x) {
    int tmp = a[i]; /* UB 37: read access */
    a[i] = x; /* UB 37: write access */
    return tmp;
}

/* In b.c */
int a[] = { 1, 2, 3, 4 }; /* UB 15 */
```

Compliant Solution {Incompatible Array Declarations}

This compliant solution declares `a` as an array in `a.c` and `b.c`:

```
/* In a.c */
extern int a[];

int f(unsigned int i, int x) {
    int tmp = a[i];
    a[i] = x;
    return tmp;
}
```

```
/* In b.c */
int a[] = { 1, 2, 3, 4 };
```

Noncompliant Code Example (Incompatible Function Declarations)

In this noncompliant code example, the function `f()` is declared in file `a.c` with one prototype but defined in file `b.c` with another. The two prototypes are incompatible, resulting in [undefined behavior 15](#). Furthermore, invoking the function is [undefined behavior 41](#) and typically has catastrophic consequences.

```
/* In a.c */
extern int f(int a); /* UB 15 */

int g(int a) {
    return f(a); /* UB 41 */
}

/* In b.c */
long f(long a) { /* UB 15 */
    return a * 2;
}
```

Compliant Solution (Incompatible Function Declarations)

This compliant solution has compatible prototypes for the function `f()`:

```
/* In a.c */
extern int f(int a);

int g(int a) {
    return f(a);
}

/* In b.c */
int f(int a) {
    return a * 2;
}
```

Noncompliant Code Example (Incompatible Variadic Function Declarations)

In this noncompliant code example, the function `buginf()` is defined to take a variable number of arguments and expects them all to be signed integers with a sentinel value of `-1`:

```
/* In a.c */
void buginf(const char *fmt, ...) {
    /* ... */
}

/* In b.c */
void buginf();
```

Although this code appears to be well defined because of the prototype-less declaration of `buginf()`, it exhibits [undefined behavior](#) in accordance with the C Standard, 6.7.6.3, paragraph 15 [[ISO/IEC 9899:2011](#)].

For two function types to be compatible, both shall specify compatible return types. Moreover, the parameter type lists, if both are present, shall agree in the number of parameters and in use of the ellipsis terminator; corresponding parameters shall have compatible types. If one type has a parameter type list and the other type is specified by a function declarator that is not part of a function definition and that contains an empty identifier list, the parameter list shall not have an ellipsis terminator and the type of each parameter shall be compatible with the type that results from the application of the default argument promotions.

Compliant Solution (Incompatible Variadic Function Declarations)

In this compliant solution, the prototype for the function `buginf()` is included in the scope in the source file where it will be used:

```
/* In a.c */
void buginf(const char *fmt, ...) {
    /* ... */
}

/* In b.c */
void buginf(const char *fmt, ...);
```

Noncompliant Code Example (Excessively Long Identifiers)

In this noncompliant code example, the length of the identifier declaring the function pointer `bash_groupname_completion_function()` in the file `bashline.h` exceeds by 3 the minimum implementation limit of 31 significant initial characters in an external identifier. This introduces the possibility of colliding with the `bash_groupname_completion_funct` integer variable defined in file `b.c`, which is exactly 31 characters long. On an implementation that exactly meets this limit, this is [undefined behavior 31](#). It results in two incompatible declarations of the same function. (See [undefined behavior 15](#).) In addition, invoking the function leads to [undefined behavior 41](#) with typically catastrophic effects.

```
/* In bashline.h */
/* UB 15, UB 31 */
extern char * bash_groupname_completion_function(const char *, int);

/* In a.c */
#include "bashline.h"

void f(const char *s, int i) {
    bash_groupname_completion_function(s, i); /* UB 41 */
}
```

```
/* In b.c */
int bash_groupname_completion_funct; /* UB 15, UB 31 */
```

NOTE: The identifier `bash_groupname_completion_function` referenced here was taken from GNU [Bash](#), version 3.2.

Compliant Solution [Excessively Long Identifiers]

In this compliant solution, the length of the identifier declaring the function pointer `bash_groupname_completion()` in `bashline.h` is less than 32 characters. Consequently, it cannot clash with `bash_groupname_funct` on any compliant platform.

```
/* In bashline.h */
extern char * bash_groupname_completion(const char *, int);

/* In a.c */
#include "bashline.h"

void f(const char *s, int i) {
    bash_groupname_completion(s, i);
}

/* In b.c */
int bash_groupname_completion_funct;
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL40-C	Low	Unlikely	Medium	P2	L3

Related Guidelines

ISO/IEC TS 17961	Declaring the same function or object in incompatible ways [funcdecl]
MISRA C:2012	Rule 8.4 (required)

Bibliography

[Hatton 1995]	Section 2.8.3
[ISO/IEC 9899:2011]	6.7.6.3, "Function Declarators (including Prototypes)" J.2, "Undefined Behavior"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/cwGTAW>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
check_undefined	Whether only-declared variables should also be checked.	False
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
maxlen		31
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
report_external_identifiers	Whether external identifiers should be compared to each other.	True
report_internal_identifiers	Whether internal identifiers should be compared to each other (including macros).	False
report_short_identifiers	If True, identifiers shorter than maxlen are considered as well.	False
reported_messages	If provided, only messages of these types are reported.	147
reported_severities	List of severities to display.	('error', 'warning', 'remark')
require_exact_match	Whether to check for identical or compatible types.	True
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

Possible Messages

Name	Message
decl_return_type_mismatch	Return type at declarations differs
decl_type_mismatch	Type at declarations differs
external_identifiers_not_distinct	External identifiers not distinct.
external_identifiers_sharing	External identifiers sharing first {} characters.
incompatible_parameters	Parameters of definition and declarations of a function shall be compatible
internal_identifiers_not_distinct	Internal identifiers not distinct.
internal_identifiers_sharing	Internal identifiers sharing first {} characters.
return_type_mismatch	Return type at definition differs from type at declaration
type_mismatch	Type at definition differs from type at declaration

CertC++-DCL50

Do not define a C-style variadic function.

Input: IR

Source languages: C++

Details

Functions can be defined to accept more formal arguments at the call site than are specified by the parameter declaration clause. Such functions are called *variadic* functions because they can accept a variable number of arguments from a caller. C++ provides two mechanisms by which a variadic function can be defined: function parameter packs and use of a C-style ellipsis as the final parameter declaration.

Variadic functions are flexible because they accept a varying number of arguments of differing types. However, they can also be hazardous. A variadic function using a C-style ellipsis (hereafter called a *C-style variadic function*) has no mechanisms to check the type safety of arguments being passed to the function or to check that the number of arguments being passed matches the semantics of the function definition. Consequently, a runtime call to a C-style variadic function that passes inappropriate arguments yields undefined behavior. Such [undefined behavior](#) could be [exploited](#) to run arbitrary code.

Do not define C-style variadic functions. (The declaration of a C-style variadic function that is never defined is permitted, as it is not harmful and can be useful in unevaluated contexts.)

Issues with C-style variadic functions can be avoided by using variadic functions defined with function parameter packs for situations in which a variable number of arguments should be passed to a function. Additionally, function currying can be used as a replacement to variadic functions. For example, in contrast to C's `printf()` family of functions, C++ output is implemented with the overloaded single-argument `std::cout::operator<<()` operators.

Noncompliant Code Example

This noncompliant code example uses a C-style variadic function to add a series of integers together. The function reads arguments until the value `0` is found. Calling this function without passing the value `0` as an argument (after the first two arguments) results in undefined behavior. Furthermore, passing any type other than an `int` also results in undefined behavior.

```
#include <cstdarg>

int add(int first, int second, ...) {
    int r = first + second;
    va_list va;
    va_start(va, second);
    while (int v = va_arg(va, int)) {
        r += v;
    }
    va_end(va);
    return r;
}
```

Compliant Solution {Recursive Pack Expansion}

In this compliant solution, a variadic function using a function parameter pack is used to implement the `add()` function, allowing identical behavior for call sites. Unlike the C-style variadic function used in the noncompliant code example, this compliant solution does not result in undefined behavior if the list of parameters is not terminated with `0`. Additionally, if any of the values passed to the function are not integers, the code is [ill-formed](#) rather than producing undefined behavior.

```
#include <type_traits>

template <typename Arg, typename std::enable_if<std::is_integral<Arg>::value>::type * = nullptr>
int add(Arg f, Arg s) { return f + s; }

template <typename Arg, typename... Ts, typename std::enable_if<std::is_integral<Arg>::value>::type * = nullptr>
int add(Arg f, Ts... rest) {
    return f + add(rest...);
}
```

This compliant solution makes use of `std::enable_if` to ensure that any nonintegral argument value results in an ill-formed program.

Compliant Solution {Braced Initializer List Expansion}

An alternative compliant solution that does not require recursive expansion of the function parameter pack instead expands the function parameter pack into a list of values as part of a braced initializer list. Since narrowing conversions are not allowed in a braced initializer list, the type safety is preserved despite the `std::enable_if` not involving any of the variadic arguments.

```
#include <type_traits>

template <typename Arg, typename... Ts, typename std::enable_if<std::is_integral<Arg>::value>::type * = nullptr>
int add(Arg i, Arg j, Ts... all) {
    int values[] = { j, all... };
    int r = i;
    for (auto v : values) {
        r += v;
    }
    return r;
}
```

Exceptions

DCL50-CPP-EX1: It is permissible to define a C-style variadic function if that function also has external C language linkage. For instance, the function may be a definition used in a C library API that is implemented in C++.

DCL50-CPP-EX2: As stated in the normative text, C-style variadic functions that are declared but never defined are permitted. For example, when a function call expression appears in an unevaluated context, such as the argument in a `sizeof` expression, overload resolution is performed to determine the result type of the call but does not require a function definition. Some template metaprogramming techniques that employ [SFINAE](#) use variadic function declarations to implement compile-time type queries, as in the following example.

```
template <typename Ty>
class has_foo_function {
    typedef char yes[1];
    typedef char no[2];

    template <typename Inner>
    static yes& test(Inner *I, decltype(I->foo()) * = nullptr); // Function is never defined.

    template <typename>
    static no& test(...); // Function is never defined.

public:
    static const bool value = sizeof(test<Ty>(nullptr)) == sizeof(yes);
};
```

In this example, the value of `value` is determined on the basis of which overload of `test()` is selected. The declaration of `Inner *I` allows use of the

variable `i` within the `decltype` specifier, which results in a pointer of some (possibly `void`) type, with a default value of `nullptr`. However, if there is no declaration of `Inner::foo()`, the `decltype` specifier will be ill-formed, and that variant of `test()` will not be a candidate function for overload resolution due to SFINAE. The result is that the C-style variadic function variant of `test()` will be the only function in the candidate set. Both `test()` functions are declared but never defined because their definitions are not required for use within an unevaluated expression context.

Risk Assessment

Incorrectly using a variadic function can result in [abnormal program termination](#), unintended information disclosure, or execution of arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL50-CPP	High	Probable	Medium	P12	L1

Bibliography

[ISO/IEC 14882-2014]	Subclause 5.2.2, "Function Call" Subclause 14.5.3, "Variadic Templates"
--------------------------------------	--

Excerpt from SEI CERT C++ Coding Standard Wiki [\[https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL50-CPP.+Do+not+define+a+C-style+variadic+function\]](https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL50-CPP.+Do+not+define+a+C-style+variadic+function), Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
ignore_declarations	Allow declarations of functions with variable number of arguments (e.g. for existing library functions); only report definitions.	True
ignore_inherited	Do not report functions inheriting a variable number of arguments.	False
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
ellipsis_parameter	Function definitions shall not use ellipsis

CertC++-DCL51

Do not declare or define a reserved identifier.

Input: IR

Source languages: C++

Details

The C++ Standard, [reserved.names] [\[ISO/IEC 14882-2014\]](#), specifies the following rules regarding reserved names:

- A translation unit that includes a standard library header shall not `#define` or `#undef` names declared in any standard library header.
- 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 7.6.
- Each name that contains a double underscore `__` or begins with an underscore followed by an uppercase letter 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.
- 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, both in namespace `std` and in the global namespace.
- Each global function signature declared with external linkage in a header is reserved to the implementation to designate that function signature with external linkage.
- 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.
- 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, or as a name of namespace scope in the global namespace.
- For each type `T` from the Standard C library, the types `::T` and `std::T` are reserved to the implementation and, when defined, `::T` shall be identical to `std::T`.
- Literal suffix identifiers that do not start with an underscore are reserved for future standardization.

The identifiers and attribute names referred to in the preceding excerpt are `override`, `final`, `alignas`, `carries_dependency`, `deprecated`, and `noreturn`.

No other identifiers are reserved. Declaring or defining an identifier in a context in which it is reserved results in [undefined behavior](#). Do not declare or define a reserved identifier.

Noncompliant Code Example (Header Guard)

A common practice is to use a macro in a preprocessor conditional that guards against multiple inclusions of a header file. While this is a recommended practice, many programs use reserved names as the header guards. Such a name may clash with reserved names defined by the implementation of the C++ standard template library in its headers or with reserved names implicitly predefined by the compiler even when no C++ standard library header is included.

```
#ifndef __MY_HEADER_H__
#define __MY_HEADER_H__

// Contents of <my_header.h>

#endif // __MY_HEADER_H__
```

Compliant Solution (Header Guard)

This compliant solution avoids using leading or trailing underscores in the name of the header guard.

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

// Contents of <my_header.h>

#endif // MY_HEADER_H__
```

Noncompliant Code Example (User-Defined Literal)

In this noncompliant code example, a user-defined literal `operator"" x` is declared. However, literal suffix identifiers are required to start with an underscore; literal suffixes without the underscore prefix are reserved for future library implementations.

```
#include <cstddef>
unsigned int operator"" x(const char *, std::size_t);
```

Compliant Solution (User-Defined Literal)

In this compliant solution, the user-defined literal is named `operator"" _x`, which is not a reserved identifier.

```
#include <cstddef>
unsigned int operator"" _x(const char *, std::size_t);
```

The name of the user-defined literal is `operator"" _x` and not `_x`, which would have otherwise been reserved for the global namespace.

Noncompliant Code Example (File Scope Objects)

In this noncompliant code example, the names of the file scope objects `_max_limit` and `_limit` both begin with an underscore. Because it is `static`, the declaration of `_max_limit` might seem to be impervious to clashes with names defined by the implementation. However, because the header `<cstddef>` is included to define `std::size_t`, a potential for a name clash exists. (Note, however, that a conforming compiler may implicitly declare reserved names regardless of whether any C++ standard template library header has been explicitly included.) In addition, because `_limit` has external linkage, it may clash with a symbol with the same name defined in the language runtime library even if such a symbol is not declared in any header. Consequently, it is unsafe to start the name of any file scope identifier with an underscore even if its linkage limits its visibility to a single translation unit.

```
#include <cstddef> // std::size_t
static const std::size_t _max_limit = 1024;
std::size_t _limit = 100;

unsigned int get_value(unsigned int count) {
    return count < _limit ? count : _limit;
}
```

Compliant Solution (File Scope Objects)

In this compliant solution, file scope identifiers do not begin with an underscore.

```
#include <cstddef> // for size_t
static const std::size_t max_limit = 1024;
std::size_t limit = 100;

unsigned int get_value(unsigned int count) {
    return count < limit ? count : limit;
}
```

Noncompliant Code Example (Reserved Macros)

In this noncompliant code example, because the C++ standard template library header `<cinttypes>` is specified to include `<cstdint>`, as per [c.files] paragraph 4 [[ISO/IEC 14882-2014](#)], the name `MAX_SIZE` conflicts with the name of the `<cstdint>` header macro used to denote the upper limit of `std::size_t`.

```
#include <cinttypes> // for int_fast16_t
void f(std::int_fast16_t val) {
```

```

enum { MAX_SIZE = 80 };
// ...
}

```

Compliant Solution [Reserved Macros]

This compliant solution avoids redefining reserved names.

```

#include <cinttypes> // for std::int_fast16_t

void f(std::int_fast16_t val) {
    enum { BufferSize = 80 };
    // ...
}

```

Exceptions

DCL51-CPP-EX1: For compatibility with other compiler vendors or language standard modes, it is acceptable to create a macro identifier that is the same as a reserved identifier so long as the behavior is semantically identical, as in this example.

```

// Sometimes generated by configuration tools such as autoconf
#define const const

// Allowed compilers with semantically equivalent extension behavior
#define inline __inline

```

DCL51-CPP-EX2: As a compiler vendor or standard library developer, it is acceptable to use identifiers reserved for your implementation. Reserved identifiers may be defined by the compiler, in standard library headers, or in headers included by a standard library header, as in this example declaration from the [libc++](#) STL implementation.

```

// The following declaration of a reserved identifier exists in the libc++ implementation of
// std::basic_string as a public member. The original source code may be found at:
// http://llvm.org/svn/llvm-project/libcxx/trunk/include/string

template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>>
class basic_string {
    // ...

    bool __invariants() const;
};

```

Risk Assessment

Using reserved identifiers can lead to incorrect program operation.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL51-CPP	Low	Unlikely	Low	P3	L3

Related Guidelines

SEI CERT C++ Coding Standard	DCL58-CPP. Do not modify the standard namespaces
SEI CERT C Coding Standard	DCL37-C. Do not declare or define a reserved identifier PRE06-C. Enclose header files in an include guard
MISRA C++:2008	Rule 17-0-1

Bibliography

[ISO/IEC 14882-2014]	Subclause 17.6.4.3, "Reserved Names"
[ISO/IEC 9899:2011]	Subclause 7.1.3, "Reserved Identifiers"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL51-CPP.+Do+not+declare+or+define+a+reserved+identifier>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
additional_reserved_identifiers	Names listed here are seen as violations as well.	set(['assert', 'errno', 'defined'])
allow_function_declarations	Whether nondefining function declarations with library names are allowed	False
allow_reserved_identifier_as_include_guard	If True, macros used for include guards are not checked for being a reserved identifier.	False
check_for_keyword	Whether macro names being keywords should be reported as well.	True
check_locals	Whether parameters and local variables should also be checked	True
check_reserved_enum_identifier	Whether enumerator names should be checked to use a reserved identifier	True
check_reserved_function_identifier	Whether function names should be checked to use a reserved identifier	True
check_reserved_type_identifier	Whether type names should be checked to use a reserved identifier	True
check_reserved_variable_identifier	Whether variable names should be checked to use a reserved identifier	True
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
enumerator_having_libname	The names of standard library macros, objects and functions, as well as reserved identifiers, shall not be reused.
field_having_libname	The names of standard library macros, objects and functions shall not be reused.
macro_having_reserved_name	Definition of reserved identifier or standard library element
reused_macro_object_libname	The names of standard library macros and objects shall not be reused.
reused_routine_libname	The names of standard library functions shall not be overridden.
routine_having_libname	The names of standard library macros, objects and functions, as well as reserved identifiers, shall not be reused.
type_having_libname	The names of standard library macros, objects and functions shall not be reused.
undef_of_reserved_name	#undef of reserved identifier or standard library element
user_defined_literal_naming	User-defined literals shall have a suffix matching "[a-zA-Z]+".
variable_having_libname	The names of standard library macros, objects and functions, as well as reserved identifiers, shall not be reused.

CertC++-DCL52

Never qualify a reference type with const or volatile.

Input: IR

Source languages: C++

Details

C++ does not allow you to change the value of a reference type, effectively treating all references as being `const` qualified. The C++ Standard, [dcl.ref], paragraph 1 [[ISO/IEC 14882-2014](#)], states the following:

Cv-qualified references are ill-formed except when the cv-qualifiers are introduced through the use of a `typedef-name` [7.1.3, 14.1] or `decltype-specifier` [7.1.6.2], in which case the cv-qualifiers are ignored.

Thus, C++ prohibits or ignores the [cv-qualification](#) of a reference type. Only a value of non-reference type may be cv-qualified.

When attempting to `const`-qualify a type as part of a declaration that uses reference type, a programmer may accidentally write

```
char &const p;
```

instead of

```
char const &p; // Or: const char &p;
```

Do not attempt to cv-qualify a reference type because it results in [undefined behavior](#). A [conforming](#) compiler is required to issue a [diagnostic message](#). However, if the compiler does not emit a [fatal diagnostic](#), the program may produce surprising results, such as allowing the character referenced by `p` to be mutated.

Noncompliant Code Example

In this noncompliant code example, a `const`-qualified reference to a `char` is formed instead of a reference to a `const`-qualified `char`. This results in undefined behavior.

```
#include <iostream>

void f(char c) {
    char &const p = c;
    p = 'p';
    std::cout << c << std::endl;
}
```

Implementation Details (MSVC)

With [Microsoft Visual Studio](#) 2015, this code compiles successfully with a warning diagnostic.

```
warning C4227: anachronism used : qualifiers on reference are ignored
```

When run, the code outputs the following.

```
p
```

Implementation Details (Clang)

With [Clang](#) 3.9, this code produces a fatal diagnostic.

```
error: 'const' qualifier may not be applied to a reference
```

Noncompliant Code Example

This noncompliant code example correctly declares `p` to be a reference to a `const`-qualified `char`. The subsequent modification of `p` makes the program [ill-formed](#).

```
#include <iostream>

void f(char c) {
    const char &p = c;
    p = 'p'; // Error: read-only variable is not assignable
    std::cout << c << std::endl;
}
```

Compliant Solution

This compliant solution removes the `const` qualifier.

```
#include <iostream>

void f(char c) {
    char &p = c;
    p = 'p';
    std::cout << c << std::endl;
}
```

Risk Assessment

A `const` or `volatile` reference type may result in undefined behavior instead of a fatal diagnostic, causing unexpected values to be stored and leading to possible data integrity violations.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL52-CPP	Low	Unlikely	Low	P3	L3

Bibliography

[Dewhurst 2002]	Gotcha #5, "Misunderstanding References"
[ISO/IEC 14882-2014]	Subclause 8.3.2, "References"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL52-CPP+Never+qualify+a+reference+type+with+const+or+volatile>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low
reported_messages	If provided, only messages of these types are reported.	512
reported_severities	List of severities to display.	('error', 'warning', 'remark')
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

CertC++-DCL55

Avoid information leakage when passing a class object across a trust boundary.

Input: IR

Source languages: C++

Details

The C++ Standard, [class.mem], paragraph 13 [[ISO/IEC 14882-2014](#)], describes the layout of non-static data members of a non-union class, specifying the following:

Nonstatic data members of a (non-union) class with the same access control 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. 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 and virtual base classes.

Further, [class.bit], paragraph 1, in part, states the following:

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.

Thus, padding bits may be present at any location within a class object instance (including at the beginning of the object, in the case of an unnamed bit-field as the first member declared in a class). Unless initialized by zero-initialization, padding bits contain [indeterminate values](#) that may contain sensitive information.

When passing a pointer to a class object instance across a [trust boundary](#) to a different trusted domain, the programmer must ensure that the padding bits of such an object do not contain sensitive information.

Noncompliant Code Example

This noncompliant code example runs in kernel space and copies data from `arg` to user space. However, padding bits may be used within the object, for example, to ensure the proper alignment of class data members. These padding bits may contain sensitive information that may then be leaked when the data is copied to user space, regardless of how the data is copied.

```
#include <cstddef>

struct test {
    int a;
    char b;
    int c;
};

// Safely copy bytes to user space
extern int copy_to_user(void *dest, void *src, std::size_t size);

void do_stuff(void *usr_buf) {
    test arg{1, 2, 3};
    copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

Noncompliant Code Example

In this noncompliant code example, `arg` is value-initialized through direct initialization. Because `test` does not have a user-provided default constructor, the value-initialization is preceded by a zero-initialization that guarantees the padding bits are initialized to 0 before any further initialization occurs. It is akin to using `std::memset()` to initialize all of the bits in the object to 0.

```
#include <cstddef>

struct test {
    int a;
    char b;
    int c;
};

// Safely copy bytes to user space
extern int copy_to_user(void *dest, void *src, std::size_t size);

void do_stuff(void *usr_buf) {
    test arg{};

    arg.a = 1;
    arg.b = 2;
    arg.c = 3;

    copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

However, compilers are free to implement `arg.b = 2` by setting the low byte of a 32-bit register to 2, leaving the high bytes unchanged, and storing all 32 bits of the register into memory. This could leak the high-order bytes resident in the register to a user.

Compliant Solution

This compliant solution serializes the structure data before copying it to an untrusted context.

```
#include <cstddef>
#include <cstring>

struct test {
    int a;
    char b;
    int c;
};

// Safely copy bytes to user space.
extern int copy_to_user(void *dest, void *src, std::size_t size);

void do_stuff(void *usr_buf) {
    test arg{1, 2, 3};
    // May be larger than strictly needed.
    unsigned char buf[sizeof(arg)];
    std::size_t offset = 0;

    std::memcpy(buf + offset, &arg.a, sizeof(arg.a));
    offset += sizeof(arg.a);
    std::memcpy(buf + offset, &arg.b, sizeof(arg.b));
    offset += sizeof(arg.b);
    std::memcpy(buf + offset, &arg.c, sizeof(arg.c));
    offset += sizeof(arg.c);

    copy_to_user(usr_buf, buf, offset /* size of info copied */);
}
```

This code ensures that no uninitialized padding bits are copied to unprivileged users. The structure copied to user space is now a packed structure and the `copy_to_user()` function would need to unpack it to recreate the original, padded structure.

Compliant Solution [Padding Bytes]

Padding bits can be explicitly declared as fields within the structure. This solution is not portable, however, because it depends on the implementation and target memory architecture. The following solution is specific to the x86-32 architecture.

```
#include <cstddef>

struct test {
    int a;
    char b;
    char padding_1, padding_2, padding_3;
    int c;
};

test(int a, char b, int c) : a(a), b(b),
    padding_1(0), padding_2(0), padding_3(0),
    c(c) {}

// Ensure c is the next byte after the last padding byte.
static_assert(offsetof(test, c) == offsetof(test, padding_3) + 1,
              "Object contains intermediate padding");
// Ensure there is no trailing padding.
static_assert(sizeof(test) == offsetof(test, c) + sizeof(int),
              "Object contains trailing padding");

// Safely copy bytes to user space.
extern int copy_to_user(void *dest, void *src, std::size_t size);

void do_stuff(void *usr_buf) {
    test arg{1, 2, 3};
    copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

The `static_assert()` declaration accepts a constant expression and an [error message](#). The expression is evaluated at compile time and, if false, the compilation is terminated and the error message is used as the diagnostic. The explicit insertion of the padding bytes into the `struct` should ensure that no additional padding bytes are added by the compiler, and consequently both static assertions should be true. However, it is necessary to validate these

assumptions to ensure that the solution is correct for a particular implementation.

Noncompliant Code Example

In this noncompliant code example, padding bits may abound, including

- alignment padding bits after a virtual method table or virtual base class data to align a subsequent data member,
- alignment padding bits to position a subsequent data member on a properly aligned boundary,
- alignment padding bits to position data members of varying access control levels.
- bit-field padding bits when the sequential set of bit-fields does not fill an entire allocation unit,
- bit-field padding bits when two adjacent bit-fields are declared with different underlying types,
- padding bits when a bit-field is declared with a length greater than the number of bits in the underlying allocation unit, or
- padding bits to ensure a class instance will be appropriately aligned for use within an array.

This code example runs in kernel space and copies data from `arg` to user space. However, the padding bits within the object instance may contain sensitive information that will then be leaked when the data is copied to user space.

```
#include <cstddef>

class base {
public:
    virtual ~base() = default;
};

class test : public virtual base {
    alignas(32) double h;
    char i;
    unsigned j : 80;
protected:
    unsigned k;
    unsigned l : 4;
    unsigned short m : 3;
public:
    char n;
    double o;

    test(double h, char i, unsigned j, unsigned k, unsigned l, unsigned short m,
         char n, double o) :
        h(h), i(i), j(j), k(k), l(l), m(m), n(n), o(o) {}

    virtual void foo();
};

// Safely copy bytes to user space.
extern int copy_to_user(void *dest, void *src, std::size_t size);

void do_stuff(void *usr_buf) {
    test arg{0.0, 1, 2, 3, 4, 5, 6, 7.0};
    copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

Padding bits are implementation-defined, so the layout of the class object may differ between compilers or architectures. When compiled with [GCC 5.3.0](#) for x86-32, the `test` object requires 96 bytes of storage to accommodate 29 bytes of data (33 bytes including the vtable) and has the following layout.

Offset (bytes [bits])	Storage Size (bytes [bits])	Reason	Offset	Storage Size	Reason
0	1 (32)	vtable pointer	56 (448)	4 (32)	unsigned k
4 (32)	28 (224)	data member alignment padding	60 (480)	0 (4)	unsigned l : 4
32 (256)	8 (64)	double h	60 (484)	0 (3)	unsigned short m : 3
40 (320)	1 (8)	char i	60 (487)	0 (1)	unused bit-field bits
41 (328)	3 (24)	data member alignment padding	61 (488)	1 (8)	char n
44 (352)	4 (32)	unsigned j : 80	62 (496)	2 (16)	data member alignment padding
48 (384)	6 (48)	extended bit-field size padding	64 (512)	8 (64)	double o
54 (432)	2 (16)	alignment padding	72 (576)	24 (192)	class alignment padding

Compliant Solution

Due to the complexity of the data structure, this compliant solution serializes the object data before copying it to an untrusted context instead of attempting to account for all of the padding bytes manually.

```

#include <cstddef>
#include <cstring>

class base {
public:
    virtual ~base() = default;
};

class test : public virtual base {
    alignas(32) double h;
    char i;
    unsigned j : 80;
protected:
    unsigned k;
    unsigned l : 4;
    unsigned short m : 3;
public:
    char n;
    double o;

    test(double h, char i, unsigned j, unsigned k, unsigned l, unsigned short m,
          char n, double o) :
        h(h), i(i), j(j), k(k), l(l), m(m), n(n), o(o) {}

    virtual void foo();
    bool serialize(unsigned char *buffer, std::size_t &size) {
        if (size < sizeof(test)) {
            return false;
        }

        std::size_t offset = 0;
        std::memcpy(buffer + offset, &h, sizeof(h));
        offset += sizeof(h);
        std::memcpy(buffer + offset, &i, sizeof(i));
        offset += sizeof(i);
        unsigned loc_j = j; // Only sizeof(unsigned) bits are valid, so this is not narrowing.
        std::memcpy(buffer + offset, &loc_j, sizeof(loc_j));
        offset += sizeof(loc_j);
        std::memcpy(buffer + offset, &k, sizeof(k));
        offset += sizeof(k);
        unsigned char loc_l = l & 0b1111;
        std::memcpy(buffer + offset, &loc_l, sizeof(loc_l));
        offset += sizeof(loc_l);
        unsigned short loc_m = m & 0b111;
        std::memcpy(buffer + offset, &loc_m, sizeof(loc_m));
        offset += sizeof(loc_m);
        std::memcpy(buffer + offset, &n, sizeof(n));
        offset += sizeof(n);
        std::memcpy(buffer + offset, &o, sizeof(o));
        offset += sizeof(o);

        size -= offset;
        return true;
    }
};

// Safely copy bytes to user space.
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
    test arg{0.0, 1, 2, 3, 4, 5, 6, 7.0};

    // May be larger than strictly needed, will be updated by
    // calling serialize() to the size of the buffer remaining.
    std::size_t size = sizeof(arg);
    unsigned char buf[sizeof(arg)];
    if (arg.serialize(buf, size)) {
        copy_to_user(usr_buf, buf, sizeof(test) - size);
    } else {
        // Handle error
    }
}

```

This code ensures that no uninitialized padding bits are copied to unprivileged users. The structure copied to user space is now a packed structure and the `copy_to_user()` function would need to unpack it to re-create the original, padded structure.

Risk Assessment

Padding bits might inadvertently contain sensitive data such as pointers to kernel data structures or passwords. A pointer to such a structure could be passed to other functions, causing information leakage.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL55-CPP	Low	Unlikely	High	P1	L3

Related Guidelines

SEI CERT C Coding Standard	DCL39-C. Avoid information leakage when passing a structure across a trust boundary
--	---

Bibliography

[ISO/IEC 14882-2014]	Subclause 8.5, "Initializers" Subclause 9.2, "Class Members" Subclause 9.6, "Bit-fields"
--------------------------------------	--

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL55-CPP+Avoid+information+leakage+when+passing+a+class+object+across+a+trust+boundary>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	1
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high
trust_boundary_functions	When names of functions are provided in this set, only calls to these functions are reported when a pointer to some struct/class with padding is passed as argument	set()

Possible Messages

Name	Message
composite_with_padding	Composite type has padding after field '{}'.
pointer_to_padded_composite_as_argument	Pointer to composite type '{}' with padding after field '{}' is passed as argument: potential information leak.

CertC++-DCL57

Do not let exceptions escape from destructors or deallocation functions.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Under certain circumstances, terminating a destructor, operator delete, or operator delete[] by throwing an exception can trigger [undefined behavior](#).

For instance, the C++ Standard, [basic.stc.dynamic.deallocation], paragraph 3 [[ISO/IEC 14882-2014](#)], in part, states the following:

If a deallocation function terminates by throwing an exception, the behavior is undefined.

In these situations, the function must logically be declared `noexcept` because throwing an exception from the function can never have well-defined behavior. The C++ Standard, [except.spec], paragraph 15, states the following:

A deallocation function with no explicit exception-specification is treated as if it were specified with `noexcept(true)`.

As such, deallocation functions (object, array, and placement forms at either global or class scope) must not terminate by throwing an exception. Do not declare such functions to be `noexcept(false)`. However, it is acceptable to rely on the implicit `noexcept(true)` specification or declare `noexcept` explicitly on the function signature.

Object destructors are likely to be called during stack unwinding as a result of an exception being thrown. If the destructor itself throws an exception, having been called as the result of an exception being thrown, then the function `std::terminate()` is called with the default effect of calling `std::abort()` [[ISO/IEC 14882-2014](#)]. When `std::abort()` is called, no further objects are destroyed, resulting in an indeterminate program state and undefined behavior. Do not terminate a destructor by throwing an exception.

The C++ Standard, [class.dtor], paragraph 3, states [[ISO/IEC 14882-2014](#)] the following:

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.

An implicit declaration of a destructor is considered to be `noexcept(true)` according to [except.spec], paragraph 14. As such, destructors must not be declared `noexcept(false)` but may instead rely on the implicit `noexcept(true)` or declare `noexcept` explicitly.

Any `noexcept` function that terminates by throwing an exception violates [ERR55-CPP. Honor exception specifications](#).

Noncompliant Code Example

In this noncompliant code example, the class destructor does not meet the implicit `noexcept` guarantee because it may throw an exception even if it was

called as the result of an exception being thrown. Consequently, it is declared as `noexcept(false)` but still can trigger [undefined behavior](#).

```
#include <stdexcept>

class S {
    bool has_error() const;

public:
    ~S() noexcept(false) {
        // Normal processing
        if (has_error()) {
            throw std::logic_error("Something bad");
        }
    }
};
```

Noncompliant Code Example (`std::uncaught_exception()`)

Use of `std::uncaught_exception()` in the destructor solves the termination problem by avoiding the propagation of the exception if an existing exception is being processed, as demonstrated in this noncompliant code example. However, by circumventing normal destructor processing, this approach may keep the destructor from releasing important resources.

```
#include <exception>
#include <stdexcept>

class S {
    bool has_error() const;

public:
    ~S() noexcept(false) {
        // Normal processing
        if (has_error() && !std::uncaught_exception()) {
            throw std::logic_error("Something bad");
        }
    }
};
```

Noncompliant Code Example (*function-try-block*)

This noncompliant code example, as well as the following compliant solution, presumes the existence of a `Bad` class with a destructor that can throw. Although the class violates this rule, it is presumed that the class cannot be modified to comply with this rule.

```
// Assume that this class is provided by a 3rd party and it is not something
// that can be modified by the user.
class Bad {
    ~Bad() noexcept(false);
};
```

To safely use the `Bad` class, the `SomeClass` destructor attempts to handle exceptions thrown from the `Bad` destructor by absorbing them.

```
class SomeClass {
    Bad bad_member;
public:
    ~SomeClass()
    try {
        // ...
    } catch(...) {
        // Handle the exception thrown from the Bad destructor.
    }
};
```

However, the C++ Standard, [except.handle], paragraph 15 [[ISO/IEC 14882-2014](#)], in part, states the following:

The currently handled exception is rethrown if control reaches the end of a handler of the function-try-block of a constructor or destructor.

Consequently, the caught exception will inevitably escape from the `SomeClass` destructor because it is implicitly rethrown when control reaches the end of the *function-try-block* handler.

Compliant Solution

A destructor should perform the same way whether or not there is an active exception. Typically, this means that it should invoke only operations that do not throw exceptions, or it should handle all exceptions and not rethrow them (even implicitly). This compliant solution differs from the previous noncompliant code example by having an explicit `return` statement in the `SomeClass` destructor. This statement prevents control from reaching the end of the exception handler. Consequently, this handler will catch the exception thrown by `Bad::~Bad()` when `bad_member` is destroyed. It will also catch any exceptions thrown within the compound statement of the *function-try-block*, but the `SomeClass` destructor will not terminate by throwing an exception.

```
class SomeClass {
    Bad bad_member;
public:
    ~SomeClass()
    try {
        // ...
    } catch(...) {
        // Catch exceptions thrown from noncompliant destructors of
        // member objects or base class subobjects.

        // NOTE: Flowing off the end of a destructor function-try-block causes
        // the caught exception to be implicitly rethrown, but an explicit
        // return statement will prevent that from happening.
        return;
    }
};
```

Noncompliant Code Example

In this noncompliant code example, a global deallocation is declared `noexcept(false)` and throws an exception if some conditions are not properly met. However, throwing from a deallocation function results in [undefined behavior](#).

```
#include <stdexcept>

bool perform_dealloc(void *);

void operator delete(void *ptr) noexcept(false) {
    if (perform_dealloc(ptr)) {
        throw std::logic_error("Something bad");
    }
}
```

Compliant Solution

The compliant solution does not throw exceptions in the event the deallocation fails but instead fails as gracefully as possible.

```
#include <cstdlib>
#include <stdexcept>

bool perform_dealloc(void *);
void log_failure(const char *);

void operator delete(void *ptr) noexcept(true) {
    if (perform_dealloc(ptr)) {
        log_failure("Deallocation of pointer failed");
        std::exit(1); // Fail, but still call destructors
    }
}
```

Risk Assessment

Attempting to throw exceptions from destructors or deallocation functions can result in undefined behavior, leading to resource leaks or [denial-of-service attacks](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL57-CPP	Low	Likely	Medium	P6	L2

Related Guidelines

SEI CERT C++ Coding Standard	ERR55-CPP. Honor exception specifications ERR50-CPP. Do not abruptly terminate the program
MISRA C++:2008	Rule 15-5-1 (Required)

Bibliography

[Henricson 1997]	Recommendation 12.5, Do not let destructors called during stack unwinding throw exceptions
[ISO/IEC 14882-2014]	Subclause 3.4.7.2, "Deallocation Functions" Subclause 15.2, "Constructors and Destructors" Subclause 15.3, "Handling an Exception" Subclause 15.4, "Exception Specifications"
[Meyers 2005]	Item 8, "Prevent Exceptions from Leaving Destructors"
[Sutter 2000]	"Never allow exceptions from escaping destructors or from an overloaded <code>operator delete()</code> " (p. 29)

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL57-CPP.+Do+not+let+exceptions+escape+from+destructors+or+deallocation+functions>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
allowed_exceptions	Exceptions that are allowed to escape from destructors.	['bad_alloc', 'bad_cast', 'failure', 'runtime_error', 'system_error']
constructors	Whether to consider constructors.	False
destructors	Whether to consider destructors.	True
generateViolationPath	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
exceptionEscapingConstructor	Escaping exception from constructor.
exceptionEscapingDestructor	Escaping exception from destructor.

CertC++-DCL58

Do not modify the standard namespaces.

Input: IR

Source languages: C++

Details

Namespaces introduce new declarative regions for declarations, reducing the likelihood of conflicting identifiers with other declarative regions. One feature of namespaces is that they can be further extended, even within separate translation units. For instance, the following declarations are well-formed.

```
namespace MyNamespace {
int i;
}

namespace MyNamespace {
int i;
}

void f() {
    MyNamespace::i = MyNamespace::i = 12;
}
```

The standard library introduces the namespace `std` for standards-provided declarations such as `std::string`, `std::vector`, and `std::for_each`. However, it is [undefined behavior](#) to introduce new declarations in namespace `std` except under special circumstances. The C++ Standard, [namespace.std], paragraphs 1 and 2 [[ISO/IEC 14882-2014](#)], states the following:

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.

² 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.

In addition to restricting extensions to the the namespace `std`, the C++ Standard, [namespace.posix], paragraph 1, further states the following:

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.

Do not add declarations or definitions to the standard namespaces `std` or `posix`, or to a namespace contained therein, except for a template specialization that depends on a user-defined type that meets the standard library requirements for the original template.

The Library Working Group, responsible for the wording of the Standard Library section of the C++ Standard, has an unresolved [issue](#) on the definition of *user-defined type*. Although the Library Working Group has no official stance on the definition [[INCITS 2014](#)], we define it to be any `class`, `struct`, `union`,

or `enum` that is not defined within namespace `std` or a namespace contained within namespace `std`. Effectively, it is a user-provided type instead of a standard library-provided type.

Noncompliant Code Example

In this noncompliant code example, the declaration of `x` is added to the namespace `std`, resulting in [undefined behavior](#).

```
namespace std {  
int x;  
}
```

Compliant Solution

This compliant solution assumes the intention of the programmer was to place the declaration of `x` into a namespace to prevent collisions with other global identifiers. Instead of placing the declaration into the namespace `std`, the declaration is placed into a namespace without a reserved name.

```
namespace nonstd {  
int x;  
}
```

Noncompliant Code Example

In this noncompliant code example, a template specialization of `std::plus` is added to the namespace `std` in an attempt to allow `std::plus` to concatenate a `std::string` and `MyString` object. However, because the template specialization is of a standard library-provided type (`std::string`), this code results in undefined behavior.

```
#include <functional>  
#include <iostream>  
#include <string>  
  
class MyString {  
    std::string data;  
  
public:  
    MyString(const std::string &data) : data(data) {}  
  
    const std::string &get_data() const { return data; }  
};  
  
namespace std {  
template <>  
struct plus<string> : binary_function<string, MyString, string> {  
    string operator()(const string &lhs, const MyString &rhs) const {  
        return lhs + rhs.get_data();  
    }  
};  
}  
  
void f() {  
    std::string s1("My String");  
    MyString s2(" + Your String");  
    std::plus<std::string> p;  
  
    std::cout << p(s1, s2) << std::endl;  
}
```

Compliant Solution

The interface for `std::plus` requires that both arguments to the function call operator and the return type are of the same type. Because the attempted specialization in the noncompliant code example results in [undefined behavior](#), this compliant solution defines a new `std::binary_function` derivative that can add a `std::string` to a `MyString` object without requiring modification of the namespace `std`.

```
#include <functional>  
#include <iostream>  
#include <string>  
  
class MyString {  
    std::string data;  
  
public:  
    MyString(const std::string &data) : data(data) {}  
  
    const std::string &get_data() const { return data; }  
};  
  
struct my_plus : std::binary_function<std::string, MyString, std::string> {  
    std::string operator()(const std::string &lhs, const MyString &rhs) const {  
        return lhs + rhs.get_data();  
    }  
};  
  
void f() {  
    std::string s1("My String");  
    MyString s2(" + Your String");  
    my_plus p;  
  
    std::cout << p(s1, s2) << std::endl;  
}
```

Compliant Solution

In this compliant solution, a specialization of `std::plus` is added to the `std` namespace, but the specialization depends on a user-defined type and meets the

Standard Template Library requirements for the original template, so it complies with this rule. However, because `MyString` can be constructed from `std::string`, this compliant solution involves invoking a converting constructor whereas the previous compliant solution does not.

```
#include <functional>
#include <iostream>
#include <string>

class MyString {
    std::string data;

public:
    MyString(const std::string &data) : data(data) {}

    const std::string &get_data() const { return data; }
};

namespace std {
template <>
struct plus<MyString> {
    MyString operator()(const MyString &lhs, const MyString &rhs) const {
        return lhs.get_data() + rhs.get_data();
    }
};

void f() {
    std::string s1("My String");
    MyString s2(" + Your String");
    std::plus<MyString> p;

    std::cout << p(s1, s2).get_data() << std::endl;
}
```

Risk Assessment

Altering the standard namespace can cause [undefined behavior](#) in the C++ standard library.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL58-CPP	High	Unlikely	Medium	P6	L2

Related Guidelines

SEI CERT C++ Coding Standard	DCL51-CPP. Do not declare or define a reserved identifier
--	---

Bibliography

[INCITS 2014]	Issue 2139, "What Is a <i>User-Defined Type</i> ?"
[ISO/IEC 14882-2014]	Subclause 17.6.4.2.1, "Namespace <code>std</code> " Subclause 17.6.4.2.2, "Namespace <code>posix</code> "

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL58-CPP.+Do+not+modify+the+standard+namespaces>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
std_specialization_blacklist		('std::integral_constant', 'std::is_void', 'std::is_null_pointer', 'std::is_integral', 'std::is_floating_point', 'std::is_array', 'std::is_enum', 'std::is_union', 'std::is_class', 'std::is_function', 'std::is_pointer', 'std::is_lvalue_reference', 'std::is_rvalue_reference', 'std::is_member_object_pointer', 'std::is_member_function_pointer', 'std::is_fundamental', 'std::is_arithmetic', 'std::is_scalar', 'std::is_object', 'std::is_compound', 'std::is_reference', 'std::is_member_pointer', 'std::is_const', 'std::is_volatile', 'std::is_trivial', 'std::is_trivially_copyable', 'std::is_standard_layout', 'std::is_pod', 'std::is_literal_type', 'std::has_unique_object_representations', 'std::is_empty', 'std::is_polymorphic', 'std::is_abstract', 'std::is_final', 'std::is_aggregate', 'std::is_signed', 'std::is_unsigned', 'std::is_constructible', 'std::is_trivially_constructible', 'std::is_nothrow_constructible', 'std::is_default_constructible', 'std::is_trivially_default_constructible', 'std::is_nothrow_default_constructible', 'std::is_copy_constructible', 'std::is_trivially_copy_constructible', 'std::is_nothrow_copy_constructible', 'std::is_nothrow_copy_constructible', 'std::is_move_constructible', 'std::is_trivially_move_constructible', 'std::is_nothrow_move_constructible', 'std::is_assignable', 'std::is_trivially_assignable', 'std::is_nothrowAssignable', 'std::is_copyAssignable', 'std::is_trivially_copyAssignable', 'std::is_nothrow_copyAssignable', 'std::is_moveAssignable', 'std::is_trivially_moveAssignable', 'std::is_nothrow_moveAssignable', 'std::is_destructible', 'std::is_trivially_destructible', 'std::is_nothrow_destructible', 'std::has_virtual_destructor', 'std::is_swappable', 'std::is_nothrow_swappable', 'std::is_nothrow_swappable_with', 'std::is_nothrow_swappable', 'std::alignment_of', 'std::rank', 'std::extent', 'std::is_same', 'std::is_base_of', 'std::is_convertible', 'std::is_nothrow_convertible', 'std::is_invocable', 'std::is_invocable_r', 'std::is_nothrow_invocable', 'std::is_nothrow_invocable_r', 'std::remove_cv', 'std::remove_const', 'std::remove_volatile', 'std::add_cv', 'std::add_const', 'std::add_volatile', 'std::remove_reference', 'std::add_lvalue_reference', 'std::add_rvalue_reference', 'std::remove_pointer', 'std::add_pointer', 'std::make_signed', 'std::make_unsigned', 'std::remove_extent', 'std::remove_all_extents', 'std::aligned_storage', 'std::aligned_union', 'std::decay', 'std::remove_cvref', 'std::enable_if', 'std::conditional', 'std::common_type', 'std::underlying_type', 'std::result_of', 'std::invoke_result', 'std::void_t', 'std::conjunction', 'std::disjunction', 'std::negation', 'std::endian', 'std::unary_function', 'std::binary_function')
std_specialization_whitelist		('std::common_type',)

Possible Messages

Name	Message
std_extension	Invalid addition to std namespace
std_specialization	Invalid std template specialization

CertC++-DCL59

Do not define an unnamed namespace in a header file.

Input: IR

Source languages: C++

Details

Unnamed namespaces are used to define a namespace that is unique to the translation unit, where the names contained within have internal linkage by default. The C++ Standard, [namespace.unnamed], paragraph 1 [[ISO/IEC 14882-2014](#)], states the following:

An *unnamed-namespace-definition* behaves as if it were replaced by:

```
inline 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.

Production-quality C++ code frequently uses *header files* as a means to share code between translation units. A header file is any file that is inserted into a translation unit through an `#include` directive. Do not define an unnamed namespace in a header file. When an unnamed namespace is defined in a header file, it can lead to surprising results. Due to default internal linkage, each translation unit will define its own unique instance of members of the unnamed namespace that are [ODR-used](#) within that translation unit. This can cause unexpected results, bloat the resulting executable, or inadvertently trigger [undefined behavior](#) due to one-definition rule (ODR) violations.

Noncompliant Code Example

In this noncompliant code example, the variable `v` is defined in an unnamed namespace within a header file and is accessed from two separate translation units. Each translation unit prints the current value of `v` and then assigns a new value into it. However, because `v` is defined within an unnamed namespace, each translation unit operates on its own instance of `v`, resulting in unexpected output.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE

namespace {
int v;
}

#endif // A_HEADER_FILE

// a.cpp
#include "a.h"
#include <iostream>

void f() {
    std::cout << "f(): " << v << std::endl;
    v = 42;
    // ...
}

// b.cpp
#include "a.h"
#include <iostream>

void g() {
    std::cout << "g(): " << v << std::endl;
    v = 100;
}

int main() {
    extern void f();
    f(); // Prints v, sets it to 42
    g(); // Prints v, sets it to 100
    f();
    g();
}
```

When executed, this program prints the following.

```
f(): 0
g(): 0
f(): 42
g(): 100
```

Compliant Solution

In this compliant solution, `v` is defined in only one translation unit but is externally visible to all translation units, resulting in the expected behavior.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE

extern int v;

#endif // A_HEADER_FILE

// a.cpp
#include "a.h"
#include <iostream>

int v; // Definition of global variable v

void f() {
    std::cout << "f(): " << v << std::endl;
    v = 42;
    // ...
}
```

```
// b.cpp
#include "a.h"
#include <iostream>

void g() {
    std::cout << "g(): " << v << std::endl;
    v = 100;
}

int main() {
    extern void f();
    f(); // Prints v, sets it to 42
    g(); // Prints v, sets it to 100
    f(); // Prints v, sets it back to 42
    g(); // Prints v, sets it back to 100
}
```

When executed, this program prints the following.

```
f(): 0
g(): 42
f(): 100
g(): 42
```

Noncompliant Code Example

In this noncompliant code example, the variable `v` is defined in an unnamed namespace within a header file, and an inline function, `get_v()`, is defined, which accesses that variable. ODR-using the inline function from multiple translation units (as shown in the implementation of `f()` and `g()`) violates the [one-definition rule](#) because the definition of `get_v()` is not identical in all translation units due to referencing a unique `v` in each translation unit.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE

namespace {
int v;
}

inline int get_v() { return v; }

#endif // A_HEADER_FILE

// a.cpp
#include "a.h"

void f() {
    int i = get_v();
    // ...
}

// b.cpp
#include "a.h"

void g() {
    int i = get_v();
    // ...
}
```

See [DCL60-CPP. Obey the one-definition rule](#) for more information on violations of the one-definition rule.

Compliant Solution

In this compliant solution, `v` is defined in only one translation unit but is externally visible to all translation units and can be accessed from the inline `get_v()` function.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE

extern int v;

inline int get_v() {
    return v;
}

#endif // A_HEADER_FILE

// a.cpp
#include "a.h"

// Externally used by get_v();
int v;

void f() {
    int i = get_v();
    // ...
}

// b.cpp
#include "a.h"

void g() {
    int i = get_v();
    // ...
}
```

Noncompliant Code Example

In this noncompliant code example, the function `f()` is defined within a header file. However, including the header file in multiple translation units causes a violation of the one-definition rule that usually results in an error diagnostic generated at link time due to multiple definitions of a function with the same name.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE

void f() { /* ... */ }

#endif // A_HEADER_FILE

// a.cpp
#include "a.h"
// ...

// b.cpp
#include "a.h"
// ...
```

Noncompliant Code Example

This noncompliant code example attempts to resolve the link-time errors by defining `f()` within an unnamed namespace. However, it produces multiple, unique definitions of `f()` in the resulting executable. If `a.h` is included from many translation units, it can lead to increased link times, a larger executable file, and reduced performance.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE

namespace {
void f() { /* ... */ }
}

#endif // A_HEADER_FILE

// a.cpp
#include "a.h"
// ...

// b.cpp
#include "a.h"
// ...
```

Compliant Solution

In this compliant solution, `f()` is not defined with an unnamed namespace and is instead defined as an inline function. Inline functions are required to be defined identically in all the translation units in which they are used, which allows an [implementation](#) to generate only a single instance of the function at runtime in the event the body of the function does not get generated for each call site.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE

inline void f() { /* ... */ }

#endif // A_HEADER_FILE

// a.cpp
#include "a.h"
// ...

// b.cpp
#include "a.h"
// ...
```

Risk Assessment

Defining an unnamed namespace within a header file can cause data integrity violations and performance problems but is unlikely to go unnoticed with sufficient testing. One-definition rule violations result in [undefined behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL59-CPP	Medium	Unlikely	Medium	P4	L3

Related Guidelines

[SEI CERT C++ Coding Standard](#) | [DCL60-CPP. Obey the one-definition rule](#)

Bibliography

[ISO/IEC 14882-2014]	Subclause 3.2, "One Definition Rule" Subclause 7.1.2, "Function Specifiers" Subclause 7.3.1, "Namespace Definition"
--------------------------------------	---

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL59-CPP.+Do+not+define+an+unnamed+namespace+in+a+header+file>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
unnamed_namespace_in_header	Use of unnamed namespace in header file.

CertC++-DCL60

Obey the one-definition rule.

Input: IR

Source languages: C++

Details

Nontrivial C++ programs are generally divided into multiple translation units that are later linked together to form an executable. To support such a model, C++ restricts named object definitions to ensure that linking will behave deterministically by requiring a single definition for an object across all translation units. This model is called the *one-definition rule* (ODR), which is defined by the C++ Standard, [basic.def.odr], in paragraph 4 [[ISO/IEC 14882-2014](#)]:

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. An inline function shall be defined in every translation unit in which it is odr-used.

The most common approach to multitranslation unit compilation involves declarations residing in a header file that is subsequently made available to a source file via `#include`. These declarations are often also definitions, such as class and function template definitions. This approach is allowed by an exception defined in paragraph 6, which, in part, states the following:

There can be more than one definition of a class type, enumeration type, inline function with external linkage, class template, non-static function template, static data member of a class template, member function of a class template, or template specialization for which some template parameters are not specified 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....

If the definitions of `D` satisfy all these requirements, then the program shall behave as if there were a single definition of `D`. If the definitions of `D` do not satisfy these requirements, then the behavior is undefined.

The requirements specified by paragraph 6 essentially state that that two definitions must be identical (not simply equivalent). Consequently, a definition introduced in two separate translation units by an `#include` directive generally will not violate the ODR because the definitions are identical in both translation units.

However, it is possible to violate the ODR of a definition introduced via `#include` using block language linkage specifications, vendor-specific language extensions, and so on. A more likely scenario for ODR violations is that accidental definitions of differing objects will exist in different translation units.

Do not violate the one-definition rule; violations result in [undefined behavior](#).

Noncompliant Code Example

In this noncompliant code example, two different translation units define a class of the same name with differing definitions. Although the two definitions are functionally equivalent (they both define a class named `S` with a single, public, nonstatic data member `int a`), they are not defined using the same sequence of tokens. This code example violates the ODR and results in [undefined behavior](#).

```
// a.cpp
struct S {
    int a;
};

// b.cpp
class S {
```

```
public:  
    int a;  
};
```

Compliant Solution

The correct mitigation depends on programmer intent. If the programmer intends for the same class definition to be visible in both translation units because of common usage, the solution is to use a header file to introduce the object into both translation units, as shown in this compliant solution.

```
// S.h  
struct S {  
    int a;  
};  
  
// a.cpp  
#include "S.h"  
  
// b.cpp  
#include "S.h"
```

Compliant Solution

If the ODR violation was a result of accidental name collision, the best mitigation solution is to ensure that both class definitions are unique, as in this compliant solution.

```
// a.cpp  
namespace {  
    struct S {  
        int a;  
    };  
}  
  
// b.cpp  
namespace {  
    class S {  
        public:  
            int a;  
    };  
}
```

Alternatively, the classes could be given distinct names in each translation unit to avoid violating the ODR.

Noncompliant Code Example [Microsoft Visual Studio]

In this noncompliant code example, a class definition is introduced into two translation units using `#include`. However, one of the translation units uses an [implementation-defined](#) `#pragma` that is supported by Microsoft Visual Studio to specify structure field alignment requirements. Consequently, the two class definitions may have differing layouts in each translation unit, which is a violation of the ODR.

```
// s.h  
struct S {  
    char c;  
    int a;  
};  
  
void init_s(S &s);  
  
// s.cpp  
#include "s.h"  
  
void init_s(S &s) {  
    s.c = 'a';  
    s.a = 12;  
}  
  
// a.cpp  
#pragma pack(push, 1)  
#include "s.h"  
#pragma pack(pop)  
  
void f() {  
    S s;  
    init_s(s);  
}
```

Implementation Details

It is possible for the preceding noncompliant code example to result in `a.cpp` allocating space for an object with a different size than expected by `init_s()` in `s.cpp`. When translating `s.cpp`, the layout of the structure may include padding bytes between the `c` and `a` data members. When translating `a.cpp`, the layout of the structure may remove those padding bytes as a result of the `#pragma pack` directive, so the object passed to `init_s()` may be smaller than expected. Consequently, when `init_s()` initializes the data members of `s`, it may result in a buffer overrun.

For more information on the behavior of `#pragma pack`, see the vendor documentation for your [implementation](#), such as [Microsoft Visual Studio](#) or [GCC](#).

Compliant Solution

In this compliant solution, the implementation-defined structure member-alignment directive is removed, ensuring that all definitions of `s` comply with the ODR.

```
// s.h  
struct S {  
    char c;
```

```

    int a;
};

void init_s(S &s);

// s.cpp
#include "s.h"

void init_s(S &s) {
    s.c = 'a';
    s.a = 12;
}

// a.cpp
#include "s.h"

void f() {
    S s;
    init_s(s);
}

```

Noncompliant Code Example

In this noncompliant code example, the constant object `n` has internal linkage but is [odr-used](#) within `f()`, which has external linkage. Because `f()` is declared as an inline function, the definition of `f()` must be identical in all translation units. However, each translation unit has a unique instance of `n`, resulting in a violation of the ODR.

```

const int n = 42;

int g(const int &lhs, const int &rhs);

inline int f(int k) {
    return g(k, n);
}

```

Compliant Solution

A compliant solution must change one of three factors: (1) it must not odr-use `n` within `f()`, (2) it must declare `n` such that it has external linkage, or (3) it must not use an inline definition of `f()`.

If circumstances allow modification of the signature of `g()` to accept parameters by value instead of by reference, then `n` will not be odr-used within `f()` because `n` would then qualify as a constant expression. This solution is compliant but it is not ideal. It may not be possible (or desirable) to modify the signature of `g()`, such as if `g()` represented `std::max()` from `<algorithm>`. Also, because of the differing linkage used by `n` and `f()`, accidental violations of the ODR are still likely if the definition of `f()` is modified to odr-use `n`.

```

const int n = 42;

int g(int lhs, int rhs);

inline int f(int k) {
    return g(k, n);
}

```

Compliant Solution

In this compliant solution, the constant object `n` is replaced with an enumerator of the same name. Named enumerations defined at namespace scope have the same linkage as the namespaces they are contained in. The global namespace has external linkage, so the definition of the named enumeration and its contained enumerators also have external linkage. Although less aesthetically pleasing, this compliant solution does not suffer from the same maintenance burdens of the previous code because `n` and `f()` have the same linkage.

```

enum Constants {
    N = 42
};

int g(const int &lhs, const int &rhs);

inline int f(int k) {
    return g(k, N);
}

```

Risk Assessment

Violating the ODR causes [undefined behavior](#), which can result in exploits as well as [denial-of-service attacks](#). As shown in "Support for Whole-Program Analysis and the Verification of the One-Definition Rule in C++" [Quinlan 06], failing to enforce the ODR enables a virtual function pointer attack known as the [VPTP exploit](#). In this exploit, an object's virtual function table is corrupted so that calling a virtual function on the object results in malicious code being executed. See the paper by Quinlan and colleagues for more details. However, note that to introduce the malicious class, the attacker must have access to the system building the code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL60-CPP	High	Unlikely	High	P3	L3

Bibliography

[ISO/IEC 14882-2014]	Subclause 3.2, "One Definition Rule"
[Quinlan 2006]	

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL60-CPP.+Obey+the+one-definition+rule>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
class_struct_difference	{}
different_enumerators	{}
different_field_types	{}
different_fields	{}
general_odrViolation	{}

CertC++-EXP34

Do not dereference null pointers.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Dereferencing a null pointer is [undefined behavior](#).

On many platforms, dereferencing a null pointer results in [abnormal program termination](#), but this is not required by the standard. See "[Clever Attack Exploits Fully-Patched Linux Kernel](#)" [Goodin 2009] for an example of a code execution [exploit](#) that resulted from a null pointer dereference.

Noncompliant Code Example

This noncompliant code example is derived from a real-world example taken from a vulnerable version of the `libpng` library as deployed on a popular ARM-based cell phone [Jack 2007]. The `libpng` library allows applications to read, create, and manipulate PNG (Portable Network Graphics) raster image files. The `libpng` library implements its own wrapper to `malloc()` that returns a null pointer on error or on being passed a 0-byte-length argument.

This code also violates [ERR33-C. Detect and handle standard library errors](#).

```
#include <png.h> /* From libpng */
#include <string.h>

void func(png_structp png_ptr, int length, const void *user_data) {
    png_charp chunkdata;
    chunkdata = (png_charp)png_malloc(png_ptr, length + 1);
    /* ... */
    memcpy(chunkdata, user_data, length);
    /* ... */
}
```

If `length` has the value -1, the addition yields 0, and `png_malloc()` subsequently returns a null pointer, which is assigned to `chunkdata`. The `chunkdata` pointer is later used as a destination argument in a call to `memcpy()`, resulting in user-defined data overwriting memory starting at address 0. In the case of the ARM and XScale architectures, the `0x0` address is mapped in memory and serves as the exception vector table; consequently, dereferencing `0x0` did not cause an [abnormal program termination](#).

Compliant Solution

This compliant solution ensures that the pointer returned by `png_malloc()` is not null. It also uses the unsigned type `size_t` to pass the `length` parameter, ensuring that negative values are not passed to `func()`.

```
#include <png.h> /* From libpng */
#include <string.h>

void func(png_structp png_ptr, size_t length, const void *user_data) {
    png_charp chunkdata;
    if (length == SIZE_MAX) {
        /* Handle error */
    }
    chunkdata = (png_charp)png_malloc(png_ptr, length + 1);
    if (NULL == chunkdata) {
        /* Handle error */
    }
    /* ... */
    memcpy(chunkdata, user_data, length);
    /* ... */
}
```

Noncompliant Code Example

In this noncompliant code example, `input_str` is copied into dynamically allocated memory referenced by `c_str`. If `malloc()` fails, it returns a null pointer that is assigned to `c_str`. When `c_str` is dereferenced in `memcpy()`, the program exhibits [undefined behavior](#). Additionally, if `input_str` is a null pointer, the call to `strlen()` dereferences a null pointer, also resulting in undefined behavior. This code also violates [ERR33-C. Detect and handle standard library errors](#).

```
#include <string.h>
#include <stdlib.h>

void f(const char *input_str) {
    size_t size = strlen(input_str) + 1;
    char *c_str = (char *)malloc(size);
    memcpy(c_str, input_str, size);
    /* ... */
    free(c_str);
    c_str = NULL;
    /* ... */
}
```

Compliant Solution

This compliant solution ensures that both `input_str` and the pointer returned by `malloc()` are not null:

```
#include <string.h>
#include <stdlib.h>

void f(const char *input_str) {
    size_t size;
    char *c_str;

    if (NULL == input_str) {
        /* Handle error */
    }

    size = strlen(input_str) + 1;
    c_str = (char *)malloc(size);
    if (NULL == c_str) {
        /* Handle error */
    }
    memcpy(c_str, input_str, size);
    /* ... */
    free(c_str);
    c_str = NULL;
    /* ... */
}
```

Noncompliant Code Example

This noncompliant code example is from a version of `drivers/net/tun.c` and affects Linux kernel 2.6.30 [[Goodin 2009](#)]:

```
static unsigned int tun_chr_poll(struct file *file, poll_table *wait) {
    struct tun_file *tfile = file->private_data;
    struct tun_struct *tun = __tun_get(tfile);
    struct sock *sk = tun->sk;
    unsigned int mask = 0;

    if (!tun)
        return POLLERR;

    DBG(KERN_INFO "%s: tun_chr_poll\n", tun->dev->name);

    poll_wait(file, &tun->socket.wait, wait);

    if (!skb_queue_empty(&tun->readq))
        mask |= POLLIN | POLLRDNORM;

    if (sock_writeable(sk) ||
        (!test_and_set_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags) &&
         sock_writeable(sk)))
        mask |= POLLOUT | POLLWRNORM;

    if (tun->dev->reg_state != NETREG_REGISTERED)
        mask = POLLERR;

    tun_put(tun);
    return mask;
}
```

The `sk` pointer is initialized to `tun->sk` before checking if `tun` is a null pointer. Because null pointer dereferencing is [undefined behavior](#), the compiler (GCC in this case) can optimize away the `if (!tun)` check because it is performed after `tun->sk` is accessed, implying that `tun` is non-null. As a result, this noncompliant code example is vulnerable to a null pointer dereference exploit, because null pointer dereferencing can be permitted on several platforms, for example, by using `mmap(2)` with the `MAP_FIXED` flag on Linux and Mac OS X, or by using the `shmat()` POSIX function with the `SHM_RND` flag [[Liu 2009](#)].

Compliant Solution

This compliant solution eliminates the null pointer deference by initializing `sk` to `tun->sk` following the null pointer check:

```
static unsigned int tun_chr_poll(struct file *file, poll_table *wait) {
    struct tun_file *tfile = file->private_data;
    struct tun_struct *tun = __tun_get(tfile);
    struct sock *sk;
    unsigned int mask = 0;

    if (!tun)
        return POLLERR;

    sk = tun->sk;

    /* The remaining code is omitted because it is unchanged... */
}
```

Risk Assessment

Dereferencing a null pointer is [undefined behavior](#), typically [abnormal program termination](#). In some situations, however, dereferencing a null pointer can lead to the execution of arbitrary code [[Jack 2007](#), [van Sprundel 2006](#)]. The indicated severity is for this more severe case; on platforms where it is not possible to exploit a null pointer dereference to execute arbitrary code, the actual severity is low.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP34-C	High	Likely	Medium	P18	L1

Related Guidelines

CERT Oracle Secure Coding Standard for Java	EXP01-J. Do not use a null in a case where an object is required
ISO/IEC TR 24772:2013	Pointer Casting and Pointer Type Changes [HFC] Null Pointer Dereference [XYH]
ISO/IEC TS 17961	Dereferencing an out-of-domain pointer [nullref]
MITRE CWE	CWE-476 , NULL Pointer Dereference

Bibliography

[Goodin 2009]	
[Jack 2007]	
[Liu 2009]	
[van Sprundel 2006]	
[Viega 2005]	Section 5.2.18, "Null-Pointer Dereference"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/PAw>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
null_deref	Pointer is NULL at dereference
possible_null_deref	Pointer may be NULL at dereference

CertC++-EXP35

Do not modify objects with temporary lifetime.

Input: IR

Source languages: C++

Details

The C11 Standard [[ISO/IEC 9899:2011](#)] introduced a new term: *temporary lifetime*. Modifying an object with temporary lifetime is [undefined behavior](#). According to subclause 6.2.4, paragraph 8

A non-lvalue expression with structure or union type, where the structure or union contains a member with array type (including, recursively, members of all contained structures and unions) refers to an object with automatic storage duration and *temporary* lifetime. Its lifetime begins when the expression is evaluated and its initial value is the value of the expression. Its lifetime ends when the evaluation of the containing full expression or full declarator ends. Any attempt to modify an object with temporary lifetime results in undefined behavior.

This definition differs from the C99 Standard (which defines modifying the result of a function call or accessing it after the next sequence point as undefined behavior) because a temporary object's lifetime ends when the evaluation containing the full expression or full declarator ends, so the result of a function call can be accessed. This extension to the lifetime of a temporary also removes a quiet change to C90 and improves compatibility with C++.

C functions may not return arrays; however, functions can return a pointer to an array or a `struct` or `union` that contains arrays. Consequently, if a function call returns by value a `struct` or `union` containing an array, do not modify those arrays within the expression containing the function call. Do not access an array returned by a function after the next sequence point or after the evaluation of the containing full expression or full declarator ends.

Noncompliant Code Example (C99)

This noncompliant code example [conforms](#) to the C11 Standard; however, it fails to conform to C99. If compiled with a C99-conforming implementation, this code has [undefined behavior](#) because the sequence point preceding the call to `printf()` comes between the call and the access by `printf()` of the string in the returned object.

```
#include <stdio.h>

struct X { char a[8]; };

struct X salutation(void) {
    struct X result = { "Hello" };
    return result;
}

struct X addressee(void) {
    struct X result = { "world" };
    return result;
}

int main(void) {
    printf("%s, %s!\n", salutation().a, addressee().a);
    return 0;
}
```

Compliant Solution

This compliant solution stores the structures returned by the call to `addressee()` before calling the `printf()` function. Consequently, this program conforms to both C99 and C11.

```
#include <stdio.h>

struct X { char a[8]; };

struct X salutation(void) {
    struct X result = { "Hello" };
    return result;
}

struct X addressee(void) {
    struct X result = { "world" };
    return result;
}

int main(void) {
    struct X my_salutation = salutation();
    struct X my_addressee = addressee();

    printf("%s, %s!\n", my_salutation.a, my_addressee.a);
    return 0;
}
```

This noncompliant code example attempts to retrieve an array and increment the array's first value. The array is part of a `struct` that is returned by a function call. Consequently, the array has temporary lifetime, and modifying the array is [undefined behavior](#).

```
#include <stdio.h>

struct X { int a[6]; };

struct X addressee(void) {
    struct X result = { { 1, 2, 3, 4, 5, 6 } };
    return result;
}

int main(void) {
    printf("%x", ++(addressee().a[0]));
    return 0;
}
```

Compliant Solution

This compliant solution stores the structure returned by the call to `addressee()` as `my_x` before calling the `printf()` function. When the array is modified, its lifetime is no longer temporary but matches the lifetime of the block in `main()`.

```
#include <stdio.h>

struct X { int a[6]; };

struct X addressee(void) {
    struct X result = { { 1, 2, 3, 4, 5, 6 } };
    return result;
}

int main(void) {
    struct X my_x = addressee();
    printf("%x", ++(my_x.a[0]));
    return 0;
}
```

Risk Assessment

Attempting to modify an array or access it after its lifetime expires may result in erroneous program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP35-C	Low	Probable	Medium	P4	L3

Related Guidelines

ISO/IEC TR 24772:2013	Dangling References to Stack Frames [DCM] Side-effects and Order of Evaluation [SAM]
---------------------------------------	---

Bibliography

ISO/IEC 9899:2011	6.2.4, "Storage Durations of Objects"
-----------------------------------	---------------------------------------

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/pYET>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
temp_array_decay	Temporary array converted to pointer
temp_mutation	Mutation of temporary array

CertC++-EXP36

Do not cast pointers into more strictly aligned pointer types.

Input: IR

Source languages: C++

Details

Do not convert a pointer value to a pointer type that is more strictly aligned than the referenced type. Different alignments are possible for different types of objects. If the type-checking system is overridden by an explicit cast or the pointer is converted to a void pointer (`void *`) and then to a different type, the alignment of an object may be changed.

The C Standard, 6.3.2.3, paragraph 7 [[ISO/IEC 9899:2011](#)], states

A pointer to an object or incomplete type may be converted to a pointer to a different object or incomplete type. If the resulting pointer is not correctly aligned for the referenced type, the behavior is undefined.

See [undefined behavior 25](#).

If the misaligned pointer is dereferenced, the program may [terminate abnormally](#). On some architectures, the cast alone may cause a loss of information even if the value is not dereferenced if the types involved have differing alignment requirements.

Noncompliant Code Example

In this noncompliant example, the `char` pointer `&c` is converted to the more strictly aligned `int` pointer `ip`. On some [implementations](#), `cp` will not match `&c`. As a result, if a pointer to one object type is converted to a pointer to a different object type, the second object type must not require stricter alignment than the first.

```
#include <assert.h>

void func(void) {
    char c = 'x';
    int *ip = (int *)&c; /* This can lose information */
    char *cp = (char *)ip;

    /* Will fail on some conforming implementations */
    assert(cp == &c);
}
```

Compliant Solution (Intermediate Object)

In this compliant solution, the `char` value is stored into an object of type `int` so that the pointer's value will be properly aligned:

```
#include <assert.h>

void func(void) {
    char c = 'x';
    int i = c;
    int *ip = &i;

    assert(ip == &i);
}
```

Noncompliant Code Example

The C Standard allows any object pointer to be cast to and from `void *`. As a result, it is possible to silently convert from one pointer type to another without the compiler diagnosing the problem by storing or casting a pointer to `void *` and then storing or casting it to the final type. In this noncompliant code example, `loop_function()` is passed the `char` pointer `loop_ptr` but returns an object of type `int` pointer:

```
int *loop_function(void *v_pointer) {
    /* ... */
    return v_pointer;
}

void func(char *loop_ptr) {
    int *int_ptr = loop_function(loop_ptr);

    /* ... */
}
```

This example compiles without warning using GCC 4.8 on Ubuntu Linux 14.04. However, `v_pointer` can be more strictly aligned than an object of type `int *`.

Compliant Solution

Because the input parameter directly influences the return value, and `loop_function()` returns an object of type `int *`, the formal parameter `v_pointer` is

redeclared to accept only an object of type `int *`:

```
int *loop_function(int *v_pointer) {
    /* ... */
    return v_pointer;
}

void func(int *loop_ptr) {
    int *int_ptr = loop_function(loop_ptr);

    /* ... */
}
```

Noncompliant Code Example

Some architectures require that pointers are correctly aligned when accessing objects larger than a byte. However, it is common in system code that unaligned data (for example, the network stacks) must be copied to a properly aligned memory location, such as in this noncompliant code example:

```
#include <string.h>

struct foo_header {
    int len;
    /* ... */
};

void func(char *data, size_t offset) {
    struct foo_header *tmp;
    struct foo_header header;

    tmp = (struct foo_header *) (data + offset);
    memcpy(&header, tmp, sizeof(header));

    /* ... */
}
```

Assigning an unaligned value to a pointer that references a type that needs to be aligned is [undefined behavior](#). An [implementation](#) may notice, for example, that `tmp` and `header` must be aligned and use an inline `memcpy()` that uses instructions that assume aligned data.

Compliant Solution

This compliant solution avoids the use of the `foo_header` pointer:

```
#include <string.h>

struct foo_header {
    int len;
    /* ... */
};

void func(char *data, size_t offset) {
    struct foo_header header;
    memcpy(&header, data + offset, sizeof(header));

    /* ... */
}
```

Exceptions

EXP36-C-EX1: Some hardware architectures have relaxed requirements with regard to pointer alignment. Using a pointer that is not properly aligned is correctly handled by the architecture, although there might be a performance penalty. On such an architecture, improper pointer alignment is permitted but remains an efficiency problem.

EXP36-C-EX2: If a pointer is known to be correctly aligned to the target type, then a cast to that type is permitted. There are several cases where a pointer is known to be correctly aligned to the target type. The pointer could point to an object declared with a suitable alignment specifier. It could point to an object returned by `aligned_alloc()`, `calloc()`, `malloc()`, or `realloc()`, as per the C standard, section 7.22.3, paragraph 1 [[ISO/IEC 9899:2011](#)].

This compliant solution uses the alignment specifier, which is new to C11, to declare the `char` object `c` with the same alignment as that of an object of type `int`. As a result, the two pointers reference equally aligned pointer types:

```
#include <stdalign.h>
#include <assert.h>

void func(void) {
    /* Align c to the alignment of an int */
    alignas(int) char c = 'x';
    int *ip = (int *)&c;
    char *cp = (char *)ip;
    /* Both cp and &c point to equally aligned objects */
    assert(cp == &c);
}
```

Risk Assessment

Accessing a pointer or an object that is not properly aligned can cause a program to crash or give erroneous information, or it can cause slow pointer accesses (if the architecture allows misaligned accesses).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP36-C	Low	Probable	Medium	P4	L3

Related Guidelines

SEI CERT C++ Coding Standard	VOID EXP56-CPP. Do not cast pointers into more strictly aligned pointer types
ISO/IEC TR 24772:2013	Pointer Casting and Pointer Type Changes [HFC]
ISO/IEC TS 17961	Converting pointer values to more strictly aligned pointer types [alignconv]
MISRA C:2012	Rule 11.1 (required) Rule 11.2 (required) Rule 11.5 (advisory) Rule 11.7 (required)

Bibliography

[Bryant 2003]	
[ISO/IEC 9899:2011]	6.3.2.3, "Pointers"
[Walfridsson 2003]	Aliasing, Pointer Casts and GCC 3.3

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/tgAV>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
allow_cast_from_char_ptr	Allow all casts from char*.	False
allow_cast_from_void_ptr	Allow all casts from void*.	False
functions_returning_aligned_pointer	No violation will be reported when casting the return value of these functions.	{'malloc', 'calloc', 'realloc', 'aligned_alloc'}
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
cast_increases_alignment	Pointer cast increases alignment from {} to {} bytes

CertC++-EXP37

Call functions with the correct number and type of arguments.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Do not call a function with the wrong number or type of arguments.

The C Standard identifies five distinct situations in which [undefined behavior](#) (UB) may arise as a result of invoking a function using a declaration that is incompatible with its definition or by supplying incorrect types or numbers of arguments:

UB	Description
26	A pointer is used to call a function whose type is not compatible with the referenced type {6.3.2.3}.
38	For a call to a function without a function prototype in scope, the number of arguments does not equal the number of parameters {6.5.2.2}.
39	For a call to a function without a function prototype in scope where the function is defined with a function prototype, either the prototype ends with an ellipsis or the types of the arguments after promotion are not compatible with the types of the parameters {6.5.2.2}.
40	For a call to a function without a function prototype in scope where the function is not defined with a function prototype, the types of the arguments after promotion are not compatible with those of the parameters after promotion (with certain exceptions) {6.5.2.2}.
41	A function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function {6.5.2.2}.

Functions that are appropriately declared (as in [DCL40-C. Do not create incompatible declarations of the same function or object](#)) will typically generate a compiler diagnostic message if they are supplied with the wrong number or types of arguments. However, there are cases in which supplying the incorrect arguments to a function will, at best, generate compiler [warnings](#). Although such warnings should be resolved, they do not prevent program compilation. (See [MSC00-C. Compile cleanly at high warning levels](#).)

Noncompliant Code Example

The header `<tgmath.h>` provides type-generic macros for math functions. Although most functions from the `<math.h>` header have a complex counterpart in `<complex.h>`, several functions do not. Calling any of the following type-generic functions with complex values is [undefined behavior](#).

Functions That Should Not Be Called with Complex Values

atan2()	erf()	fdim()	fmin()	ilogb()	llround()	logb()	nextafter()	rint()	tgamma()
cbrt()	erfc()	floor()	fmod()	ldexp()	log10()	lrint()	nexttoward()	round()	trunc()
ceil()	exp2()	fma()	frexp()	lgamma()	log1p()	lround()	remainder()	scalbn()	
copysign()	expm1()	fmax()	hypot()	llrint()	log2()	nearbyint()	remquo()	scalbln()	

This noncompliant code example attempts to take the base-2 logarithm of a complex number, resulting in undefined behavior:

```
#include <tgmath.h>

void func(void) {
    double complex c = 2.0 + 4.0 * I;
    double complex result = log2(c);
}
```

Compliant Solution [Complex Number]

If the `clog2()` function is not available for an implementation as an extension, the programmer can take the base-2 logarithm of a complex number, using `log()` instead of `log2()`, because `log()` can be used on complex arguments, as shown in this compliant solution:

```
#include <tgmath.h>

void func(void) {
    double complex c = 2.0 + 4.0 * I;
    double complex result = log(c)/log(2);
}
```

Compliant Solution [Real Number]

The programmer can use this compliant solution if the intent is to take the base-2 logarithm of the real part of the complex number:

```
#include <tgmath.h>

void func(void) {
    double complex c = 2.0 + 4.0 * I;
    double complex result = log2(creal(c));
}
```

Noncompliant Code Example

In this noncompliant example, the C standard library function `strchr()` is called through the function pointer `fp` declared with a prototype with incorrectly typed arguments. According to the C Standard, 6.3.2.3, paragraph 8 [[ISO/IEC 9899:2011](#)]

A pointer to a function of one type may be converted to a pointer to a function of another type and back again; the result shall compare equal to the original pointer. If a converted pointer is used to call a function whose type is not compatible with the referenced type, the behavior is undefined.

See [undefined behavior 26](#).

```
#include <stdio.h>
#include <string.h>

char *(*fp)();

int main(void) {
    const char *c;
    fp = strchr;
    c = fp('e', "Hello");
    printf("%s\n", c);
    return 0;
}
```

Compliant Solution

In this compliant solution, the function pointer `fp`, which points to the C standard library function `strchr()`, is declared with the correct parameters and is invoked with the correct number and type of arguments:

```
#include <stdio.h>
#include <string.h>

char *(*fp)(const char *, int);

int main(void) {
    const char *c;
    fp = strchr;
    c = fp("Hello",'e');
    printf("%s\n", c);
    return 0;
}
```

Noncompliant Code Example

In this noncompliant example, the function `f()` is defined to take an argument of type `long` but `f()` is called from another file with an argument of type `int`:

```
/* In another source file */
long f(long x) {
    return x < 0 ? -x : x;
}

/* In this source file, no f prototype in scope */
long f();

long g(int x) {
    return f(x);
}
```

Compliant Solution

In this compliant solution, the prototype for the function `f()` is included in the source file in the scope of where it is called, and the function `f()` is correctly called with an argument of type `long`:

```
/* In another source file */

long f(long x) {
    return x < 0 ? -x : x;
}

/* f prototype in scope in this source file */

long f(long x);

long g(int x) {
    return f((long)x);
}
```

Noncompliant Code Example [POSIX]

The POSIX function `open()` [IEEE Std 1003.1:2013] is a variadic function with the following prototype:

```
int open(const char *path, int oflag, ... );
```

The `open()` function accepts a third argument to determine a newly created file's access mode. If `open()` is used to create a new file and the third argument is omitted, the file may be created with unintended access permissions. (See [F1006-C. Create files with appropriate access permissions.](#))

In this noncompliant code example from a [vulnerability](#) in the `useradd()` function of the `shadow-utils` package [CVE-2006-1174](#), the third argument to `open()` is accidentally omitted:

```
fd = open(ms, O_CREAT | O_EXCL | O_WRONLY | O_TRUNC);
```

Technically, it is incorrect to pass a third argument to `open()` when not creating a new file (that is, with the `O_CREAT` flag not set).

Compliant Solution (POSIX)

In this compliant solution, a third argument is specified in the call to `open()`:

```
#include <fcntl.h>

void func(const char *ms, mode_t perms) {
    /* ... */
    int fd;
    fd = open(ms, O_CREAT | O_EXCL | O_WRONLY | O_TRUNC, perms);
    if (fd == -1) {
        /* Handle error */
    }
}
```

Risk Assessment

Calling a function with incorrect arguments can result in [unexpected](#) or unintended program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP37-C	Medium	Probable	High	P4	L3

Related Guidelines

CERT C Secure Coding Standard	DCL07-C. Include the appropriate type information in function declarators MSC00-C. Compile cleanly at high warning levels FI006-C. Create files with appropriate access permissions
ISO/IEC TR 24772:2013	Subprogram Signature Mismatch [OTR]
ISO/IEC TS 17961	Calling functions with incorrect arguments [argcomp]
MISRA C:2012	Rule 8.2 (required) Rule 17.3 (mandatory)
MITRE CWE	CWE-628 , Function Call with Incorrectly Specified Arguments CWE-686 , Function Call with Incorrect Argument Type

Bibliography

[CVE]	CVE-2006-1174
[ISO/IEC 9899:2011]	6.3.2.3, "Pointers" 6.5.2.2, "Function Calls"
[IEEE Std 1003.1:2013]	open()
[Spinellis 2006]	Section 2.6.1, "Incorrect Routine or Arguments"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/VQBc>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	4
real_only_macros		set(['atan2', 'nexttoward', 'nearbyint', 'rint', 'ldexp', 'hypot', 'logb', 'log2', 'floor', 'remainder', 'lround', 'ilogb', 'frexp', 'cbrt', 'log10', 'nextafter', 'remquo', 'scalbln', 'fmin', 'fmax', 'tgamma', 'scalbn', 'copysign', 'ceil', 'llrint', 'lrint', 'trunc', 'expm1', 'fdim', 'exp2', 'lgamma', 'erf', 'erfc', 'fmod', 'llround', 'fma', 'log1p', 'round'])
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high
use_pointer_analysis	Whether to use pointer analysis to verify indirect calls. Note: pointer analysis can only be used if CONFIG.Run_IRAnalysis_Checks is enabled.	True

Possible Messages

Name	Message
ellipsis_called_without_prototype	Cannot call function with ellipsis without a prototype declaration
open_mode_missing	{()} calls with {} must specify the 'mode' argument.
open_mode_redundant	The 'mode' argument is redundant in {()} calls without O_CREAT.
open_too_many_args	Too many arguments passed to {()}.
real_only_called_with_complex_arg	Calling a real-only function with a complex argument results in undefined behavior.
wrong_argument_number	Number of arguments at function call does not match number of parameters
wrong_argument_type	Parameter {} expects type '{}', but '{}' was given.

CertC++-EXP42

Do not compare padding data.

Input: IR

Source languages: C++

Details

The C Standard, 6.7.2.1 [[ISO/IEC 9899:2011](#)], states

There may be unnamed padding within a structure object, but not at its beginning. . . . There may be unnamed padding at the end of a structure or union.

Subclause 6.7.9, paragraph 9, states that

unnamed members of objects of structure and union type do not participate in initialization. Unnamed members of structure objects have indeterminate value even after initialization.

The only exception is that padding bits are set to zero when a static or thread-local object is implicitly initialized (paragraph 10):

If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate. If an object that has static or thread storage duration is not initialized explicitly, then:

- if it is an aggregate, every member is initialized (recursively) according to these rules, and any padding is initialized to zero bits;
- if it is a union, the first named member is initialized (recursively) according to these rules, and any padding is initialized to zero bits;

Because these padding values are unspecified, attempting a byte-by-byte comparison between structures can lead to incorrect results [[Summit 1995](#)].

Noncompliant Code Example

In this noncompliant code example, `memcmp()` is used to compare the contents of two structures, including any padding bytes:

```
#include <string.h>

struct s {
    char c;
    int i;
    char buffer[13];
};

void compare(const struct s *left, const struct s *right) {
    if (0 == memcmp(left, right, sizeof(struct s))) {
        /* ... */
    }
}
```

Compliant Solution

In this compliant solution, all of the fields are compared manually to avoid comparing any padding bytes:

```
#include <string.h>

struct s {
    char c;
    int i;
    char buffer[13];
};

void compare(const struct s *left, const struct s *right) {
    if ((left && right) &&
        (left->c == right->c) &&
        (left->i == right->i) &&
        (0 == memcmp(left->buffer, right->buffer, 13))) {
        /* ... */
    }
}
```

Exceptions

EXP42-C-EX1: A structure can be defined such that the members are aligned properly or the structure is packed using implementation-specific packing instructions. This is true only when the members' data types have no padding bits of their own and when their object representations are the same as their value representations. This frequently is not true for the `_Bool` type or floating-point types and need not be true for pointers. In such cases, the compiler does not insert padding, and use of functions such as `memcmp()` is acceptable.

This compliant example uses the `#pragma pack` compiler extension from Microsoft Visual Studio to ensure the structure members are packed as tightly as possible:

```
#include <string.h>

#pragma pack(push, 1)
struct s {
    char c;
    int i;
    char buffer[13];
};
#pragma pack(pop)

void compare(const struct s *left, const struct s *right) {
    if (0 == memcmp(left, right, sizeof(struct s))) {
        /* ... */
    }
}
```

Risk Assessment

Comparing padding bytes, when present, can lead to [unexpected program behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP42-C	Medium	Probable	Medium	P8	L2

Related Guidelines

ISO/IEC TS 17961	Comparison of padding data [padcomp]
SEI CERT C++ Coding Standard	EXP62-CPP. Do not access the bits of an object representation that are not part of the object's value representation

Bibliography

[ISO/IEC 9899:2011]	6.7.2.1, "Structure and Union Specifiers" 6.7.9, "Initialization"
[Summit 1995]	Question 2.8 Question 2.12

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/CoDYBq>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
allow_char	Whether to allow memcmp on 'char' type.	True
allow_composites_without_padding	Whether to allow using memcmp on structs and unions that have no padding bytes.	True
allow_float	Whether to allow memcmp on floating point types.	True
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
disallowed_memcmp_pointer_arg	Disallowed type of pointer argument.
memcmp_char_pointer_arg	memcmp shall not be used with char pointer argument, use strncmp instead.
memcmp_float	memcmp shall not be used to compare floats as the same value may be stored using different representations.
memcmp_padding	memcmp shall not be used to compare structs with padding.
memcmp_struct_pointer_arg	memcmp shall not be used with struct pointer argument as it would compare padding as well.
memcmp_union_pointer_arg	memcmp shall not be used with union pointer argument as it would compare padding and different kinds of representation.

CertC++-EXP45

Do not perform assignments in selection statements.

Input: IR

Source languages: C++

Details

Do not use the assignment operator in the contexts listed in the following table because doing so typically indicates programmer error and can result in [unexpected behavior](#).

Operator	Context
if	Controlling expression
while	Controlling expression
do ... while	Controlling expression
for	Second operand
?:	First operand
?:	Second or third operands, where the ternary expression is used in any of these contexts
&&	Either operand
	either operand
,	Second operand, when the comma expression is used in any of these contexts

Noncompliant Code Example

In this noncompliant code example, an assignment expression is the outermost expression in an `if` statement:

```
if (a = b) {
/* ... */
}
```

Although the intent of the code may be to assign `b` to `a` and test the value of the result for equality to 0, it is frequently a case of the programmer mistakenly using the assignment operator `=` instead of the equals operator `==`. Consequently, many compilers will warn about this condition, making this coding error detectable by adhering to [MSC00-C. Compile cleanly at high warning levels](#).

Compliant Solution (Unintentional Assignment)

When the assignment of `b` to `a` is not intended, the conditional block is now executed when `a` is equal to `b`:

```
if (a == b) {
/* ... */
}
```

Compliant Solution (Intentional Assignment)

When the assignment is intended, this compliant solution explicitly uses inequality as the outermost expression while performing the assignment in the inner expression:

```
if ((a = b) != 0) {
/* ... */
}
```

It is less desirable in general, depending on what was intended, because it mixes the assignment in the condition, but it is clear that the programmer intended the assignment to occur.

Noncompliant Code Example

In this noncompliant code example, the expression `x = y` is used as the controlling expression of the `while` statement:

```
do { /* ... */ } while (foo(), x = y);
```

The same result can be obtained using the `for` statement, which is specifically designed to evaluate an expression on each iteration of the loop, just before performing the test in its controlling expression:

```
for (; x; foo(), x = y) { /* ... */ }
```

Compliant Solution (Unintentional Assignment)

When the assignment of `y` to `x` is not intended, the conditional block should be executed only when `x` is equal to `y`, as in this compliant solution:

```
do { /* ... */ } while (foo(), x == y);
```

Compliant Solution (Intentional Assignment)

When the assignment is intended, this compliant solution can be used:

```
do { /* ... */ } while (foo(), (x = y) != 0);
```

Noncompliant Code Example

In this noncompliant example, the expression `p = q` is used as the controlling expression of the `while` statement:

```
do { /* ... */ } while (x = y, p = q);
```

Compliant Solution

In this compliant solution, the expression `x = y` is not used as the controlling expression of the `while` statement:

```
do { /* ... */ } while (x = y, p == q);
```

Noncompliant Code Example

This noncompliant code example has a typo that results in an assignment rather than a comparison.

```
while (ch = '\t' && ch == ' ' && ch == '\n') {
    /* ... */
}
```

Many compilers will warn about this condition. This coding error would typically be eliminated by adherence to [MSC00-C. Compile cleanly at high warning levels](#). Although this code compiles, it will cause [unexpected behavior](#) to an unsuspecting programmer. If the intent was to verify a string such as a password, user name, or group user ID, the code may produce significant [vulnerabilities](#) and require significant debugging.

Compliant Solution (RHS Variable)

When comparisons are made between a variable and a literal or const-qualified variable, placing the variable on the right of the comparison operation can prevent a spurious assignment.

In this code example, the literals are placed on the left-hand side of each comparison. If the programmer were to inadvertently use an assignment operator, the statement would assign `ch` to '`\t`', which is invalid and produces a diagnostic message.

```
while ('\t' == ch && ' ' == ch && '\n' == ch) {
    /* ... */
}
```

Due to the diagnostic, the typo will be easily spotted and fixed.

```
while ('\t' == ch && ' ' == ch && '\n' == ch) {
    /* ... */
}
```

As a result, any mistaken use of the assignment operator that could otherwise create a [vulnerability](#) for operations such as string verification will result in a compiler diagnostic regardless of compiler, warning level, or [implementation](#).

Exceptions

EXP45-C-EX1: Assignment can be used where the result of the assignment is itself an operand to a comparison expression or relational expression. In this compliant example, the expression `x = y` is itself an operand to a comparison operation:

```
if ((x = y) != 0) { /* ... */ }
```

EXP45-C-EX2: Assignment can be used where the expression consists of a single primary expression. The following code is compliant because the expression `x = y` is a single primary expression:

```
if ((x = y)) { /* ... */ }
```

The following controlling expression is noncompliant because `&&` is not a comparison or relational operator and the entire expression is not primary:

```
if ((v = w) && flag) { /* ... */ }
```

When the assignment of `v` to `w` is not intended, the following controlling expression can be used to execute the conditional block when `v` is equal to `w`:

```
if ((v == w) && flag) { /* ... */ };
```

When the assignment is intended, the following controlling expression can be used:

```
if (((v = w) != 0) && flag) { /* ... */ };
```

EXP45-C-EX3: Assignment can be used in a function argument or array index. In this compliant solution, the expression `x = y` is used in a function argument:

```
if (foo(x = y)) { /* ... */ }
```

Risk Assessment

Errors of omission can result in unintended program flow.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
EXP45-C	Low	Likely	Medium	P6	L2

Related Guidelines

SEI CERT C++ Coding Standard	EXP19-CPP. Do not perform assignments in conditional expressions
CERT Oracle Secure Coding Standard for Java	EXP51-J. Do not perform assignments in conditional expressions
ISO/IEC TR 24772:2013	Likely Incorrect Expression [KOA]
ISO/IEC TS 17961	No assignment in conditional expressions [boolasgn]
MITRE CWE	CWE-480, Use of Incorrect Operator

Bibliography

[Dutta 03]	"Best Practices for Programming in C"
[Hatton 1995]	Section 2.7.2, "Errors of Omission and Addition"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/nYFtAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
allow_in_extra_parens	Whether if <code>{(x = y)}</code> should be allowed (note the extra parens)	True
allow_in_relational_operator	Whether an assignment is allowed as operand of a comparison or relational operator, e.g. <code>(x = y) != 0</code>	True
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
assignment_inside_bool	Assignment inside boolean expression.

CertC++-EXP46

Do not use a bitwise operator with a Boolean-like operand.

Input: IR

Source languages: C++

Details

Mixing bitwise and relational operators in the same full expression can be a sign of a logic error in the expression where a logical operator is usually the intended operator. Do not use the bitwise AND (`&`), bitwise OR (`||`), or bitwise XOR (`^`) operators with an operand of type `_Bool`, or the result of a *relational-expression* or *equality-expression*. If the bitwise operator is intended, it should be indicated with use of a parenthesized expression.

Noncompliant Code Example

In this noncompliant code example, a bitwise `&` operator is used with the results of an *equality-expression*:

```
if (!(getuid() & geteuid() == 0)) {
/* ... */
}
```

Compliant Solution

This compliant solution uses the `&&` operator for the logical operation within the conditional expression:

```
if (!(getuid() && geteuid() == 0)) {
/* ... */
}
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP46-C	Low	Likely	Low	P9	L2

Related Guidelines

ISO/IEC TR 24772:2013	Likely Incorrect Expression [KOA]
MITRE CWE	CWE-480 , Use of incorrect operator

Bibliography

[Hatton 1995]	Section 2.7.2, "Errors of Omission and Addition"
-------------------------------	--

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/g4FtAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
allow_sizeof	Allows the use of sizeof(bool_var). The Misra rule forbids all uses of expressions of type bool except for those explicitly allowed; and does not allow sizeof. However this is likely an oversight; there is no good reason to prevent the use of sizeof(bool_var). This option also allows alignof(bool_var) in C++11.	True
allow_unevaluated_operator	Allows the use of boolean expressions in contexts where the expression is not evaluated. As of C++17, these are: typeid(bool_var), noexcept(bool_var) and decltype(bool_var). sizeof(bool_var) is controlled by the separate option 'allow_sizeof'.	True
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	9
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
bool_operand_in_bad_operator	Use of boolean operand in '{}' operator

CertC++-EXP47

Do not call va_arg with an argument of the incorrect type.

Input: IR

Source languages: C++

Details

The variable arguments passed to a variadic function are accessed by calling the `va_arg()` macro. This macro accepts the `va_list` representing the variable arguments of the function invocation and the type denoting the expected argument type for the argument being retrieved. The macro is typically invoked within a loop, being called once for each expected argument. However, there are no type safety guarantees that the type passed to `va_arg` matches the type passed by the caller, and there are generally no compile-time checks that prevent the macro from being invoked with no argument available to the function call. The C Standard, 7.16.1.1, states [[ISO/IEC 9899:2011](#)], in part:

If there is no actual next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined, except for the following cases:

- one type is a signed integer type, the other type is the corresponding unsigned integer type, and the value is representable in both types;
- one type is pointer to `void` and the other is a pointer to a character type.

Ensure that an invocation of the `va_arg()` macro does not attempt to access an argument that was not passed to the variadic function. Further, the type passed to the `va_arg()` macro must match the type passed to the variadic function after default argument promotions have been applied. Either circumstance results in [undefined behavior](#).

Noncompliant Code Example

This noncompliant code example attempts to read a variadic argument of type `unsigned char` with `va_arg()`. However, when a value of type `unsigned char` is passed to a variadic function, the value undergoes default argument promotions, resulting in a value of type `int` being passed.

```
#include <stdarg.h>
#include <stddef.h>

void func(size_t count, ...) {
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        unsigned char c = va_arg(ap, unsigned char);
        // ...
    }
    va_end(ap);
}

void f(void) {
    unsigned char c = 0x12;
    func(1, c);
}
```

Compliant Solution

The compliant solution accesses the variadic argument with type `int`, and then casts the resulting value to type `unsigned char`:

```
#include <stdarg.h>
#include <stddef.h>

void func(size_t count, ...) {
    va_list ap;
    va_start(ap, count);
    if (count > 0) {
        unsigned char c = (unsigned char)va_arg(ap, int);
        // ...
    }
    va_end(ap);
}

void f(void) {
    unsigned char c = 0x12;
    func(1, c);
}
```

Noncompliant Code Example

This noncompliant code example assumes that at least one variadic argument is passed to the function, and attempts to read it using the `va_arg()` macro. This pattern arises frequently when a variadic function uses a sentinel value to denote the end of the variable argument list. However, the caller does not pass any extra arguments to the function, resulting in undefined behavior.

```
#include <stdarg.h>

void func(const char *cp, ...) {
    va_list ap;
    va_start(ap, cp);
    int val = va_arg(ap, int);
    // ...
    va_end(ap);
}

void f(void) {
    func("The only argument");
}
```

Compliant Solution

It is not possible for the variadic function to determine how many arguments are actually provided to the function call; that information must be passed in an out-of-band way. Oftentimes this results in the information being encoded in the initial parameter, as in this compliant solution:

```
#include <stdarg.h>
#include <stddef.h>

void func(const char *cp, size_t numArgs, ...) {
    va_list ap;
    va_start(ap, cp);
    if (numArgs > 0) {
        int val = va_arg(ap, int);
        // ...
    }
    va_end(ap);
}

void f(void) {
    func("The only argument", 0);
}
```

Risk Assessment

Incorrect use of `va_arg()` results in undefined behavior that can include accessing stack memory.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP47-C	Medium	Likely	High	P6	L2

Bibliography

[ISO/IEC 9899:2011]	Subclause 7.16, "Variable Arguments <stdarg.h>" Subclause 6.5.2.2, "Function calls"
-------------------------------------	--

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/BYACQW>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
va_arg_missing	Call requires at least one argument for variadic parameter.
va_arg_with_unpromoted_type	Type {} does not match promoted type {}.

CertC++-EXP50

Do not depend on the order of evaluation for side effects.

Input: IR

Source languages: C++

Details

In C++, modifying an object, calling a library I/O function, accessing a `volatile`-qualified value, or calling a function that performs one of these actions are ways to modify the state of the execution environment. These actions are called *side effects*. All relationships between value computations and side effects can be described in terms of sequencing of their evaluations. The C++ Standard, [intro.execution], paragraph 13 [[ISO/IEC 14882-2014](#)], establishes three sequencing terms:

Sequenced before is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, 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]

Paragraph 15 further states (nonnormative text removed for brevity) the following:

Except where noted, evaluations of operands of individual operators and of subexpressions of individual expressions are unsequenced. ... 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, the behavior is undefined. ... 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. ... 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. Several contexts in C++ cause evaluation of a function call, even though no corresponding function call syntax appears in the translation unit. ... 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.

Do not allow the same scalar object to appear in side effects or value computations in both halves of an unsequenced or indeterminately sequenced operation.

The following expressions have sequencing restrictions that deviate from the usual unsequenced ordering [[ISO/IEC 14882-2014](#)]:

- In postfix `++` and `--` expressions, the value computation is sequenced before the modification of the operand. ([expr.post.incr], paragraph 1)
- In logical `&&` expressions, 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. ([expr.log.and], paragraph 2)
- In logical `||` expressions, 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. ([expr.log.or], paragraph 2)

- In conditional ?: expressions, 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 (whichever is evaluated). ([expr.cond], paragraph 1)
- In assignment expressions (including compound assignments), the assignment is sequenced after the value computations of left and right operands and before the value computation of the assignment expression. ([expr.ass], paragraph 1)
- In comma , expressions, 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. ([expr.comma], paragraph 1)
- When evaluating initializer lists, the value computation and side effect associated with each *initializer-clause* is sequenced before every value computation and side effect associated with a subsequent *initializer-clause*. ([dcl.init.list], paragraph 4)
- When a signal handler is executed as a result of a call to `std::raise()`, the execution of the handler is sequenced after the invocation of `std::raise()` and before its return. ([intro.execution], paragraph 6)
- The completions of the destructors for all initialized objects with thread storage duration within a thread are sequenced before the initiation of the destructors of any object with static storage duration. ([basic.start.term], paragraph 1)
- In a *new-expression*, initialization of an allocated object is sequenced before the value computation of the *new-expression*. ([expr.new], paragraph 18)
- When a default constructor is called to initialize an element of an array and the constructor has at least one default argument, the destruction of every temporary created in a default argument is sequenced before the construction of the next array element, if any. ([class.temporary], paragraph 4)
- The destruction of a temporary whose lifetime is not extended by being bound to a reference is sequenced before the destruction of every temporary that is constructed earlier in the same full-expression. ([class.temporary], paragraph 5)
- Atomic memory ordering functions can explicitly determine the sequencing order for expressions. ([atomics.order] and [atomics.fences])

This rule means that statements such as

```
i = i + 1;
a[i] = i;
```

have defined behavior, and statements such as the following do not.

```
// i is modified twice in the same full expression
i = ++i + 1;

// i is read other than to determine the value to be stored
a[i++] = i;
```

Not all instances of a comma in C++ code denote use of the comma operator. For example, the comma between arguments in a function call is *not* the comma operator. Additionally, overloaded operators behave the same as a function call, with the operands to the operator acting as arguments to a function call.

Noncompliant Code Example

In this noncompliant code example, `i` is evaluated more than once in an unsequenced manner, so the behavior of the expression is [undefined](#).

```
void f(int i, const int *b) {
    int a = i + b[++i];
    // ...
}
```

Compliant Solution

These examples are independent of the order of evaluation of the operands and can each be interpreted in only one way.

```
void f(int i, const int *b) {
    ++i;
    int a = i + b[i];
    // ...
}
```

```
void f(int i, const int *b) {
    int a = i + b[i + 1];
    ++i;
    // ...
}
```

Noncompliant Code Example

The call to `func()` in this noncompliant code example has [undefined behavior](#) because the argument expressions are unsequenced.

```
extern void func(int i, int j);

void f(int i) {
    func(i++, i);
}
```

The first (left) argument expression reads the value of `i` (to determine the value to be stored) and then modifies `i`. The second (right) argument expression reads the value of `i`, but not to determine the value to be stored in `i`. This additional attempt to read the value of `i` has [undefined behavior](#).

Compliant Solution

This compliant solution is appropriate when the programmer intends for both arguments to `func()` to be equivalent.

```
extern void func(int i, int j);

void f(int i) {
    i++;
    func(i, i);
}
```

This compliant solution is appropriate when the programmer intends for the second argument to be 1 greater than the first.

```
extern void func(int i, int j);
void f(int i) {
    int j = i++;
    func(j, i);
}
```

Noncompliant Code Example

This noncompliant code example is similar to the previous noncompliant code example. However, instead of calling a function directly, this code calls an overloaded `operator<<()`. Overloaded operators are equivalent to a function call and have the same restrictions regarding the sequencing of the function call arguments. This means that the operands are not evaluated left-to-right, but are unsequenced with respect to one another. Consequently, this noncompliant code example has undefined behavior.

```
#include <iostream>
void f(int i) {
    std::cout << i++ << i << std::endl;
}
```

Compliant Solution

In this compliant solution, two calls are made to `operator<<()`, ensuring that the arguments are printed in a well-defined order.

```
#include <iostream>
void f(int i) {
    std::cout << i++;
    std::cout << i << std::endl;
}
```

Noncompliant Code Example

The order of evaluation for function arguments is unspecified. This noncompliant code example exhibits [unspecified behavior](#) but not [undefined behavior](#).

```
extern void c(int i, int j);
int glob;

int a() {
    return glob + 10;
}

int b() {
    glob = 42;
    return glob;
}

void f() {
    c(a(), b());
}
```

The order in which `a()` and `b()` are called is unspecified; the only guarantee is that both `a()` and `b()` will be called before `c()` is called. If `a()` or `b()` rely on shared state when calculating their return value, as they do in this example, the resulting arguments passed to `c()` may differ between compilers or architectures.

Compliant Solution

In this compliant solution, the order of evaluation for `a()` and `b()` is fixed, and so no [unspecified behavior](#) occurs.

```
extern void c(int i, int j);
int glob;

int a() {
    return glob + 10;
}

int b() {
    glob = 42;
    return glob;
}

void f() {
    int a_val, b_val;

    a_val = a();
    b_val = b();

    c(a_val, b_val);
}
```

Risk Assessment

Attempting to modify an object in an unsequenced or indeterminately sequenced evaluation may cause that object to take on an unexpected value, which can lead to unexpected program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP50-CPP	Medium	Probable	Medium	P8	L2

SEI CERT C Coding Standard	EXP30-C. Do not depend on the order of evaluation for side effects
--	--

Bibliography

[ISO/IEC 14882-2014]	Subclause 1.9, "Program Execution"
[MISRA 2008]	Rule 5-0-1 (Required)

Excerpt from SEI CERT C++ Coding Standard Wiki [\[https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP50-
CPP.+Do+not+depend+on+the+order+of+evaluation+for+side+effects\]](https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP50-CPP.+Do+not+depend+on+the+order+of+evaluation+for+side+effects), Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
ignore_exceptions	Ignore possible exceptions in the computation of side effects. (this option only has an effect if report_calls is enabled)	False
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
report_calls	If True, unsequenced function calls are reported.	True

Possible Messages

Name	Message
unsequenced_call_volatile	Function calls unsequenced with volatile access
unsequenced_calls	Unsequenced function calls
unsequenced_read_write	Unsequenced read and write accesses
unsequenced_volatile	Unsequenced volatile accesses
unsequenced_writes	Unsequenced writes

CertC++-EXP52

Do not rely on side effects in unevaluated operands.

Input: IR

Source languages: C++

Details

Some expressions involve operands that are *unevaluated*. The C++ Standard, [expr], paragraph 8 [\[ISO/IEC 14882-2014\]](#) states the following:

In some contexts, *unevaluated operands* appear. 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 (5.1) and naming of objects or functions does not, by itself, require that a definition be provided. — end note]

The following expressions do not evaluate their operands: `sizeof()`, `typeid()`, `noexcept()`, `decltype()`, and `declval()`.

Because an unevaluated operand in an expression is not evaluated, no side effects from that operand are triggered. Reliance on those side effects will result in unexpected behavior. Do not rely on side effects in unevaluated operands.

Unevaluated expression operands are used when the declaration of an object is required but the definition of the object is not. For instance, in the following example, the function `f()` is overloaded, relying on the unevaluated expression operand to select the desired overload, which is then used to determine the result of the `sizeof()` expression.

```
int f(int);
double f(double);
size_t size = sizeof(f(0));
```

Such a use does not rely on the side effects of `f()` and consequently conforms to this guideline.

Noncompliant Code Example (`sizeof`)

In this noncompliant code example, the expression `a++` is not evaluated.

```
#include <iostream>
void f() {
    int a = 14;
    int b = sizeof(a++);
    std::cout << a << ", " << b << std::endl;
}
```

Consequently, the value of `a` after `b` has been initialized is 14.

Compliant Solution (`sizeof`)

In this compliant solution, the variable `a` is incremented outside of the `sizeof` operator.

```
#include <iostream>
void f() {
    int a = 14;
    int b = sizeof(a);
    ++a;
    std::cout << a << ", " << b << std::endl;
}
```

Noncompliant Code Example (`decltype`)

In this noncompliant code example, the expression `i++` is not evaluated within the `decltype` specifier.

```
#include <iostream>

void f() {
    int i = 0;
    decltype(i++) h = 12;
    std::cout << i;
}
```

Consequently, the value of `i` remains 0.

Compliant Solution (`decltype`)

In this compliant solution, `i` is incremented outside of the `decltype` specifier so that it is evaluated as desired.

```
#include <iostream>

void f() {
    int i = 0;
    decltype(i) h = 12;
    ++i;
    std::cout << i;
}
```

Exceptions

EXP52-CPP-EX1: It is permissible for an expression with side effects to be used as an unevaluated operand in a macro definition or [SFINAE](#) context. Although these situations rely on the side effects to produce valid code, they typically do not rely on values produced as a result of the side effects.

The following code is an example of compliant code using an unevaluated operand in a macro definition.

```
void small(int x);
void large(long long x);

#define m(x) do { if (sizeof(x) == sizeof(int)) {
```

The expansion of the macro `m` will result in the expression `++i` being used as an unevaluated operand to `sizeof()`. However, the expectation of the programmer at the expansion loci is that `i` is preincremented only once. Consequently, this is a safe macro and complies with [PRE31-C. Avoid side effects in arguments to unsafe macros](#). Compliance with that rule is especially important for code that follows this exception.

The following code is an example of compliant code using an unevaluated operand in a SFINAE context to determine whether a type can be postfix incremented.

```
#include <iostream>
#include <type_traits>
#include <utility>

template <typename T>
class is_incrementable {
    typedef char one[1];
    typedef char two[2];
    static one &is_incrementable_helper decltype(std::declval()++) *p;
    static two &is_incrementable_helper(...);

public:
    static const bool value = sizeof(is_incrementable_helper(nullptr)) == sizeof(one);
};

void f() {
    std::cout << std::boolalpha << is_incrementable::value;
```

In an instantiation of `is_incrementable`, the use of the postfix increment operator generates side effects that are used to determine whether the type is postfix incrementable. However, the value result of these side effects is discarded, so the side effects are used only for SFINAE.

Risk Assessment

If expressions that appear to produce side effects are an unevaluated operand, the results may be different than expected. Depending on how this result is used, it can lead to unintended program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP52-CPP	Low	Unlikely	Low	P3	L3

Related Guidelines

SEI CERT C Coding Standard	EXP44-C. Do not rely on side effects in operands to sizeof, _Alignof, or _Generic
--	---

Bibliography

[ISO/IEC 14882-2014]	Clause 5, "Expressions" Subclause 20.2.5, "Function Template <code>declval</code> "
--------------------------------------	--

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP52-CPP.+Do+not+rely+on+side+effects+in+unevaluated+operands>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
<code>allow_in_macro_definition</code>	Whether to allow unevaluated operands in macro definition contexts.	True
<code>expressions_to_check</code>	Mapping of physical expression classes to tuples of (<code>field_name</code> , <code>expr_name</code>) to check.	<code>dict{...}</code>
<code>level</code>	Grouping of priorities into different levels	3
<code>likelihood</code>	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
<code>priority</code>	Priority based on the combination of severity, likelihood and remediation cost	3
<code>recommendation</code>	Whether this check is classified as a recommendation or rule	False
<code>remediation_cost</code>	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
<code>side_effect_in_unevaluated</code>	Operand of "{expr}" shall not contain side effects

CertC++-EXP55

Do not access a cv-qualified object through a cv-unqualified type.

Input: IR

Source languages: C++

Details

The C++ Standard, [dcl.type.cv], paragraph 4 [[ISO/IEC 14882-2014](#)], states the following:

Except that any class member declared `mutable` can be modified, any attempt to modify a `const` object during its lifetime results in undefined behavior.

Similarly, paragraph 6 states the following:

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.

Do not cast away a `const` qualification to attempt to modify the resulting object. The `const` qualifier implies that the API designer does not intend for that object to be modified despite the possibility it may be modifiable. Do not cast away a `volatile` qualification; the `volatile` qualifier implies that the API designer intends the object to be accessed in ways unknown to the compiler, and any access of the volatile object results in `undefined behavior`.

Noncompliant Code Example

In this noncompliant code example, the function `g()` is passed a `const int &`, which is then cast to an `int &` and modified. Because the referenced value was previously declared as `const`, the assignment operation results in [undefined behavior](#).

```
void g(const int &ci) {
    int &ir = const_cast<int &>(ci);
    ir = 42;
}

void f() {
    const int i = 4;
    g(i);
}
```

Compliant Solution

In this compliant solution, the function `g()` is passed an `int &`, and the caller is required to pass an `int` that can be modified.

```
void g(int &i) {
    i = 42;
}

void f() {
    int i = 4;
    g(i);
}
```

Noncompliant Code Example

In this noncompliant code example, a `const`-qualified method is called that attempts to cache results by casting away the `const`-qualifier of `this`. Because `s` was declared `const`, the mutation of `cachedValue` results in [undefined behavior](#).

```
#include <iostream>

class S {
    int cachedValue;

    int compute_value() const; // expensive
public:
    S() : cachedValue(0) {}

    // ...
    int get_value() const {
        if (!cachedValue) {
            const_cast<S *>(this)->cachedValue = compute_value();
        }
        return cachedValue;
    }
};

void f() {
    const S s;
    std::cout << s.get_value() << std::endl;
}
```

Compliant Solution

This compliant solution uses the `mutable` keyword when declaring `cachedValue`, which allows `cachedValue` to be mutated within a `const` context without triggering [undefined behavior](#).

```
#include <iostream>

class S {
    mutable int cachedValue;

    int compute_value() const; // expensive
public:
    S() : cachedValue(0) {}

    // ...
    int get_value() const {
        if (!cachedValue) {
            cachedValue = compute_value();
        }
        return cachedValue;
    }
};

void f() {
    const S s;
    std::cout << s.get_value() << std::endl;
}
```

Noncompliant Code Example

In this noncompliant code example, the `volatile` value `s` has the `volatile` qualifier cast away, and an attempt is made to read the value within `g()`, resulting in [undefined behavior](#).

```
#include <iostream>

struct S {
    int i;
```

```

S(int i) : i(i) {}

};

void g(S &s) {
    std::cout << s.i << std::endl;
}

void f() {
    volatile S s(12);
    g(const_cast<S &>(s));
}

```

Compliant Solution

This compliant solution assumes that the volatility of `s` is required, so `g()` is modified to accept a `volatile S &`.

```

#include <iostream>

struct S {
    int i;

    S(int i) : i(i) {}

};

void g(volatile S &s) {
    std::cout << s.i << std::endl;
}

void f() {
    volatile S s(12);
    g(s);
}

```

Exceptions

EXP55-CPP-EX1: An exception to this rule is allowed when it is necessary to cast away `const` when invoking a legacy API that does not accept a `const` argument, provided the function does not attempt to modify the referenced variable. However, it is always preferable to modify the API to be `const`-correct when possible. For example, the following code casts away the `const` qualification of `INVFNNAME` in the call to the `audit_log()` function.

```

// Legacy function defined elsewhere - cannot be modified; does not attempt to
// modify the contents of the passed parameter.
void audit_log(char *errstr);

void f() {
    const char INVFNNAME[] = "Invalid file name.";
    audit_log(const_cast<char *>(INVFNNAME));
}

```

Risk Assessment

If the object is declared as being constant, it may reside in write-protected memory at runtime. Attempting to modify such an object may lead to [abnormal program termination](#) or a [denial-of-service attack](#). If an object is declared as being volatile, the compiler can make no assumptions regarding access of that object. Casting away the volatility of an object can result in reads or writes to the object being reordered or elided entirely, resulting in abnormal program execution.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP55-CPP	Medium	Probable	Medium	P8	L2

Related Guidelines

SEI CERT C Coding Standard	EXP32-C. Do not access a volatile object through a nonvolatile reference EXP40-C. Do not modify constant objects
--	---

Bibliography

[ISO/IEC 14882-2014]	Subclause 7.1.6.1, "The cv-qualifiers"
[Sutter 2004]	Item 94, "Avoid Casting Away const"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP55-CPP.+Do+not+access+a+cv-qualified+object+through+a+cv-unqualified+type>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
check_const	Whether casts affecting const-qualifiers should be reported	True
check_volatile	Whether casts affecting volatile-qualifiers should be reported	True
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
only_top_level	Whether the check only applies to the top-level pointee or recursively to deeper pointer levels as well	False
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
cast_removes_const	Cast removes const qualification
cast_removes_volatile	Cast removes volatile qualification
implicit_cast_removes_const	Implicit cast removes const qualification
implicit_cast_removes_volatile	Implicit cast removes volatile qualification

CertC++-EXP59

Use `offsetof()` on valid types and members.

Input: IR

Source languages: C++

Details

The `offsetof()` macro is defined by the C Standard as a portable way to determine the offset, expressed in bytes, from the start of the object to a given member of that object. The C Standard, subclause 7.17, paragraph 3 [[ISO/IEC 9899:1999](#)], in part, specifies the following:

`offsetof(type, member-designator)` which expands to an integer constant expression that has type `size_t`, the value of which is the offset in bytes, to the structure member [designated by *member-designator*], from the beginning of its structure [designated by *type*]. The type and member designator shall be such that given `static type t`; then the expression `&(t.member-designator)` evaluates to an address constant. [If the specified member is a bit-field, the behavior is undefined.]

The C++ Standard, [support.types], paragraph 4 [[ISO/IEC 14882-2014](#)], places additional restrictions beyond those set by the C Standard:

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*, the results are undefined. The expression `offsetof(type, member-designator)` is never type-dependent and it is value-dependent 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.

When specifying the type argument for the `offsetof()` macro, pass only a standard-layout class. The full description of a standard-layout class can be found in paragraph 7 of the [class] clause of the C++ Standard, or the type can be checked with the `std::is_standard_layout<>` type trait. When specifying the member designator argument for the `offsetof()` macro, do not pass a bit-field, static data member, or function member. Passing an invalid type or member to the `offsetof()` macro is [undefined behavior](#).

Noncompliant Code Example

In this noncompliant code example, a type that is not a standard-layout class is passed to the `offsetof()` macro, resulting in [undefined behavior](#).

```
#include <cstddef>

struct D {
    virtual void f() {}
    int i;
};

void f() {
    size_t off = offsetof(D, i);
    // ...
}
```

Implementation Details

The noncompliant code example does not emit a diagnostic when compiled with the `/Wall` switch in [Microsoft Visual Studio](#) 2015 on x86, resulting in `off`

being 4, due to the presence of a vtable for type D.

Compliant Solution

It is not possible to determine the offset to i within D because D is not a standard-layout class. However, it is possible to make a standard-layout class within D if this functionality is critical to the application, as demonstrated by this compliant solution.

```
#include <cstddef>

struct D {
    virtual void f() {}
    struct InnerStandardLayout {
        int i;
    } inner;
};

void f() {
    size_t off = offsetof(D::InnerStandardLayout, i);
    // ...
}
```

Noncompliant Code Example

In this noncompliant code example, the offset to i is calculated so that a value can be stored at that offset within buffer. However, because i is a static data member of the class, this example results in [undefined behavior](#). According to the C++ Standard, [class.static.data], paragraph 1 [[ISO/IEC 14882-2014](#)], static data members are not part of the subobjects of a class.

```
#include <cstddef>

struct S {
    static int i;
    // ...
};

int S::i = 0;

extern void store_in_some_buffer(void *buffer, size_t offset, int val);
extern void *buffer;

void f() {
    size_t off = offsetof(S, i);
    store_in_some_buffer(buffer, off, 42);
}
```

Implementation Details

The noncompliant code example does not emit a diagnostic when compiled with the /Wall switch in Microsoft Visual Studio 2015 on x86, resulting in off being a large value representing the offset between the null pointer address 0 and the address of the static variable S::i.

Compliant Solution

Because static data members are not a part of the class layout, but are instead an entity of their own, this compliant solution passes the address of the static member variable as the buffer to store the data in and passes 0 as the offset.

```
#include <cstddef>

struct S {
    static int i;
    // ...
};

int S::i = 0;

extern void store_in_some_buffer(void *buffer, size_t offset, int val);

void f() {
    store_in_some_buffer(&S::i, 0, 42);
}
```

Risk Assessment

Passing an invalid type or member to offsetof() can result in [undefined behavior](#) that might be [exploited](#) to cause data integrity violations or result in incorrect values from the macro expansion.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP59-CPP	Medium	Unlikely	Medium	P4	L3

Bibliography

[ISO/IEC 9899:1999]	Subclause 7.17, "Common Definitions <stddef.h>"
[ISO/IEC 14882-2014]	Subclause 9.4.2, "Static Data Members" Subclause 18.2, "Types"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP59-CPP+Use+of+offsetof%28%29+on+valid+types+and+members>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
reported_messages	The set of messages regarding syntax and constraint violations.	set([1424, 1425, 1426, 1427])
reported_severities	List of severities to display.	('error', 'warning')
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

CertC++-INT34

Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand.

Input: IR

Source languages: C++

Details

Bitwise shifts include left-shift operations of the form *shift-expression* \ll *additive-expression* and right-shift operations of the form *shift-expression* \gg *additive-expression*. The standard integer promotions are first performed on the operands, each of which has an integer type. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is [undefined](#). (See [undefined behavior 51](#).)

Do not shift an expression by a negative number of bits or by a number greater than or equal to the *precision* of the promoted left operand. The precision of an integer type is the number of bits it uses to represent values, excluding any sign and padding bits. For unsigned integer types, the width and the precision are the same; whereas for signed integer types, the width is one greater than the precision. This rule uses precision instead of width because, in almost every case, an attempt to shift by a number of bits greater than or equal to the precision of the operand indicates a bug (logic error). A logic error is different from overflow, in which there is simply a representational deficiency. In general, shifts should be performed only on unsigned operands. (See [INT13-C. Use bitwise operators only on unsigned operands](#).)

Noncompliant Code Example (Left Shift, Unsigned Type)

The result of `E1 << E2` is `E1` left-shifted `E2` bit positions; vacated bits are filled with zeros.

According to the C Standard, if `E1` has an unsigned type, the value of the result is $E1 * 2^{E2}$, reduced modulo 1 more than the maximum value representable in the result type.

This noncompliant code example fails to ensure that the right operand is less than the precision of the promoted left operand:

```
void func(unsigned int ui_a, unsigned int ui_b) {
    unsigned int urestult = ui_a << ui_b;
    /* ... */
}
```

Compliant Solution (Left Shift, Unsigned Type)

This compliant solution eliminates the possibility of shifting by greater than or equal to the number of bits that exist in the precision of the left operand:

```
#include <limits.h>
#include <stddef.h>
#include <inttypes.h>

extern size_t __popcount(uintmax_t);
#define PRECISION(x) __popcount(x)

void func(unsigned int ui_a, unsigned int ui_b) {
    unsigned int urestult = 0;
    if (ui_b >= PRECISION(UINT_MAX)) {
        /* Handle error */
    } else {
        urestult = ui_a << ui_b;
    }
    /* ... */
}
```

The `PRECISION()` macro and `popcount()` function provide the correct precision for any integer type. (See [INT35-C. Use correct integer precisions](#).)

Modulo behavior resulting from left-shifting an unsigned integer type is permitted by exception INT30-EX3 to [INT30-C. Ensure that unsigned integer operations do not wrap](#).

Noncompliant Code Example {Left Shift, Signed Type}

The result of $E1 \ll E2$ is $E1$ left-shifted $E2$ bit positions; vacated bits are filled with zeros. If $E1$ has a signed type and nonnegative value, and $E1 * 2^{E2}$ is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

This noncompliant code example fails to ensure that left and right operands have nonnegative values and that the right operand is less than the precision of the promoted left operand. This example does check for signed integer overflow in compliance with [INT32-C. Ensure that operations on signed integers do not result in overflow](#).

```
#include <limits.h>
#include <stddef.h>
#include <inttypes.h>

void func(signed long si_a, signed long si_b) {
    signed long result;
    if (si_a > (LONG_MAX >> si_b)) {
        /* Handle error */
    } else {
        result = si_a << si_b;
    }
    /* ... */
}
```

Shift operators and other bitwise operators should be used only with unsigned integer operands in accordance with [INT13-C. Use bitwise operators only on unsigned operands](#).

Compliant Solution {Left Shift, Signed Type}

In addition to the check for overflow, this compliant solution ensures that both the left and right operands have nonnegative values and that the right operand is less than the precision of the promoted left operand:

```
#include <limits.h>
#include <stddef.h>
#include <inttypes.h>

extern size_t __builtin_popcount(uintmax_t);
#define PRECISION(x) __builtin_popcount(x)

void func(signed long si_a, signed long si_b) {
    signed long result;
    if ((si_a < 0) || (si_b < 0) || (si_b >= PRECISION(ULONG_MAX)) ||
        (si_a > (LONG_MAX >> si_b))) {
        /* Handle error */
    } else {
        result = si_a << si_b;
    }
    /* ... */
}
```

Noncompliant Code Example {Right Shift}

The result of $E1 \gg E2$ is $E1$ right-shifted $E2$ bit positions. If $E1$ has an unsigned type or if $E1$ has a signed type and a nonnegative 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](#) and can be either an arithmetic (signed) shift or a logical (unsigned) shift.

This noncompliant code example fails to test whether the right operand is greater than or equal to the precision of the promoted left operand, allowing undefined behavior:

```
void func(unsigned int ui_a, unsigned int ui_b) {
    unsigned int urest = ui_a >> ui_b;
    /* ... */
}
```

When working with signed operands, making assumptions about whether a right shift is implemented as an arithmetic (signed) shift or a logical (unsigned) shift can also lead to [vulnerabilities](#). (See [INT13-C. Use bitwise operators only on unsigned operands](#).)

Compliant Solution {Right Shift}

This compliant solution eliminates the possibility of shifting by greater than or equal to the number of bits that exist in the precision of the left operand:

```
#include <limits.h>
#include <stddef.h>
#include <inttypes.h>

extern size_t __builtin_popcount(uintmax_t);
#define PRECISION(x) __builtin_popcount(x)

void func(unsigned int ui_a, unsigned int ui_b) {
    unsigned int urest = 0;
    if (ui_b >= PRECISION(UINT_MAX)) {
        /* Handle error */
    } else {
        urest = ui_a >> ui_b;
    }
    /* ... */
}
```

Implementation Details

GCC has no options to handle shifts by negative amounts or by amounts outside the width of the type predictably or to trap on them; they are always treated as undefined. Processors may reduce the shift amount modulo the width of the type. For example, 32-bit right shifts are implemented using the following instruction on x86-32:

```
sarl %cl, %eax
```

The `sar` instruction takes a bit mask of the least significant 5 bits from `%cl` to produce a value in the range [0, 31] and then shift `%eax` that many bits:

```
// 64-bit right shifts on IA-32 platforms become
shrdl %edx, %eax
sarl %cl, %edx
```

where `%eax` stores the least significant bits in the doubleword to be shifted, and `%edx` stores the most significant bits.

Risk Assessment

Although shifting a negative number of bits or shifting a number of bits greater than or equal to the width of the promoted left operand is undefined behavior in C, the risk is generally low because processors frequently reduce the shift amount modulo the width of the type.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT34-C	Low	Unlikely	Medium	P2	L3

Related Guidelines

SEI CERT C Coding Standard	INT13-C. Use bitwise operators only on unsigned operands INT35-C. Use correct integer precisions INT32-C. Ensure that operations on signed integers do not result in overflow
ISO/IEC TR 24772:2013	Arithmetic Wrap-Around Error [FIF]

Bibliography

[C99 Rationale 2003]	6.5.7, "Bitwise Shift Operators"
[Dowd 2006]	Chapter 6, "C Language Issues"
[Seacord 2013b]	Chapter 5, "Integer Security"
[Viega 2005]	Section 5.2.7, "Integer Overflow"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/IRe>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
report_nonconst_cases	Whether a message should be printed for non-constant cases missing a range-checking if statement	True
use_essential_type	Whether essential type should be computed for allowed value range.	False

Possible Messages

Name	Message
excessive_shift	Right operand of shift operator has value {}, but should be in range 0 to {}
possible_excessive_shift	Right operand of shift operator should be checked to be in range 0 to {}
shift_right_negative	If first operand is negative, using this bitwise operator relies on implementation-defined behaviour

CertC++-INT36

Converting a pointer to integer or integer to pointer.

Input: IR

Source languages: C++

Details

Although programmers often use integers and pointers interchangeably in C, pointer-to-integer and integer-to-pointer conversions are [implementation-defined](#).

Conversions between integers and pointers can have undesired consequences depending on the [implementation](#). According to the C Standard, subclause 6.3.2.3 [[ISO/IEC 9899:2011](#)],

An integer may be converted to any pointer type. Except as previously specified, the result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation.

Any pointer type may be converted to an integer type. Except as previously specified, the result is implementation-defined. If the result cannot be represented in the integer type, the behavior is undefined. The result need not be in the range of values of any integer type.

Do not convert an integer type to a pointer type if the resulting pointer is incorrectly aligned, does not point to an entity of the referenced type, or is a [trap representation](#).

Do not convert a pointer type to an integer type if the result cannot be represented in the integer type. (See [undefined behavior 24](#).)

The mapping between pointers and integers must be consistent with the addressing structure of the execution environment. Issues may arise, for example, on architectures that have a segmented memory model.

Noncompliant Code Example

The size of a pointer can be greater than the size of an integer, such as in an implementation where pointers are 64 bits and unsigned integers are 32 bits. This code example is noncompliant on such implementations because the result of converting the 64-bit `ptr` cannot be represented in the 32-bit integer type:

```
void f(void) {
    char *ptr;
    /* ... */
    unsigned int number = (unsigned int)ptr;
    /* ... */
}
```

Compliant Solution

Any valid pointer to `void` can be converted to `intptr_t` or `uintptr_t` and back with no change in value. (See INT36-EX2.) The C Standard guarantees that a pointer to `void` may be converted to or from a pointer to any object type and back again and that the result must compare equal to the original pointer. Consequently, converting directly from a `char *` pointer to a `uintptr_t`, as in this compliant solution, is allowed on implementations that support the `uintptr_t` type.

```
#include <stdint.h>

void f(void) {
    char *ptr;
    /* ... */
    uintptr_t number = (uintptr_t)ptr;
    /* ... */
}
```

Noncompliant Code Example

In this noncompliant code example, the pointer `ptr` is converted to an integer value. The high-order 9 bits of the number are used to hold a flag value, and the result is converted back into a pointer. This example is noncompliant on an implementation where pointers are 64 bits and unsigned integers are 32 bits because the result of converting the 64-bit `ptr` cannot be represented in the 32-bit integer type.

```
void func(unsigned int flag) {
    char *ptr;
    /* ... */
    unsigned int number = (unsigned int)ptr;
    number = (number & 0x7fffff) | (flag << 23);
    ptr = (char *)number;
}
```

A similar scheme was used in early versions of Emacs, limiting its portability and preventing the ability to edit files larger than 8MB.

Compliant Solution

This compliant solution uses a `struct` to provide storage for both the pointer and the flag value. This solution is portable to machines of different word sizes, both smaller and larger than 32 bits, working even when pointers cannot be represented in any integer type.

```
struct ptrflag {
    char *pointer;
    unsigned int flag : 9;
} ptrflag;

void func(unsigned int flag) {
    char *ptr;
    /* ... */
    ptrflag.pointer = ptr;
```

```
ptrflag.flag = flag;
}
```

Noncompliant Code Example

It is sometimes necessary to access memory at a specific location, requiring a literal integer to pointer conversion. In this noncompliant code, a pointer is set directly to an integer constant, where it is unknown whether the result will be as intended:

```
unsigned int *g(void) {
    unsigned int *ptr = 0xdeadbeef;
    /* ... */
    return ptr;
}
```

The result of this assignment is [implementation-defined](#), might not be correctly aligned, might not point to an entity of the referenced type, and might be a [trap representation](#).

Compliant Solution

Adding an explicit cast may help the compiler convert the integer value into a valid pointer. A common technique is to assign the integer to a volatile-qualified object of type `intptr_t` or `uintptr_t` and then assign the integer value to the pointer:

```
unsigned int *g(void) {
    volatile uintptr_t iptr = 0xdeadbeef;
    unsigned int *ptr = (unsigned int *)iptr;
    /* ... */
    return ptr;
}
```

Exceptions

INT36-C-EX1: A null pointer can be converted to an integer; it takes on the value 0. Likewise, the integer value 0 can be converted to a pointer; it becomes the null pointer.

INT36-C-EX2: Any valid pointer to `void` can be converted to `intptr_t` or `uintptr_t` or their underlying types and back again with no change in value. Use of underlying types instead of `intptr_t` or `uintptr_t` is discouraged, however, because it limits portability.

```
#include <assert.h>
#include <stdint.h>

void h(void) {
    intptr_t i = (intptr_t)(void *)&i;
    uintptr_t j = (uintptr_t)(void *)&j;

    void *ip = (void *)i;
    void *jp = (void *)j;

    assert(ip == &i);
    assert(jp == &j);
}
```

Risk Assessment

Converting from pointer to integer or vice versa results in code that is not portable and may create unexpected pointers to invalid memory locations.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT36-C	Low	Probable	High	P2	L3

Related Guidelines

SEI CERT C++ Coding Standard	INT11-CPP. Take care when converting from pointer to integer or integer to pointer
ISO/IEC TR 24772:2013	Pointer Casting and Pointer Type Changes [HFC]
ISO/IEC TS 17961:2013	Converting a pointer to integer or integer to pointer [intptrconv]
MITRE CWE	CWE-466 , Return of Pointer Value Outside of Expected Range CWE-587 , Assignment of a Fixed Address to a Pointer

Bibliography

[ISO/IEC 9899:2011]	6.3.2.3, "Pointers"
-------------------------------------	---------------------

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/XAAV>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high
reported_messages	If provided, only messages of these types are reported.	[767, 152, 1053]
reported_severities	List of severities to display.	('error', 'warning', 'remark')
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

CertC++-INT50

Do not cast to an out-of-range enumeration value.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Enumerations in C++ come in two forms: *scoped* enumerations in which the underlying type is fixed and *unscoped* enumerations in which the underlying type may or may not be fixed. The range of values that can be represented by either form of enumeration may include enumerator values not specified by the enumeration itself. The range of valid enumeration values for an enumeration type is defined by the C++ Standard, [dcl.enum], in paragraph 8 [[ISO/IEC 14882-2014](#)].

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.

The C++ Standard, [expr.static.cast], paragraph 10, states the following:

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 (7.2). 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 (4.9), and subsequently to the enumeration type.

To avoid operating on [unspecified values](#), the arithmetic value being cast must be within the range of values the enumeration can represent. When dynamically checking for out-of-range values, checking must be performed before the cast expression.

Noncompliant Code Example (Bounds Checking)

This noncompliant code example attempts to check whether a given value is within the range of acceptable enumeration values. However, it is doing so after casting to the enumeration type, which may not be able to represent the given integer value. On a two's complement system, the valid range of values that can be represented by `EnumType` are [0..3], so if a value outside of that range were passed to `f()`, the cast to `EnumType` would result in an unspecified value, and using that value within the `if` statement results in [unspecified behavior](#).

```
enum EnumType {
    First,
    Second,
    Third
};

void f(int intVar) {
    EnumType enumVar = static_cast<EnumType>(intVar);

    if (enumVar < First || enumVar > Third) {
        // Handle error
    }
}
```

Compliant Solution (Bounds Checking)

This compliant solution checks that the value can be represented by the enumeration type before performing the conversion to guarantee the conversion does not result in an unspecified value. It does this by restricting the converted value to one for which there is a specific enumerator value.

```
enum EnumType {
    First,
    Second,
    Third
};

void f(int intVar) {
    if (intVar < First || intVar > Third) {
        // Handle error
    }
    EnumType enumVar = static_cast<EnumType>(intVar);
}
```

Compliant Solution {Scoped Enumeration}

This compliant solution uses a scoped enumeration, which has a fixed underlying `int` type by default, allowing any value from the parameter to be converted into a valid enumeration value. It does not restrict the converted value to one for which there is a specific enumerator value, but it could do so as shown in the previous compliant solution.

```
enum class EnumType {
    First,
    Second,
    Third
};

void f(int intVar) {
    EnumType enumVar = static_cast<EnumType>(intVar);
}
```

Compliant Solution {Fixed Unscoped Enumeration}

Similar to the previous compliant solution, this compliant solution uses an unscoped enumeration but provides a fixed underlying `int` type allowing any value from the parameter to be converted to a valid enumeration value.

```
enum EnumType : int {
    First,
    Second,
    Third
};

void f(int intVar) {
    EnumType enumVar = static_cast<EnumType>(intVar);
}
```

Although similar to the previous compliant solution, this compliant solution differs from the noncompliant code example in the way the enumerator values are expressed in code and which implicit conversions are allowed. The previous compliant solution requires a nested name specifier to identify the enumerator (for example, `EnumType::First`) and will not implicitly convert the enumerator value to `int`. As with the noncompliant code example, this compliant solution does not allow a nested name specifier and will implicitly convert the enumerator value to `int`.

Risk Assessment

It is possible for unspecified values to result in a buffer overflow, leading to the execution of arbitrary code by an attacker. However, because enumerators are rarely used for indexing into arrays or other forms of pointer arithmetic, it is more likely that this scenario will result in data integrity violations rather than arbitrary code execution.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT50-CPP	Medium	Unlikely	Medium	P4	L3

Bibliography

[Becker 2009]	Section 7.2, "Enumeration Declarations"
[ISO/IEC 14882-2014]	Subclause 5.2.9, "Static Cast" Subclause 7.2, "Enumeration Declarations"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/INT50-CPP.+Do+not+cast+to+an+out-of-range+enumeration+value>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
asgn_not_an_enumerator	Assignment of value which does not correspond to an enumerator
asgn_out_of_enum_range	Assignment of value outside the range of the enumerators
conversion_creating_bad_enum_value	Expression does not correspond to an enumerator in {}

CertC++-ARR30

Do not form or use out-of-bounds pointers or array subscripts.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

The C Standard identifies the following distinct situations in which undefined behavior (UB) can arise as a result of invalid pointer operations:

UB	Description	Example Code
46	Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object.	Forming Out-of-Bounds Pointer, Null Pointer Arithmetic
47	Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated.	Dereferencing Past the End Pointer, Using Past the End Index
49	An array subscript is out of range, even if an object is apparently accessible with the given subscript, for example, in the lvalue expression a[1][7] given the declaration int a[4][5].	Apparently Accessible Out-of-Range Index
62	An attempt is made to access, or generate a pointer to just past, a flexible array member of a structure when the referenced object provides no elements for that array.	Pointer Past Flexible Array Member

Noncompliant Code Example (Forming Out-of-Bounds Pointer)

In this noncompliant code example, the function `f()` attempts to validate the `index` before using it as an offset to the statically allocated `table` of integers. However, the function fails to reject negative `index` values. When `index` is less than zero, the behavior of the addition expression in the return statement of the function is [undefined behavior 46](#). On some implementations, the addition alone can trigger a hardware trap. On other implementations, the addition may produce a result that when dereferenced triggers a hardware trap. Other implementations still may produce a dereferenceable pointer that points to an object distinct from `table`. Using such a pointer to access the object may lead to information exposure or cause the wrong object to be modified.

```
enum { TABLESIZE = 100 };

static int table[TABLESIZE];

int *f(int index) {
    if (index < TABLESIZE) {
        return table + index;
    }
    return NULL;
}
```

Compliant Solution

One compliant solution is to detect and reject invalid values of `index` if using them in pointer arithmetic would result in an invalid pointer:

```
enum { TABLESIZE = 100 };

static int table[TABLESIZE];

int *f(int index) {
    if (index >= 0 && index < TABLESIZE) {
```

```

        return table + index;
    }
    return NULL;
}

```

Compliant Solution

Another slightly simpler and potentially more efficient compliant solution is to use an unsigned type to avoid having to check for negative values while still rejecting out-of-bounds positive values of `index`:

```

#include <stddef.h>

enum { TABLESIZE = 100 };

static int table[TABLESIZE];

int *f(size_t index) {
    if (index < TABLESIZE) {
        return table + index;
    }
    return NULL;
}

```

Noncompliant Code Example (Dereferencing Past-the-End Pointer)

This noncompliant code example shows the flawed logic in the Windows Distributed Component Object Model (DCOM) Remote Procedure Call (RPC) interface that was exploited by the W32.Blaster.Worm. The error is that the `while` loop in the `GetMachineName()` function (used to extract the host name from a longer string) is not sufficiently bounded. When the character array pointed to by `pwszTemp` does not contain the backslash character among the first `MAX_COMPUTERNAME_LENGTH_FQDN + 1` elements, the final valid iteration of the loop will dereference past the end pointer, resulting in exploitable [undefined behavior 47](#). In this case, the actual exploit allowed the attacker to inject executable code into a running program. Economic damage from the Blaster worm has been estimated to be at least \$525 million [[Pethia 2003](#)].

For a discussion of this programming error in the Common Weakness Enumeration database, see [CWE-119](#), "Improper Restriction of Operations within the Bounds of a Memory Buffer," and [CWE-121](#), "Stack-based Buffer Overflow" [[MITRE 2013](#)].

```

error_status_t _RemoteActivation(
    /* ... */, WCHAR *pwszObjectName, ... ) {
    *phr = GetServerPath(
        pwszObjectName, &pwszObjectName);
    /* ... */
}

HRESULT GetServerPath(
    WCHAR *pwszPath, WCHAR **pwszServerPath ){
    WCHAR *pwszFinalPath = pwszPath;
    WCHAR wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN+1];
    hr = GetMachineName(pwszPath, wszMachineName);
    *pwszServerPath = pwszFinalPath;
}

HRESULT GetMachineName(
    WCHAR *pwszPath,
    WCHAR wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN+1])
{
    wszMachineName = wszMachineName;
    LPWSTR pwszTemp = pwszPath + 2;
    while (*pwszTemp != L'\\')
        *pwszServerName++ = *pwszTemp++;
    /* ... */
}

```

Compliant Solution

In this compliant solution, the `while` loop in the `GetMachineName()` function is bounded so that the loop terminates when a backslash character is found, the null-termination character (`L'\0'`) is discovered, or the end of the buffer is reached. This code does not result in a buffer overflow even if no backslash character is found in `wszMachineName`.

```

HRESULT GetMachineName(
    wchar_t *pwszPath,
    wchar_t wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN+1])
{
    wchar_t *pwszServerName = wszMachineName;
    wchar_t *pwszTemp = pwszPath + 2;
    wchar_t *end_addr
        = pwszServerName + MAX_COMPUTERNAME_LENGTH_FQDN;
    while ( (*pwszTemp != L'\\')
        && ((*pwszTemp != L'\0'))
        && (pwszServerName < end_addr) )
    {
        *pwszServerName++ = *pwszTemp++;
    }
    /* ... */
}

```

This compliant solution is for illustrative purposes and is not necessarily the solution implemented by Microsoft. This particular solution may not be correct because there is no guarantee that a backslash is found.

Noncompliant Code Example (Using Past-the-End Index)

Similar to the [dereferencing-past-the-end-pointer](#) error, the function `insert_in_table()` in this noncompliant code example uses an otherwise valid index to attempt to store a value in an element just past the end of an array.

First, the function incorrectly validates the index `pos` against the size of the buffer. When `pos` is initially equal to `size`, the function attempts to store `value` in a memory location just past the end of the buffer.

Second, when the index is greater than `size`, the function modifies `size` before growing the size of the buffer. If the call to `realloc()` fails to increase the size of the buffer, the next call to the function with a value of `pos` equal to or greater than the original value of `size` will again attempt to store `value` in a memory location just past the end of the buffer or beyond.

Third, the function violates [INT30-C. Ensure that unsigned integer operations do not wrap](#), which could lead to wrapping when 1 is added to `pos` or when `size` is multiplied by the size of `int`.

For a discussion of this programming error in the Common Weakness Enumeration database, see [CWE-122](#), "Heap-based Buffer Overflow," and [CWE-129](#), "Improper Validation of Array Index" [MITRE 2013].

```
#include <stdlib.h>

static int *table = NULL;
static size_t size = 0;

int insert_in_table(size_t pos, int value) {
    if (size < pos) {
        int *tmp;
        size = pos + 1;
        tmp = (int *)realloc(table, sizeof(*table) * size);
        if (tmp == NULL) {
            return -1; /* Failure */
        }
        table = tmp;
    }

    table[pos] = value;
    return 0;
}
```

Compliant Solution

This compliant solution correctly validates the index `pos` by using the `<=` relational operator, ensures the multiplication will not overflow, and avoids modifying `size` until it has verified that the call to `realloc()` was successful:

```
#include <stdint.h>
#include <stdlib.h>

static int *table = NULL;
static size_t size = 0;

int insert_in_table(size_t pos, int value) {
    if (size <= pos) {
        if ((SIZE_MAX - 1 < pos) ||
            ((pos + 1) > SIZE_MAX / sizeof(*table))) {
            return -1;
        }

        int *tmp = (int *)realloc(table, sizeof(*table) * (pos + 1));
        if (tmp == NULL) {
            return -1;
        }
        /* Modify size only after realloc() succeeds */
        size = pos + 1;
        table = tmp;
    }

    table[pos] = value;
    return 0;
}
```

Noncompliant Code Example (Apparently Accessible Out-of-Range Index)

This noncompliant code example declares `matrix` to consist of 7 rows and 5 columns in row-major order. The function `init_matrix` iterates over all 35 elements in an attempt to initialize each to the value given by the function argument `x`. However, because multidimensional arrays are declared in C in row-major order, the function iterates over the elements in column-major order, and when the value of `j` reaches the value `COLS` during the first iteration of the outer loop, the function attempts to access element `matrix[0][5]`. Because the type of `matrix` is `int[7][5]`, the `j` subscript is out of range, and the access has [undefined behavior 49](#).

```
#include <stddef.h>
#define COLS 5
#define ROWS 7
static int matrix[ROWS][COLS];

void init_matrix(int x) {
    for (size_t i = 0; i < COLS; i++) {
        for (size_t j = 0; j < ROWS; j++) {
            matrix[i][j] = x;
        }
    }
}
```

Compliant Solution

This compliant solution avoids using out-of-range indices by initializing `matrix` elements in the same row-major order as multidimensional objects are declared in C:

```
#include <stddef.h>
```

```
#define COLS 5
#define ROWS 7
static int matrix[ROWS][COLS];

void init_matrix(int x) {
    for (size_t i = 0; i < ROWS; i++) {
        for (size_t j = 0; j < COLS; j++) {
            matrix[i][j] = x;
        }
    }
}
```

Noncompliant Code Example [Pointer Past Flexible Array Member]

In this noncompliant code example, the function `find()` attempts to iterate over the elements of the flexible array member `buf`, starting with the second element. However, because function `g()` does not allocate any storage for the member, the expression `first++` in `find()` attempts to form a pointer just past the end of `buf` when there are no elements. This attempt is [undefined behavior 62](#). (See [MSC21-C. Use robust loop termination conditions](#) for more information.)

```
#include <stdlib.h>

struct S {
    size_t len;
    char buf[]; /* Flexible array member */
};

const char *find(const struct S *s, int c) {
    const char *first = s->buf;
    const char *last = s->buf + s->len;

    while (first++ != last) { /* Undefined behavior */
        if (*first == (unsigned char)c) {
            return first;
        }
    }
    return NULL;
}

void g(void) {
    struct S *s = (struct S *)malloc(sizeof(struct S));
    if (s == NULL) {
        /* Handle error */
    }
    s->len = 0;
    find(s, 'a');
}
```

Compliant Solution

This compliant solution avoids incrementing the pointer unless a value past the pointer's current value is known to exist:

```
#include <stdlib.h>

struct S {
    size_t len;
    char buf[]; /* Flexible array member */
};

const char *find(const struct S *s, int c) {
    const char *first = s->buf;
    const char *last = s->buf + s->len;

    while (first != last) { /* Avoid incrementing here */
        if (++first == (unsigned char)c) {
            return first;
        }
    }
    return NULL;
}

void g(void) {
    struct S *s = (struct S *)malloc(sizeof(struct S));
    if (s == NULL) {
        /* Handle error */
    }
    s->len = 0;
    find(s, 'a');
}
```

Noncompliant Code Example [Null Pointer Arithmetic]

This noncompliant code example is similar to an [Adobe Flash Player vulnerability](#) that was first exploited in 2008. This code allocates a block of memory and initializes it with some data. The data does not belong at the beginning of the block, which is left uninitialized. Instead, it is placed `offset` bytes within the block. The function ensures that the data fits within the allocated block.

```
#include <string.h>
#include <stdlib.h>

char *init_block(size_t block_size, size_t offset,
                 char *data, size_t data_size) {
    char *buffer = malloc(block_size);
    if (data_size > block_size || block_size - data_size < offset) {
        /* Data won't fit in buffer, handle error */
    }
    memncpy(buffer + offset, data, data_size);
}
```

```

    return buffer;
}

```

This function fails to check if the allocation succeeds, which is a violation of [ERR33-C. Detect and handle standard library errors](#). If the allocation fails, then `malloc()` returns a null pointer. The null pointer is added to `offset` and passed as the destination argument to `memcpy()`. Because a null pointer does not point to a valid object, the result of the pointer arithmetic is [undefined behavior 46](#).

An attacker who can supply the arguments to this function can exploit it to execute arbitrary code. This can be accomplished by providing an overly large value for `block_size`, which causes `malloc()` to fail and return a null pointer. The `offset` argument will then serve as the destination address to the call to `memcpy()`. The attacker can specify the `data` and `data_size` arguments to provide the address and length of the address, respectively, that the attacker wishes to write into the memory referenced by `offset`. The overall result is that the call to `memcpy()` can be exploited by an attacker to overwrite an arbitrary memory location with an attacker-supplied address, typically resulting in arbitrary code execution.

Compliant Solution (Null Pointer Arithmetic)

This compliant solution ensures that the call to `malloc()` succeeds:

```

#include <string.h>
#include <stdlib.h>

char *init_block(size_t block_size, size_t offset,
                 char *data, size_t data_size) {
    char *buffer = malloc(block_size);
    if (NULL == buffer) {
        /* Handle error */
    }
    if (data_size > block_size || block_size - data_size < offset) {
        /* Data won't fit in buffer, handle error */
    }
    memcpy(buffer + offset, data, data_size);
    return buffer;
}

```

Risk Assessment

Writing to out-of-range pointers or array subscripts can result in a buffer overflow and the execution of arbitrary code with the permissions of the vulnerable process. Reading from out-of-range pointers or array subscripts can result in unintended information disclosure.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ARR30-C	High	Likely	High	P9	L2

Related Guidelines

ISO/IEC TR 24772:2013	Arithmetic Wrap-Around Error [FIF] Unchecked Array Indexing [XYZ]
ISO/IEC TS 17961	Forming or using out-of-bounds pointers or array subscripts [invptr]
MITRE CWE	CWE-119 , Improper Restriction of Operations within the Bounds of a Memory Buffer CWE-122 , Heap-based Buffer Overflow CWE-123 , Write-what-where Condition CWE-125 , Out-of-bounds Read CWE-129 , Improper Validation of Array Index CWE-788 , Access of Memory Location after End of Buffer
MISRA C:2012	Rule 18.1 (required)

Bibliography

[Finlay 2003]	
[Microsoft 2003]	
[Pethia 2003]	
[Seacord 2013b]	Chapter 1, "Running with Scissors"
[Viega 2005]	Section 5.2.13, "Unchecked Array Indexing"
[xort 2009]	" CVE-2008-1517: Apple Mac OS X (XNU) Missing Array Index Validation "

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/DYDXAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	9
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
arithmetic_out_of_bounds	Pointer arithmetic on {1} creates pointer outside array bounds of {0}
out_of_bounds	Access into array is out of bounds
possible_indirect_out_of_bounds	Pointer-indirect access through {1} might be out of bounds accessing {0}
possible_out_of_bounds	Access into array might be out of bounds

CertC++-ARR37

Do not add or subtract an integer to a pointer to a non-array object.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Pointer arithmetic must be performed only on pointers that reference elements of array objects.

The C Standard, 6.5.6 [[ISO/IEC 9899:2011](#)], states the following about pointer arithmetic:

When an expression that has integer 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 integer expression.

Noncompliant Code Example

This noncompliant code example attempts to access structure members using pointer arithmetic. This practice is dangerous because structure members are not guaranteed to be contiguous.

```
struct numbers {
    short num_a, num_b, num_c;
};

int sum_numbers(const struct numbers *numb) {
    int total = 0;
    const short *numb_ptr;

    for (numb_ptr = &numb->num_a;
         numb_ptr <= &numb->num_c;
         numb_ptr++) {
        total += *(numb_ptr);
    }

    return total;
}

int main(void) {
    struct numbers my_numbers = { 1, 2, 3 };
    sum_numbers(&my_numbers);
    return 0;
}
```

Compliant Solution

It is possible to use the `->` operator to dereference each structure member:

```
total = numb->num_a + numb->num_b + numb->num_c;
```

However, this solution results in code that is hard to write and hard to maintain (especially if there are many more structure members), which is exactly what the author of the noncompliant code example was likely trying to avoid.

Compliant Solution

A better solution is to define the structure to contain an array member to store the numbers in an array rather than a structure, as in this compliant solution:

```
#include <stddef.h>

struct numbers {
    short a[3];
};

int sum_numbers(const short *numb, size_t dim) {
    int total = 0;
    for (size_t i = 0; i < dim; ++i) {
        total += numb[i];
    }
    return total;
}

int main(void) {
    struct numbers my_numbers = { .a[0]= 1, .a[1]= 2, .a[2]= 3 };
    sum_numbers(
        my_numbers.a,
        sizeof(my_numbers.a)/sizeof(my_numbers.a[0])
    );
    return 0;
}
```

Array elements are guaranteed to be contiguous in memory, so this solution is completely portable.

Exceptions

ARR37-C-EX1: Any non-array object in memory can be considered an array consisting of one element. Adding one to a pointer for such an object yields a pointer one element past the array, and subtracting one from that pointer yields the original pointer. This allows for code such as the following:

```
#include <stdlib.h>
#include <string.h>

struct s {
    char *c_str;
    /* Other members */
};

struct s *create_s(const char *c_str) {
    struct s *ret;
    size_t len = strlen(c_str) + 1;

    ret = (struct s *)malloc(sizeof(struct s) + len);
    if (ret != NULL) {
        ret->c_str = (char *)(ret + 1);
        memcpy(ret + 1, c_str, len);
    }
    return ret;
}
```

A more general and safer solution to this problem is to use a flexible array member that guarantees the array that follows the structure is properly aligned by inserting padding, if necessary, between it and the member that immediately precedes it.

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ARR37-C	Medium	Probable	Medium	P8	L2

Related Guidelines

[MITRE CWE](#) [CWE-469](#), Use of Pointer Subtraction to Determine Size

Bibliography

[Banahan 2003]	Section 5.3, "Pointers" Section 5.7, "Expressions Involving Pointers"
[ISO/IEC 9899:2011]	6.5.6, "Additive Operators"
[VU#162289]	

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/UgHm>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
arithmetic_on_nonarray	Pointer arithmetic on {1} operates on non-array target {0}

CertC++-ARR39

Do not add or subtract a scaled integer to a pointer.

Input: IR

Source languages: C++

Details

Pointer arithmetic is appropriate only when the pointer argument refers to an array (see [ARR37-C. Do not add or subtract an integer to a pointer to a non-array object](#)), including an array of bytes. When performing pointer arithmetic, the size of the value to add to or subtract from a pointer is automatically scaled to the size of the type of the referenced array object. Adding or subtracting a scaled integer value to or from a pointer is invalid because it may yield a pointer that does not point to an element within or one past the end of the array. (See [ARR30-C. Do not form or use out-of-bounds pointers or array subscripts](#).)

Adding a pointer to an array of a type other than character to the result of the `sizeof` operator or `offsetof` macro, which returns a size and an offset, respectively, violates this rule. However, adding an array pointer to the number of array elements, for example, by using the `arr[sizeof(arr)/sizeof(arr[0])]` idiom, is allowed provided that `arr` refers to an array and not a pointer.

Noncompliant Code Example

In this noncompliant code example, `sizeof(buf)` is added to the array `buf`. This example is noncompliant because `sizeof(buf)` is scaled by `int` and then scaled again when added to `buf`.

```
enum { INTBUFSIZE = 80 };

extern int getdata(void);
int buf[INTBUFSIZE];

void func(void) {
    int *buf_ptr = buf;

    while (buf_ptr < (buf + sizeof(buf))) {
        *buf_ptr++ = getdata();
    }
}
```

Compliant Solution

This compliant solution uses an unscaled integer to obtain a pointer to the end of the array:

```
enum { INTBUFSIZE = 80 };

extern int getdata(void);
int buf[INTBUFSIZE];

void func(void) {
    int *buf_ptr = buf;

    while (buf_ptr < (buf + INTBUFSIZE)) {
        *buf_ptr++ = getdata();
    }
}
```

Noncompliant Code Example

In this noncompliant code example, `skip` is added to the pointer `s`. However, `skip` represents the byte offset of `ull_b` in `struct big`. When added to `s`, `skip` is scaled by the size of `struct big`.

```
#include <string.h>
#include <stdlib.h>
#include <stddef.h>

struct big {
    unsigned long long ull_a;
```

```

unsigned long long ull_b;
unsigned long long ull_c;
int si_e;
int si_f;
};

void func(void) {
    size_t skip = offsetof(struct big, ull_b);
    struct big *s = (struct big *)malloc(sizeof(struct big));
    if (s == NULL) {
        /* Handle malloc() error */
    }

    memset(s + skip, 0, sizeof(struct big) - skip);
    /* ... */
    free(s);
    s = NULL;
}

```

Compliant Solution

This compliant solution uses an `unsigned char *` to calculate the offset instead of using a `struct big *`, which would result in scaled arithmetic:

```

#include <string.h>
#include <stdlib.h>
#include <stddef.h>

struct big {
    unsigned long long ull_a;
    unsigned long long ull_b;
    unsigned long long ull_c;
    int si_d;
    int si_e;
};

void func(void) {
    size_t skip = offsetof(struct big, ull_b);
    unsigned char *ptr = (unsigned char *)malloc(
        sizeof(struct big)
    );
    if (ptr == NULL) {
        /* Handle malloc() error */
    }

    memset(ptr + skip, 0, sizeof(struct big) - skip);
    /* ... */
    free(ptr);
    ptr = NULL;
}

```

Noncompliant Code Example

In this noncompliant code example, `wcslen(error_msg) * sizeof(wchar_t)` bytes are scaled by the size of `wchar_t` when added to `error_msg`:

```

#include <wchar.h>
#include <stdio.h>

enum { WCHAR_BUF = 128 };

void func(void) {
    wchar_t error_msg[WCHAR_BUF];

    wcscpy(error_msg, L"Error: ");
    fgetws(error_msg + wcslen(error_msg) * sizeof(wchar_t),
           WCHAR_BUF - 7, stdin);
    /* ... */
}

```

Compliant Solution

This compliant solution does not scale the length of the string; `wcslen()` returns the number of characters and the addition to `error_msg` is scaled:

```

#include <wchar.h>
#include <stdio.h>

enum { WCHAR_BUF = 128 };
const wchar_t ERROR_PREFIX[7] = L"Error: ";

void func(void) {
    const size_t prefix_len = wcslen(ERROR_PREFIX);
    wchar_t error_msg[WCHAR_BUF];

    wcscpy(error_msg, ERROR_PREFIX);
    fgetws(error_msg + prefix_len,
           WCHAR_BUF - prefix_len, stdin);
    /* ... */
}

```

Risk Assessment

Failure to understand and properly use pointer arithmetic can allow an attacker to execute arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ARR39-C	High	Probable	High	P6	L2

Related Guidelines

CERT C Secure Coding Standard	ARR30-C. Do not form or use out-of-bounds pointers or array subscripts ARR37-C. Do not add or subtract an integer to a pointer to a non-array object
ISO/IEC TR 24772:2013	Pointer Casting and Pointer Type Changes [HFC] Pointer Arithmetic [RVG]
MISRA C:2012	Rule 18.1 (required) Rule 18.2 (required) Rule 18.3 (required) Rule 18.4 (advisory)
MITRE CWE	CWE-468 , Incorrect Pointer Scaling

Bibliography

[Dowd 2006]	Chapter 6, "C Language Issues"
[Murenin 07]	

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/HADXAQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
scaled_arith	Do not add or subtract a scaled integer to a pointer.

CertC++-CTR56

Do not use pointer arithmetic on polymorphic objects.

Input: IR
Source languages: C++

Details

The definition of *pointer arithmetic* from the C++ Standard, [expr.add], paragraph 7 [[ISO/IEC 14882-2014](#)], states the following:

For addition or subtraction, if the expressions \mathbf{P} or \mathbf{Q} have type "pointer to $\mathbf{cv}\mathbf{T}$ ", where \mathbf{T} is different from the \mathbf{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]

Pointer arithmetic does not account for polymorphic object sizes, and attempting to perform pointer arithmetic on a polymorphic object value results in [undefined behavior](#).

The C++ Standard, [expr.sub], paragraph 1 [[ISO/IEC 14882-2014](#)], defines array subscripting as being identical to pointer arithmetic. Specifically, it states the following:

The expression $\mathbf{E1}[\mathbf{E2}]$ is identical (by definition) to $\ast((\mathbf{E1}) + (\mathbf{E2}))$.

Do not use pointer arithmetic, including array subscripting, on polymorphic objects.

The following code examples assume the following static variables and class definitions.

```

int globI;
double globD;

struct S {
    int i;
    S() : i(globI++) {}
};

struct T : S {
    double d;
    T() : S(), d(globD++) {}
};

```

Noncompliant Code Example [Pointer Arithmetic]

In this noncompliant code example, `f()` accepts an array of `S` objects as its first parameter. However, `main()` passes an array of `T` objects as the first argument to `f()`, which results in [undefined behavior](#) due to the pointer arithmetic used within the `for` loop.

```

#include <iostream>

// ... definitions for S, T, globI, globD ...

void f(const S *someSes, std::size_t count) {
    for (const S *end = someSes + count; someSes != end; ++someSes) {
        std::cout << someSes->i << std::endl;
    }
}

int main() {
    T test[5];
    f(test, 5);
}

```

Noncompliant Code Example [Array Subscripting]

In this noncompliant code example, the `for` loop uses array subscripting. Since array subscripts are computed using pointer arithmetic, this code also results in undefined behavior.

```

#include <iostream>

// ... definitions for S, T, globI, globD ...

void f(const S *someSes, std::size_t count) {
    for (std::size_t i = 0; i < count; ++i) {
        std::cout << someSes[i].i << std::endl;
    }
}

int main() {
    T test[5];
    f(test, 5);
}

```

Compliant Solution [Array]

Instead of having an array of objects, an array of pointers solves the problem of the objects being of different sizes, as in this compliant solution.

```

#include <iostream>

// ... definitions for S, T, globI, globD ...

void f(const S * const *someSes, std::size_t count) {
    for (const S * const *end = someSes + count; someSes != end; ++someSes) {
        std::cout << (*someSes)->i << std::endl;
    }
}

int main() {
    S *test[] = {new T, new T, new T, new T, new T};
    f(test, 5);
    for (auto v : test) {
        delete v;
    }
}

```

The elements in the arrays are no longer polymorphic objects (instead, they are pointers to polymorphic objects), and so there is no [undefined behavior](#) with the pointer arithmetic.

Compliant Solution [`std::vector`]

Another approach is to use a standard template library (STL) container instead of an array and have `f()` accept iterators as parameters, as in this compliant solution. However, because STL containers require homogeneous elements, pointers are still required within the container.

```

#include <iostream>
#include <vector>

// ... definitions for S, T, globI, globD ...

template <typename Iter>
void f(Iter i, Iter e) {
    for (; i != e; ++i) {
        std::cout << (*i)->i << std::endl;
    }
}

```

```

}
int main() {
    std::vector<S *> test{new T, new T, new T, new T, new T};
    f(test.cbegin(), test.cend());
    for (auto v : test) {
        delete v;
    }
}

```

Risk Assessment

Using arrays polymorphically can result in memory corruption, which could lead to an attacker being able to execute arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CTR56-CPP	High	Likely	High	P9	L2

Related Guidelines

SEI CERT C Coding Standard	ARR39-C. Do not add or subtract a scaled integer to a pointer
--	---

Bibliography

[ISO/IEC 14882-2014]	Subclause 5.7, "Additive Operators" Subclause 5.2.1, "Subscripting"
[Lockheed Martin 2005]	AV Rule 96, "Arrays shall not be treated polymorphically"
[Meyers 1996]	Item 3, "Never Treat Arrays Polymorphically"
[Stroustrup 2006]	"What's Wrong with Arrays?"
[Sutter 2004]	Item 100, "Don't Treat Arrays Polymorphically"

Excerpt from SEI CERT C++ Coding Standard Wiki [https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR56-
CPP.+Do+not+use+pointer+arithmetic+on+polymorphic+objects](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR56-CPP.+Do+not+use+pointer+arithmetic+on+polymorphic+objects), Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
consider_non_base_final	Whether to consider a class that is not used as a base class as a sort-of final class.	False
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	9
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
non_final_pointer_arithmetic	Pointer arithmetic on non-final classes not allowed

CertC++-STR30

Do not attempt to modify string literals.

Input: IR

Source languages: C++

Details

According to the C Standard, 6.4.5, paragraph 3 [[ISO/IEC 9899:2011](#)]:

A *character string literal* is a sequence of zero or more multibyte characters enclosed in double-quotes, as in "xyz". A UTF-8 string literal is the same, except prefixed by u8. A wide string literal is the same, except prefixed by the letter L, u, or U.

At compile time, string literals are used to create an array of static storage duration of sufficient length to contain the character sequence and a terminating null character. String literals are usually referred to by a pointer to (or array of) characters. Ideally, they should be assigned only to pointers to (or arrays of)

`const char` or `const wchar_t`. It is unspecified whether these arrays of string literals are distinct from each other. The behavior is [undefined](#) if a program attempts to modify any portion of a string literal. Modifying a string literal frequently results in an access violation because string literals are typically stored in read-only memory. (See [undefined behavior 33](#).)

Avoid assigning a string literal to a pointer to `non-const` or casting a string literal to a pointer to `non-const`. For the purposes of this rule, a pointer to (or array of) `const` characters must be treated as a string literal. Similarly, the returned value of the following library functions must be treated as a string literal if the first argument is a string literal:

- `strpbrk()`, `strchr()`, `strrchr()`, `strstr()`
- `wcspbrk()`, `wcschr()`, `wcsrchr()`, `wcsstr()`
- `memchr()`, `wmemchr()`

This rule is a specific instance of [EXP40-C. Do not modify constant objects](#).

Noncompliant Code Example

In this noncompliant code example, the `char` pointer `p` is initialized to the address of a string literal. Attempting to modify the string literal is [undefined behavior](#):

```
char *p = "string literal";
p[0] = 'S';
```

Compliant Solution

As an array initializer, a string literal specifies the initial values of characters in an array as well as the size of the array. (See [STR11-C. Do not specify the bound of a character array initialized with a string literal](#).) This code creates a copy of the string literal in the space allocated to the character array `a`. The string stored in `a` can be modified safely.

```
char a[] = "string literal";
a[0] = 'S';
```

Noncompliant Code Example [POSIX]

In this noncompliant code example, a string literal is passed to the (`pointer to non-const`) parameter of the POSIX function [`mkstemp\(\)`](#), which then modifies the characters of the string literal:

```
#include <stdlib.h>

void func(void) {
    mkstemp("/tmp/edXXXXXX");
}
```

The behavior of `mkstemp()` is described in more detail in [FI021-C. Do not create temporary files in shared directories](#).

Compliant Solution [POSIX]

This compliant solution uses a named array instead of passing a string literal:

```
#include <stdlib.h>

void func(void) {
    static char fname[] = "/tmp/edXXXXXX";
    mkstemp(fname);
}
```

Noncompliant Code Example [Result of `strrchr()`]

In this noncompliant example, the `char *` result of the `strrchr()` function is used to modify the object pointed to by `pathname`. Because the argument to `strrchr()` points to a string literal, the effects of the modification are undefined.

```
#include <stdio.h>
#include <string.h>

const char *get dirname(const char *pathname) {
    char *slash;
    slash = strrchr(pathname, '/');
    if (slash)
        *slash = '\0'; /* Undefined behavior */
    return pathname;
}

int main(void) {
    puts(get dirname(__FILE__));
    return 0;
}
```

Compliant Solution [Result of `strrchr()`]

This compliant solution avoids modifying a `const` object, even if it is possible to obtain a `non-const` pointer to such an object by calling a standard C library function, such as `strrchr()`. To reduce the risk to callers of `get dirname()`, a buffer and length for the directory name are passed into the function. It is insufficient to change `pathname` to require a `char *` instead of a `const char *` because conforming compilers are not required to diagnose passing a string

literal to a function accepting a `char *`.

```
#include <stddef.h>
#include <stdio.h>
#include <string.h>

char *get dirname(const char *pathname, char *dirname, size_t size) {
    const char *slash;
    slash = strrchr(pathname, '/');
    if (slash) {
        ptrdiff_t slash_idx = slash - pathname;
        if ((size_t)slash_idx < size) {
            memcpy(dirname, pathname, slash_idx);
            dirname[slash_idx] = '\0';
            return dirname;
        }
    }
    return 0;
}

int main(void) {
    char dirname[260];
    if (get dirname(__FILE__, dirname, sizeof(dirname))) {
        puts(dirname);
    }
    return 0;
}
```

Risk Assessment

Modifying string literals can lead to [abnormal program termination](#) and possibly [denial-of-service attacks](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR30-C	Low	Likely	Low	P9	L2

Related Guidelines

CERT C Secure Coding Standard	EXP05-C. Do not cast away a const qualification STR11-C. Do not specify the bound of a character array initialized with a string literal
ISO/IEC TS 17961:2013	Modifying string literals [strmod]

Bibliography

[ISO/IEC 9899:2011]	6.4.5, "String Literals"
[Plum 1991]	Topic 1.26, "Strings-String Literals"
[Summit 1995]	comp.lang.c FAQ List, Question 1.32

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/TQE>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
funcs		('strpbrk', 'strchr', 'strrchr', 'strstr', 'wcspbrk', 'wcschr', 'wcsrchr', 'wcsstr', 'memchr', 'wmemchr')
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	9
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
call_with_const	Result of call to {}() with '{}' input should be used as 'const {}*'.
call_with_literal	Result of call to {}() with string literal should be used as 'const {}*'.
nonconst_string_literal	String literal should only be used as 'const char*'

CertC++-STR31

Guarantee that storage for strings has sufficient space for character data and the null terminator.

Input: IR

Source languages: C++

Details

Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. Buffer overflows occur frequently when manipulating strings [Seacord 2013b]. To prevent such errors, either limit copies through truncation or, preferably, ensure that the destination is of sufficient size to hold the character data to be copied and the null-termination character. (See [STR03-C. Do not inadvertently truncate a string](#).)

When strings live on the heap, this rule is a specific instance of [MEM35-C. Allocate sufficient memory for an object](#). Because strings are represented as arrays of characters, this rule is related to both [ARR30-C. Do not form or use out-of-bounds pointers or array subscripts](#) and [ARR38-C. Guarantee that library functions do not form invalid pointers](#).

Noncompliant Code Example {Off-by-One Error}

This noncompliant code example demonstrates an *off-by-one* error [Dowd 2006]. The loop copies data from `src` to `dest`. However, because the loop does not account for the null-termination character, it may be incorrectly written 1 byte past the end of `dest`.

```
#include <stddef.h>

void copy(size_t n, char src[n], char dest[n]) {
    size_t i;

    for (i = 0; src[i] && (i < n); ++i) {
        dest[i] = src[i];
    }
    dest[i] = '\0';
}
```

Compliant Solution {Off-by-One Error}

In this compliant solution, the loop termination condition is modified to account for the null-termination character that is appended to `dest`:

```
#include <stddef.h>

void copy(size_t n, char src[n], char dest[n]) {
    size_t i;

    for (i = 0; src[i] && (i < n - 1); ++i) {
        dest[i] = src[i];
    }
    dest[i] = '\0';
}
```

Noncompliant Code Example {`gets()`}

The `gets()` function, which was deprecated in the C99 Technical Corrigendum 3 and removed from C11, is inherently unsafe and should never be used because it provides no way to control how much data is read into a buffer from `stdin`. This noncompliant code example assumes that `gets()` will not read more than `BUFFER_SIZE - 1` characters from `stdin`. This is an invalid assumption, and the resulting operation can result in a buffer overflow.

The `gets()` function reads characters from the `stdin` into a destination array until end-of-file is encountered or a newline character is read. Any newline character is discarded, and a null character is written immediately after the last character read into the array.

```
#include <stdio.h>

#define BUFFER_SIZE 1024

void func(void) {
    char buf[BUFFER_SIZE];
    if (gets(buf) == NULL) {
        /* Handle error */
    }
}
```

See also [MSC24-C. Do not use deprecated or obsolescent functions](#).

Compliant Solution {`fgets()`}

The `fgets()` function reads, at most, one less than the specified number of characters from a stream into an array. This solution is compliant because the number of characters copied from `stdin` to `buf` cannot exceed the allocated memory:

```
#include <stdio.h>
#include <string.h>

enum { BUFSIZE = 32 };

void func(void) {
    char buf[BUFSIZE];
    int ch;

    if (fgets(buf, sizeof(buf), stdin)) {
        /* fgets() succeeded; scan for newline character */
        char *p = strchr(buf, '\n');
        if (p) {
            *p = '\0';
        } else {
            /* Newline not found; flush stdin to end of line */
            while ((ch = getchar()) != '\n' && ch != EOF)
                ;
            if (ch == EOF && !feof(stdin) && !ferror(stdin)) {
                /* Character resembles EOF; handle error */
            }
        }
    } else {
        /* fgets() failed; handle error */
    }
}
```

The `fgets()` function is not a strict replacement for the `gets()` function because `fgets()` retains the newline character (if read) and may also return a partial line. It is possible to use `fgets()` to safely process input lines too long to store in the destination array, but this is not recommended for performance reasons. Consider using one of the following compliant solutions when replacing `gets()`.

Compliant Solution (`gets_s()`)

The `gets_s()` function reads, at most, one less than the number of characters specified from the stream pointed to by `stdin` into an array.

The C Standard, Annex K [[ISO/IEC 9899:2011](#)], states

No additional characters are read after a new-line character [which is discarded] or after end-of-file. The discarded new-line character does not count towards number of characters read. A null character is written immediately after the last character read into the array.

If end-of-file is encountered and no characters have been read into the destination array, or if a read error occurs during the operation, then the first character in the destination array is set to the null character and the other elements of the array take unspecified values:

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>

enum { BUFSIZE = 32 };

void func(void) {
    char buf[BUFSIZE];

    if (gets_s(buf, sizeof(buf)) == NULL) {
        /* Handle error */
    }
}
```

Compliant Solution (`getline()`, POSIX)

The `getline()` function is similar to the `fgets()` function but can dynamically allocate memory for the input buffer. If passed a null pointer, `getline()` dynamically allocates a buffer of sufficient size to hold the input. If passed a pointer to dynamically allocated storage that is too small to hold the contents of the string, the `getline()` function resizes the buffer, using `realloc()`, rather than truncating the input. If successful, the `getline()` function returns the number of characters read, which can be used to determine if the input has any null characters before the newline. The `getline()` function works only with dynamically allocated buffers. Allocated memory must be explicitly deallocated by the caller to avoid memory leaks. (See [MEM31-C. Free dynamically allocated memory when no longer needed](#).)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void func(void) {
    int ch;
    size_t buffer_size = 32;
    char *buffer = malloc(buffer_size);

    if (!buffer) {
        /* Handle error */
        return;
    }

    if ((ssize_t size = getline(&buffer, &buffer_size, stdin))
        == -1) {
        /* Handle error */
    } else {
        char *p = strchr(buffer, '\n');
        if (p) {
            *p = '\0';
        } else {
            /* Newline not found; flush stdin to end of line */
            while ((ch = getchar()) != '\n' && ch != EOF)
                ;
            if (ch == EOF && !feof(stdin) && !ferror(stdin)) {
                /* Character resembles EOF; handle error */
            }
        }
    }
}
```

```

        }
    }
    free (buffer);
}

```

Note that the `getline()` function uses an [in-band error indicator](#), in violation of [ERR02-C. Avoid in-band error indicators](#).

Noncompliant Code Example (`getchar()`)

Reading one character at a time provides more flexibility in controlling behavior, though with additional performance overhead. This noncompliant code example uses the `getchar()` function to read one character at a time from `stdin` instead of reading the entire line at once. The `stdin` stream is read until end-of-file is encountered or a newline character is read. Any newline character is discarded, and a null character is written immediately after the last character read into the array. Similar to the noncompliant code example that invokes `gets()`, there are no guarantees that this code will not result in a buffer overflow.

```
#include <stdio.h>

enum { BUFSIZE = 32 };

void func(void) {
    char buf[BUFSIZE];
    char *p;
    int ch;
    p = buf;
    while ((ch = getchar()) != '\n' && ch != EOF) {
        *p++ = (char)ch;
    }
    *p++ = 0;
    if (ch == EOF) {
        /* Handle EOF or error */
    }
}
```

After the loop ends, if `ch == EOF`, the loop has read through to the end of the stream without encountering a newline character, or a read error occurred before the loop encountered a newline character. To conform to [FI034-C. Distinguish between characters read from a file and EOF or WEOF](#), the error-handling code must verify that an end-of-file or error has occurred by calling `feof()` or `ferror()`.

Compliant Solution (`getchar()`)

In this compliant solution, characters are no longer copied to `buf` once `index == BUFSIZE - 1`, leaving room to null-terminate the string. The loop continues to read characters until the end of the line, the end of the file, or an error is encountered. When `chars_read > index`, the input string has been truncated.

```
#include <stdio.h>

enum { BUFSIZE = 32 };

void func(void) {
    char buf[BUFSIZE];
    int ch;
    size_t index = 0;
    size_t chars_read = 0;

    while ((ch = getchar()) != '\n' && ch != EOF) {
        if (index < sizeof(buf) - 1) {
            buf[index++] = (char)ch;
        }
        chars_read++;
    }
    buf[index] = '\0'; /* Terminate string */
    if (ch == EOF) {
        /* Handle EOF or error */
    }
    if (chars_read > index) {
        /* Handle truncation */
    }
}
```

Noncompliant Code Example (`fscanf()`)

In this noncompliant example, the call to `fscanf()` can result in a write outside the character array `buf`:

```
#include <stdio.h>

enum { BUF_LENGTH = 1024 };

void get_data(void) {
    char buf[BUF_LENGTH];
    if (1 != fscanf(stdin, "%s", buf)) {
        /* Handle error */
    }
    /* Rest of function */
}
```

Compliant Solution (`fscanf()`)

In this compliant solution, the call to `fscanf()` is constrained not to overflow `buf`:

```
#include <stdio.h>

enum { BUF_LENGTH = 1024 };

void get_data(void) {
    char buf[BUF_LENGTH];
    if (1 != fscanf(stdin, "%1023s", buf)) {
        /* Handle error */
    }
    /* Rest of function */
}
```

Noncompliant Code Example {argv}

In a [hosted environment](#), arguments read from the command line are stored in process memory. The function `main()`, called at program startup, is typically declared as follows when the program accepts command-line arguments:

```
int main(int argc, char *argv[]) { /* ... */ }
```

Command-line arguments are passed to `main()` as pointers to strings in the array members `argv[0]` through `argv[argc - 1]`. If the value of `argc` is greater than 0, the string pointed to by `argv[0]` is, by convention, the program name. If the value of `argc` is greater than 1, the strings referenced by `argv[1]` through `argv[argc - 1]` are the program arguments.

[Vulnerabilities](#) can occur when inadequate space is allocated to copy a command-line argument or other program input. In this noncompliant code example, an attacker can manipulate the contents of `argv[0]` to cause a buffer overflow:

```
#include <string.h>

int main(int argc, char *argv[]) {
    /* Ensure argv[0] is not null */
    const char *const name = (argc && argv[0]) ? argv[0] : "";
    char prog_name[128];
    strcpy(prog_name, name);

    return 0;
}
```

Compliant Solution {argv}

The `strlen()` function can be used to determine the length of the strings referenced by `argv[0]` through `argv[argc - 1]` so that adequate memory can be dynamically allocated.

```
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    /* Ensure argv[0] is not null */
    const char *const name = (argc && argv[0]) ? argv[0] : "";
    char *prog_name = (char *)malloc(strlen(name) + 1);
    if (prog_name != NULL) {
        strcpy(prog_name, name);
    } else {
        /* Handle error */
    }
    free(prog_name);
    return 0;
}
```

Remember to add a byte to the destination string size to accommodate the null-termination character.

Compliant Solution {argv}

The `strcpy_s()` function provides additional safeguards, including accepting the size of the destination buffer as an additional argument. (See [STR07-C. Use the bounds-checking interfaces for string manipulation](#).)

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    /* Ensure argv[0] is not null */
    const char *const name = (argc && argv[0]) ? argv[0] : "";
    char *prog_name;
    size_t prog_size;

    prog_size = strlen(name) + 1;
    prog_name = (char *)malloc(prog_size);

    if (prog_name != NULL) {
        if (strcpy_s(prog_name, prog_size, name)) {
            /* Handle error */
        }
    } else {
        /* Handle error */
    }
    /* ... */
    free(prog_name);
    return 0;
}
```

The `strcpy_s()` function can be used to copy data to or from dynamically allocated memory or a statically allocated array. If insufficient space is available, `strcpy_s()` returns an error.

Compliant Solution (`argv`)

If an argument will not be modified or concatenated, there is no reason to make a copy of the string. Not copying a string is the best way to prevent a buffer overflow and is also the most efficient solution. Care must be taken to avoid assuming that `argv[0]` is non-null.

```
int main(int argc, char *argv[]) {
    /* Ensure argv[0] is not null */
    const char * const prog_name = (argc && argv[0]) ? argv[0] : "";
    /* ... */
    return 0;
}
```

Noncompliant Code Example (`getenv()`)

According to the C Standard, 7.22.4.6 [ISO/IEC 9899:2011]

The `getenv` function searches an environment list, provided by the host environment, for a string that matches the string pointed to by `name`. The set of environment names and the method for altering the environment list are implementation defined.

Environment variables can be arbitrarily large, and copying them into fixed-length arrays without first determining the size and allocating adequate storage can result in a buffer overflow.

```
#include <stdlib.h>
#include <string.h>

void func(void) {
    char buff[256];
    char *editor = getenv("EDITOR");
    if (editor == NULL) {
        /* EDITOR environment variable not set */
    } else {
        strcpy(buff, editor);
    }
}
```

Compliant Solution (`getenv()`)

Environmental variables are loaded into process memory when the program is loaded. As a result, the length of these strings can be determined by calling the `strlen()` function, and the resulting length can be used to allocate adequate dynamic memory:

```
#include <stdlib.h>
#include <string.h>

void func(void) {
    char *buff;
    char *editor = getenv("EDITOR");
    if (editor == NULL) {
        /* EDITOR environment variable not set */
    } else {
        size_t len = strlen(editor) + 1;
        buff = (char *)malloc(len);
        if (buff == NULL) {
            /* Handle error */
        }
        memcpy(buff, editor, len);
        free(buff);
    }
}
```

Noncompliant Code Example (`sprintf()`)

In this noncompliant code example, `name` refers to an external string; it could have originated from user input, the file system, or the network. The program constructs a file name from the string in preparation for opening the file.

```
#include <stdio.h>

void func(const char *name) {
    char filename[128];
    sprintf(filename, "%s.txt", name);
}
```

Because the `sprintf()` function makes no guarantees regarding the length of the generated string, a sufficiently long string in `name` could generate a buffer overflow.

Compliant Solution (`sprintf()`)

The buffer overflow in the preceding noncompliant example can be prevented by adding a precision to the `%s` conversion specification. If the precision is specified, no more than that many bytes are written. The precision `123` in this compliant solution ensures that `filename` can contain the first 123 characters of `name`, the `.txt` extension, and the null terminator.

```
#include <stdio.h>
```

```

void func(const char *name) {
    char filename[128];
    sprintf(filename, "%.123s.txt", name);
}

```

Compliant Solution (`snprintf()`)

A more general solution is to use the `snprintf()` function:

```

#include <stdio.h>

void func(const char *name) {
    char filename[128];
    snprintf(filename, sizeof(filename), "%s.txt", name);
}

```

Risk Assessment

Copying string data to a buffer that is too small to hold that data results in a buffer overflow. Attackers can exploit this condition to execute arbitrary code with the permissions of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR31-C	High	Likely	Medium	P18	L1

Related Guidelines

CERT C Secure Coding Standard	STR03-C. Do not inadvertently truncate a string STR07-C. Use the bounds-checking interfaces for remediation of existing string manipulation code MSC24-C. Do not use deprecated or obsolescent functions MEM00-C. Allocate and free memory in the same module, at the same level of abstraction FI034-C. Distinguish between characters read from a file and EOF or WEOF
ISO/IEC TR 24772:2013	String Termination [CJM] Buffer Boundary Violation (Buffer Overflow) [HCB] Unchecked Array Copying [XYW]
ISO/IEC TS 17961:2013	Using a tainted value to write to an object using a formatted input or output function [taintformatio] Tainted strings are passed to a string copying function [taintstrncpy]
MITRE CWE	CWE-119 , Improper Restriction of Operations within the Bounds of a Memory Buffer CWE-120 , Buffer Copy without Checking Size of Input ("Classic Buffer Overflow") CWE-123 , Write-what-where Condition CWE-125 , Out-of-bounds Read CWE-193 , Off-by-one Error

Bibliography

[Dowd 2006]	Chapter 7, "Program Building Blocks" ("Loop Constructs," pp. 327-336)
[Drepper 2006]	Section 2.1.1, "Respecting Memory Bounds"
[ISO/IEC 9899:2011]	K.3.5.4.1, "The <code>gets_s</code> Function"
[Lai 2006]	
[NIST 2006]	SAMATE Reference Dataset Test Case ID 000-000-088
[Seacord 2013b]	Chapter 2, "Strings"
[xorl 2009]	FreeBSD-SA-09:11: NTPd Remote Stack Based Buffer Overflows

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/KAE>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
allow_extra_args	Whether to allow additional arguments that are not used by the format string.	True
allow_gnu_extensions	Whether to allow the GNU extensions to format specifications.	True
allow_unknown_specs	Whether to allow unknown format specifications. It may be necessary to set this option when using implementation-specific extensions. Arguments are not checked when the format string contains unknown format specifications.	True
blacklist	Dictionary of header globbing to [list of] function name globbing[s] of forbidden functions.	dict(...)
functions	A dictionary mapping the names of the functions to check, to a triple {function_kind, fmt_param_index, arg_start_index} where function_kind is either 'printf' or 'scanf', fmt_param_index is the index of the format-string parameter, and arg_start_index is the index of the first variadic argument.	dict(...)
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
arg_type_mismatch	{} expects argument of type '{}', but argument {} has type '{}' (disabled)
buffer_too_small	{} may write up to {} characters to buffer of size {}.
forbidden_libfunc_call	Call to forbidden function.
invalid_conversion	Invalid or non-standard conversion specification (disabled)
matching_arg_expected	{} expects a matching '{}' argument (disabled)
maybe_too_small	Target buffer may be too small. Use snprintf() instead.
precision_for_conversion	Precision must not be used with %{} conversion specifier (disabled)
too_many_args	Too many arguments for format. (disabled)
too_small	Target buffer has {} characters, but sprintf() may write up to {} characters (including null terminator).
unknown_buffer_size	Potential buffer overflow: {} used with buffer of unknown size.
unlimited_read	Potential buffer overflow: {} has no limit on amount of characters read.
unsupported_assignment_suppression	%n does not support assignment suppression (disabled)
unsupported_field_width	%n does not support field width (disabled)
unsupported_flags	%n does not support flags (disabled)
unsupported_flags_modifiers	Cannot use any flags or modifiers with '%%' (disabled)
unsupported_hash	%{} does not support the '#' flag (disabled)
unsupported_i_flag	%{} does not support the 'l' flag (disabled)
unsupported_length_modifier	%{} does not support the '{}' length modifier (disabled)
unsupported_tick	%{} does not support the "" flag (disabled)
unsupported_zero	%{} does not support the '0' flag (disabled)

CertC++-STR32

Do not pass a non-null-terminated character sequence to a library function that expects a string.

Input: IR

Source languages: C++

Details

Many library functions accept a string or wide string argument with the constraint that the string they receive is properly null-terminated. Passing a character sequence or wide character sequence that is not null-terminated to such a function can result in accessing memory that is outside the bounds of the object. Do not pass a character sequence or wide character sequence that is not null-terminated to a library function that expects a string or wide string argument.

Noncompliant Code Example

This code example is noncompliant because the character sequence `c_str` will not be null-terminated when passed as an argument to `printf()`. (See [STR11-C. Do not specify the bound of a character array initialized with a string literal](#) on how to properly initialize character arrays.)

```
#include <stdio.h>

void func(void) {
    char c_str[3] = "abc";
    printf("%s\n", c_str);
}
```

Compliant Solution

This compliant solution does not specify the bound of the character array in the array declaration. If the array bound is omitted, the compiler allocates sufficient storage to store the entire string literal, including the terminating null character.

```
#include <stdio.h>

void func(void) {
    char c_str[] = "abc";
    printf("%s\n", c_str);
}
```

Noncompliant Code Example

This code example is noncompliant because the wide character sequence `cur_msg` will not be null-terminated when passed to `wcslen()`. This will occur if `lessen_memory_usage()` is invoked while `cur_msg_size` still has its initial value of 1024.

```
#include <stdlib.h>
#include <wchar.h>

wchar_t *cur_msg = NULL;
size_t cur_msg_size = 1024;
size_t cur_msg_len = 0;

void lessen_memory_usage(void) {
    wchar_t *temp;
    size_t temp_size;

    /* ... */

    if (cur_msg != NULL) {
        temp_size = cur_msg_size / 2 + 1;
        temp = realloc(cur_msg, temp_size * sizeof(wchar_t));
        /* temp &and cur_msg may no longer be null-terminated */
        if (temp == NULL) {
            /* Handle error */
        }

        cur_msg = temp;
        cur_msg_size = temp_size;
        cur_msg_len = wcslen(cur_msg);
    }
}
```

Compliant Solution

In this compliant solution, `cur_msg` will always be null-terminated when passed to `wcslen()`:

```
#include <stdlib.h>
#include <wchar.h>

wchar_t *cur_msg = NULL;
size_t cur_msg_size = 1024;
size_t cur_msg_len = 0;

void lessen_memory_usage(void) {
    wchar_t *temp;
    size_t temp_size;

    /* ... */

    if (cur_msg != NULL) {
        temp_size = cur_msg_size / 2 + 1;
        temp = realloc(cur_msg, temp_size * sizeof(wchar_t));
        /* temp and cur_msg may no longer be null-terminated */
        if (temp == NULL) {
            /* Handle error */
        }

        cur_msg = temp;
        /* Properly null-terminate cur_msg */
    }
}
```

```

        cur_msg[temp_size - 1] = L'\0';
        cur_msg_size = temp_size;
        cur_msg_len = wcslen(cur_msg);
    }
}

```

Noncompliant Code Example (`strncpy()`)

Although the `strncpy()` function takes a string as input, it does not guarantee that the resulting value is still null-terminated. In the following noncompliant code example, if no null character is contained in the first `n` characters of the `source` array, the result will not be null-terminated. Passing a non-null-terminated character sequence to `strlen()` is undefined behavior.

```
#include <string.h>

enum { STR_SIZE = 32 };

size_t func(const char *source) {
    char c_str[STR_SIZE];
    size_t ret = 0;

    if (source) {
        c_str[sizeof(c_str) - 1] = '\0';
        strncpy(c_str, source, sizeof(c_str));
        ret = strlen(c_str);
    } else {
        /* Handle null pointer */
    }
    return ret;
}
```

Compliant Solution (Truncation)

This compliant solution is correct if the programmer's intent is to truncate the string:

```
#include <string.h>

enum { STR_SIZE = 32 };

size_t func(const char *source) {
    char c_str[STR_SIZE];
    size_t ret = 0;

    if (source) {
        strncpy(c_str, source, sizeof(c_str) - 1);
        c_str[sizeof(c_str) - 1] = '\0';
        ret = strlen(c_str);
    } else {
        /* Handle null pointer */
    }
    return ret;
}
```

Compliant Solution (Truncation, `strncpy_s()`)

The C Standard, Annex K `strncpy_s()` function can also be used to copy with truncation. The `strncpy_s()` function copies up to `n` characters from the source array to a destination array. If no null character was copied from the source array, then the `n`th position in the destination array is set to a null character, guaranteeing that the resulting string is null-terminated.

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>

enum { STR_SIZE = 32 };

size_t func(const char *source) {
    char a[STR_SIZE];
    size_t ret = 0;

    if (source) {
        errno_t err = strncpy_s(
            a, sizeof(a), source, strlen(source)
        );
        if (err != 0) {
            /* Handle error */
        } else {
            ret = strnlen_s(a, sizeof(a));
        }
    } else {
        /* Handle null pointer */
    }
    return ret;
}
```

Compliant Solution (Copy without Truncation)

If the programmer's intent is to copy without truncation, this compliant solution copies the data and guarantees that the resulting array is null-terminated. If the string cannot be copied, it is handled as an error condition.

```
#include <string.h>

enum { STR_SIZE = 32 },
```

```

size_t func(const char *source) {
    char c_str[STR_SIZE];
    size_t ret = 0;

    if (source) {
        if (strlen(source) < sizeof(c_str)) {
            strcpy(c_str, source);
            ret = strlen(c_str);
        } else {
            /* Handle string-too-large */
        }
    } else {
        /* Handle null pointer */
    }
    return ret;
}

```

Risk Assessment

Failure to properly null-terminate a character sequence that is passed to a library function that expects a string can result in buffer overflows and the execution of arbitrary code with the permissions of the vulnerable process. Null-termination errors can also result in unintended information disclosure.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR32-C	High	Probable	Medium	P12	L1

Related Guidelines

ISO/IEC TR 24772:2013	String Termination [CMJ]
ISO/IEC TS 17961:2013	Passing a non-null-terminated character sequence to a library function that expects a string [strmod]
MITRE CWE	CWE-119 , Improper Restriction of Operations within the Bounds of a Memory Buffer CWE-123 , Write-what-where Condition CWE-125 , Out-of-bounds Read CWE-170 , Improper Null Termination

Bibliography

[Seacord 2013]	Chapter 2, "Strings"
[Viega 2005]	Section 5.2.14, "Miscalculated <code>NULL</code> Termination"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/KgE>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
functions_under_test	Functions which should not use non-null-terminated character sequences.	['strcpy', 'strncpy', 'strcat', 'strncat', 'strcmp', 'strncmp', 'strcoll', 'strxfrm', 'strchr', 'strcspn', 'strpbrk', 'strchr', 'strrchr', 'strspn', 'strtok', 'strlen', 'strftime', 'fprintf', 'printf', 'fscanf', 'scanf', 'snprintf', 'sprintf', 'sscanf', 'vfprintf', 'vfprintf', 'vscanf', 'vsnprintf', 'vsprintf', 'vsscanf', 'fwprintf', 'fwscanf', 'swprintf', 'swscanf', 'vswprintf', 'vswscanf', 'vwprintf', 'wprintf', 'wscanf', 'fputws', 'wcstod', 'wcstold', 'wcstol', 'wcstoll', 'wcstoul', 'wcstoull', 'wcscpy', 'wcsncpy', 'wcscat', 'wcscmp', 'wcscoll', 'wcscncmp', 'wcsxfrm', 'wcschr', 'wcscspn', 'wcspbrk', 'wCSRchr', 'wcspchr', 'wcspn', 'wcstr', 'wcstok', 'wcslen', 'wcsftime', 'mbrtoc16', 'c16rtomb', 'mbrtoc32', 'c32rtomb']
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
non-null-termination	Do not pass a non-null-terminated character sequence to a library function that expects a string.

CertC++-STR34

Cast characters to unsigned char before converting to larger integer sizes.

Input: IR

Source languages: C++

Details

Signed character data must be converted to `unsigned char` before being assigned or converted to a larger signed type. This rule applies to both `signed char` and `(plain) char` characters on implementations where `char` is defined to have the same range, representation, and behaviors as `signed char`.

However, this rule is applicable only in cases where the character data may contain values that can be interpreted as negative numbers. For example, if the `char` type is represented by a two's complement 8-bit value, any character value greater than +127 is interpreted as a negative value.

This rule is a generalization of [STR37-C. Arguments to character-handling functions must be representable as an `unsigned char`.](#)

Noncompliant Code Example

This noncompliant code example is taken from a [vulnerability](#) in bash versions 1.14.6 and earlier that led to the release of CERT Advisory [CA-1996-22](#). This vulnerability resulted from the sign extension of character data referenced by the `c_str` pointer in the `yy_string_get()` function in the `parse.y` module of the bash source code:

```
static int yy_string_get(void) {
    register char *c_str;
    register int c;

    c_str = bash_input.location.string;
    c = EOF;

    /* If the string doesn't exist or is empty, EOF found */
    if (*c_str && *c_str) {
        c = *c_str++;
        bash_input.location.string = c_str;
    }
}
```

```
    return (c);
}
```

The `c_str` variable is used to traverse the character string containing the command line to be parsed. As characters are retrieved from this pointer, they are stored in a variable of type `int`. For implementations in which the `char` type is defined to have the same range, representation, and behavior as `signed char`, this value is sign-extended when assigned to the `int` variable. For character code 255 decimal (-1 in two's complement form), this sign extension results in the value -1 being assigned to the integer, which is indistinguishable from `EOF`.

Noncompliant Code Example

This problem can be repaired by explicitly declaring the `c_str` variable as `unsigned char`:

```
static int yy_string_get(void) {
    register unsigned char *c_str;
    register int c;

    c_str = bash_input.location.string;
    c = EOF;

    /* If the string doesn't exist or is empty, EOF found */
    if (c_str && *c_str) {
        c = *c_str++;
        bash_input.location.string = c_str;
    }
    return (c);
}
```

This example, however, violates [STR04-C. Use plain char for characters in the basic character set](#).

Compliant Solution

In this compliant solution, the result of the expression `*c_str++` is cast to `unsigned char` before assignment to the `int` variable `c`:

```
static int yy_string_get(void) {
    register char *c_str;
    register int c;

    c_str = bash_input.location.string;
    c = EOF;

    /* If the string doesn't exist or is empty, EOF found */
    if (c_str && *c_str) {
        /* Cast to unsigned type */
        c = (unsigned char)*c_str++;

        bash_input.location.string = c_str;
    }
    return (c);
}
```

Noncompliant Code Example

In this noncompliant code example, the cast of `*s` to `unsigned int` can result in a value in excess of `UCHAR_MAX` because of integer promotions, a violation of [ARR30-C. Do not form or use out-of-bounds pointers or array subscripts](#):

```
#include <limits.h>
#include <stddef.h>

static const char table[UCHAR_MAX + 1] = { 'a' /* ... */ };

ptrdiff_t first_not_in_table(const char *c_str) {
    for (const char *s = c_str; *s; ++s) {
        if (table[(unsigned int)*s] != *s) {
            return s - c_str;
        }
    }
    return -1;
}
```

Compliant Solution

This compliant solution casts the value of type `char` to `unsigned char` before the implicit promotion to a larger type:

```
#include <limits.h>
#include <stddef.h>

static const char table[UCHAR_MAX + 1] = { 'a' /* ... */ };

ptrdiff_t first_not_in_table(const char *c_str) {
    for (const char *s = c_str; *s; ++s) {
        if (table[(unsigned char)*s] != *s) {
            return s - c_str;
        }
    }
    return -1;
}
```

Risk Assessment

Conversion of character data resulting in a value in excess of `UCHAR_MAX` is an often-missed error that can result in a disturbingly broad range of potentially severe [vulnerabilities](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR34-C	Medium	Probable	Medium	P8	L2

Related Guidelines

CERT C Secure Coding Standard	STR37-C. Arguments to character-handling functions must be representable as an unsigned char STR04-C. Use plain char for characters in the basic character set ARR30-C. Do not form or use out-of-bounds pointers or array subscripts
ISO/IEC TS 17961:2013	Conversion of signed characters to wider integer types before a check for EOF [signconv]
MISRA-C:2012	Rule 10.1 (required) Rule 10.2 (required) Rule 10.3 (required) Rule 10.4 (required)
MITRE CWE	CWE-704 , Incorrect Type Conversion or Cast

Bibliography

[xorl 2009]	CVE-2009-0887: Linux-PAM Signedness Issue
-------------	---

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/QgBi>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
only_arguments_of	Can be used to provide a set of function/macro names; only arguments to them will be considered then	set([])
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_from_char_to_larger_type	Cast characters to unsigned char before converting to larger integer sizes

CertC++-STR37

Arguments to character-handling functions must be representable as an unsigned char.

Input: IR
Source languages: C++

Details

According to the C Standard, 7.4 [[ISO/IEC 9899:2011](#)],

The header `<cctype.h>` declares several functions useful for classifying and mapping characters. In all cases the argument is an `int`, the value of which shall be representable as an `unsigned char` or shall equal the value of the macro `EOF`. If the argument has any other value, the behavior is [undefined](#).

See also [undefined behavior 113](#).

This rule is applicable only to code that runs on platforms where the `char` data type is defined to have the same range, representation, and behavior as `signed`

char.

Following are the character classification functions that this rule addresses:

isalnum	isalpha	isascii ^{XSI}	isblank
iscntrl	isdigit	isgraph	islower
isprint	ispunct	isspace	isupper
isxdigit	toascii ^{XSI}	toupper	tolower

XSI denotes an X/Open System Interfaces Extension to ISO/IEC 9945-POSIX. These functions are not defined by the C Standard.

This rule is a specific instance of [STR34-C. Cast characters to unsigned char before converting to larger integer sizes](#).

Noncompliant Code Example

On implementations where plain `char` is signed, this code example is noncompliant because the parameter to `isspace`, `*t`, is defined as a `const char *`, and this value might not be representable as an `unsigned char`:

```
#include <ctype.h>
#include <string.h>

size_t count_preceding_whitespace(const char *s) {
    const char *t = s;
    size_t length = strlen(s) + 1;
    while (isspace(*t) && (t - s < length)) {
        ++t;
    }
    return t - s;
}
```

The argument to `isspace` must be `EOF` or representable as an `unsigned char`; otherwise, the result is undefined.

Compliant Solution

This compliant solution casts the character to `unsigned char` before passing it as an argument to the `isspace` function:

```
#include <ctype.h>
#include <string.h>

size_t count_preceding_whitespace(const char *s) {
    const char *t = s;
    size_t length = strlen(s) + 1;
    while (isspace((unsigned char)*t) && (t - s < length)) {
        ++t;
    }
    return t - s;
}
```

Risk Assessment

Passing values to character handling functions that cannot be represented as an `unsigned char` to character handling functions is [undefined behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR37-C	Low	Unlikely	Low	P3	L3

Related Guidelines

CERT C Secure Coding Standard	STR34-C. Cast characters to unsigned char before converting to larger integer sizes
ISO/IEC TS 17961	Passing arguments to character-handling functions that are not representable as <code>unsigned char</code> [<code>chrsgnext</code>]
MITRE CWE	CWE-704 , Incorrect Type Conversion or Cast CWE-686 , Function Call with Incorrect Argument Type

Bibliography

[ISO/IEC 9899:2011]	7.4, "Character Handling < <code>ctype.h</code> >"
[Kettlewell 2002]	Section 1.1, "< <code>ctype.h</code> > and Characters Types"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/fAs>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
only_arguments_of	Can be used to provide a set of function/macro names; only arguments to them will be considered then	set(['isupper', 'toupper', 'isalnum', 'toascii', 'isxdigit', 'isgraph', 'isspace', 'iscntrl', 'islower', 'isprint', 'isalpha', 'isascii', 'isdigit', 'ispunct', 'isblank', 'tolower'])
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_from_char_to_larger_type	Arguments to character-handling functions must be representable as an unsigned char

CertC++-STR38

Do not confuse narrow and wide character strings and functions.

Input: IR

Source languages: C++

Details

Passing narrow string arguments to wide string functions or wide string arguments to narrow string functions can lead to [unexpected](#) and [undefined behavior](#). Scaling problems are likely because of the difference in size between wide and narrow characters. (See [ARR39-C. Do not add or subtract a scaled integer to a pointer](#).) Because wide strings are terminated by a null wide character and can contain null bytes, determining the length is also problematic.

Because `wchar_t` and `char` are distinct types, many compilers will produce a warning diagnostic if an inappropriate function is used. (See [MSC00-C. Compile cleanly at high warning levels](#).)

Noncompliant Code Example (Wide Strings with Narrow String Functions)

This noncompliant code example incorrectly uses the `strncpy()` function in an attempt to copy up to 10 wide characters. However, because wide characters can contain null bytes, the copy operation may end earlier than anticipated, resulting in the truncation of the wide string.

```
#include <stddef.h>
#include <string.h>

void func(void) {
    wchar_t wide_str1[] = L"0123456789";
    wchar_t wide_str2[] = L"0000000000";
    strncpy(wide_str2, wide_str1, 10);
}
```

Noncompliant Code Example (Narrow Strings with Wide String Functions)

This noncompliant code example incorrectly invokes the `wcsncpy()` function to copy up to 10 wide characters from `narrow_str1` to `narrow_str2`. Because `narrow_str2` is a narrow string, it has insufficient memory to store the result of the copy and the copy will result in a buffer overflow.

```
#include <wchar.h>

void func(void) {
    char narrow_str1[] = "01234567890123456789";
    char narrow_str2[] = "0000000000";
    wcsncpy(narrow_str2, narrow_str1, 10);
}
```

Compliant Solution

This compliant solution uses the proper-width functions. Using `wcsncpy()` for wide character strings and `strncpy()` for narrow character strings ensures

that data is not truncated and buffer overflow does not occur.

```
#include <string.h>
#include <wchar.h>

void func(void) {
    wchar_t wide_str1[] = L"0123456789";
    wchar_t wide_str2[] = L"0000000000";
    /* Use of proper-width function */
    wcsncpy(wide_str2, wide_str1, 10);

    char narrow_str1[] = "0123456789";
    char narrow_str2[] = "0000000000";
    /* Use of proper-width function */
    strncpy(narrow_str2, narrow_str1, 10);
}
```

Noncompliant Code Example (`strlen()`)

In this noncompliant code example, the `strlen()` function is used to determine the size of a wide character string:

```
#include <stdlib.h>
#include <string.h>

void func(void) {
    wchar_t wide_str1[] = L"0123456789";
    wchar_t *wide_str2 = (wchar_t *)malloc(strlen(wide_str1) + 1);
    if (wide_str2 == NULL) {
        /* Handle error */
    }
    /* ... */
    free(wide_str2);
    wide_str2 = NULL;
}
```

The `strlen()` function determines the number of characters that precede the terminating null character. However, wide characters can contain null bytes, particularly when expressing characters from the ASCII character set, as in this example. As a result, the `strlen()` function will return the number of bytes preceding the first null byte in the wide string.

Compliant Solution

This compliant solution correctly calculates the number of bytes required to contain a copy of the wide string, including the terminating null wide character:

```
#include <stdlib.h>
#include <wchar.h>

void func(void) {
    wchar_t wide_str1[] = L"0123456789";
    wchar_t *wide_str2 = (wchar_t *)malloc(
        (wcslen(wide_str1) + 1) * sizeof(wchar_t));
    if (wide_str2 == NULL) {
        /* Handle error */
    }
    /* ... */

    free(wide_str2);
    wide_str2 = NULL;
}
```

Risk Assessment

Confusing narrow and wide character strings can result in buffer overflows, data truncation, and other defects.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR38-C	High	Likely	Low	P27	L1

Bibliography

[ISO/IEC 9899:2011]	7.24.2.4, "The <code>strncpy</code> Function" 7.29.4.2.2, "The <code>wcsncpy</code> Function"
-------------------------------------	--

Excerpt from SEI CERT C++ Coding Standard Wiki [\[https://www.securecoding.cert.org/confluence/x/FADAAQ\]](https://www.securecoding.cert.org/confluence/x/FADAAQ), Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	msg_for_narrow_and_wide_char(['klass', 'node'])
priority	Priority based on the combination of severity, likelihood and remediation cost	27
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low
reported_messages	If provided, only messages of these types are reported.	167
reported_severities	List of severities to display.	{'error', 'warning', 'remark'}
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

CertC++-MEM30

Do not access freed memory.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Evaluating a pointer - including dereferencing the pointer, using it as an operand of an arithmetic operation, type casting it, and using it as the right-hand side of an assignment - into memory that has been deallocated by a memory management function is [undefined behavior](#). Pointers to memory that has been deallocated are called *dangling pointers*. Accessing a dangling pointer can result in exploitable [vulnerabilities](#).

According to the C Standard, using the value of a pointer that refers to space deallocated by a call to the `free()` or `realloc()` function is undefined behavior. (See [undefined behavior 177](#).)

Reading a pointer to deallocated memory is undefined behavior because the pointer value is [indeterminate](#) and might be a [trap representation](#). Fetching a trap representation might perform a hardware trap (but is not required to).

It is at the memory manager's discretion when to reallocate or recycle the freed memory. When memory is freed, all pointers into it become invalid, and its contents might either be returned to the operating system, making the freed space inaccessible, or remain intact and accessible. As a result, the data at the freed location can appear to be valid but change unexpectedly. Consequently, memory must not be written to or read from once it is freed.

Noncompliant Code Example

This example from Brian Kernighan and Dennis Ritchie [[Kernighan 1988](#)] shows both the incorrect and correct techniques for freeing the memory associated with a linked list. In their (intentionally) incorrect example, `p` is freed before `p->next` is executed, so that `p->next` reads memory that has already been freed.

```
#include <stdlib.h>

struct node {
    int value;
    struct node *next;
};

void free_list(struct node *head) {
    for (struct node *p = head; p != NULL; p = p->next) {
        free(p);
    }
}
```

Compliant Solution

Kernighan and Ritchie correct this error by storing a reference to `p->next` in `q` before freeing `p`:

```
#include <stdlib.h>

struct node {
    int value;
    struct node *next;
};

void free_list(struct node *head) {
    struct node *q;
    for (struct node *p = head; p != NULL; p = q) {
        q = p->next;
        free(p);
    }
}
```

}

Noncompliant Code Example

In this noncompliant code example, `buf` is written to after it has been freed. Write-after-free vulnerabilities can be [exploited](#) to run arbitrary code with the permissions of the vulnerable process. Typically, allocations and frees are far removed, making it difficult to recognize and diagnose these problems.

```
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char *return_val = 0;
    const size_t bufsize = strlen(argv[0]) + 1;
    char *buf = (char *)malloc(bufsize);
    if (!buf) {
        return EXIT_FAILURE;
    }
    /* ... */
    free(buf);
    /* ... */
    strcpy(buf, argv[0]);
    /* ... */
    return EXIT_SUCCESS;
}
```

Compliant Solution

In this compliant solution, the memory is freed after its final use:

```
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char *return_val = 0;
    const size_t bufsize = strlen(argv[0]) + 1;
    char *buf = (char *)malloc(bufsize);
    if (!buf) {
        return EXIT_FAILURE;
    }
    /* ... */
    strcpy(buf, argv[0]);
    /* ... */
    free(buf);
    return EXIT_SUCCESS;
}
```

Noncompliant Code Example

In this noncompliant example, `realloc()` may free `c_str1` when it returns a null pointer, resulting in `c_str1` being freed twice. The C Standards Committee's proposed response to [Defect Report #400](#) makes it implementation-defined whether or not the old object is deallocated when `size` is zero and memory for the new object is not allocated. The current implementation of `realloc()` in the GNU C Library and Microsoft Visual Studio's Runtime Library will free `c_str1` and return a null pointer for zero byte allocations. Freeing a pointer twice can result in a potentially exploitable vulnerability commonly referred to as a *double-free vulnerability* [[Seacord 2013b](#)].

```
#include <stdlib.h>

void f(char *c_str1, size_t size) {
    char *c_str2 = (char *)realloc(c_str1, size);
    if (c_str2 == NULL) {
        free(c_str1);
    }
}
```

Compliant Solution

This compliant solution does not pass a size argument of zero to the `realloc()` function, eliminating the possibility of `c_str1` being freed twice:

```
#include <stdlib.h>

void f(char *c_str1, size_t size) {
    if (size != 0) {
        char *c_str2 = (char *)realloc(c_str1, size);
        if (c_str2 == NULL) {
            free(c_str1);
        }
    }
    else {
        free(c_str1);
    }
}
```

If the intent of calling `f()` is to reduce the size of the object, then doing nothing when the size is zero would be unexpected; instead, this compliant solution frees the object.

Noncompliant Code Example

In this noncompliant example [[CVE-2009-1364](#)] from `libwmf` version 0.2.8.4, the return value of `gdRealloc` (a simple wrapper around `realloc()` that reallocates space pointed to by `im->clip->list`) is set to `more`. However, the value of `im->clip->list` is used directly afterwards in the code, and the C Standard specifies that if `realloc()` moves the area pointed to, then the original block is freed. An attacker can then execute arbitrary code by forcing a reallocation (with a sufficient `im->clip->count`) and accessing freed memory [[xorl 2009](#)].

```

void gdClipSetAdd(gdImagePtr im, gdClipRectanglePtr rect) {
    gdClipRectanglePtr more;
    if (im->clip == 0) {
        /* ... */
    }
    if (im->clip->count == im->clip->max) {
        more = gdRealloc (im->clip->list,(im->clip->max + 8) *
                         sizeof (gdClipRectangle));
        /*
         * If the realloc fails, then we have not lost the
         * im->clip->list value.
         */
        if (more == 0) return;
        im->clip->max += 8;
    }
    im->clip->list[im->clip->count] = *rect;
    im->clip->count++;
}

```

Compliant Solution

This compliant solution simply reassigns `im->clip->list` to the value of `more` after the call to `realloc()`:

```

void gdClipSetAdd(gdImagePtr im, gdClipRectanglePtr rect) {
    gdClipRectanglePtr more;
    if (im->clip == 0) {
        /* ... */
    }
    if (im->clip->count == im->clip->max) {
        more = gdRealloc (im->clip->list,(im->clip->max + 8) *
                         sizeof (gdClipRectangle));
        if (more == 0) return;
        im->clip->max += 8;
        im->clip->list = more;
    }
    im->clip->list[im->clip->count] = *rect;
    im->clip->count++;
}

```

Risk Assessment

Reading memory that has already been freed can lead to abnormal program termination and denial-of-service attacks. Writing memory that has already been freed can additionally lead to the execution of arbitrary code with the permissions of the vulnerable process.

Freeing memory multiple times has similar consequences to accessing memory after it is freed. Reading a pointer to deallocated memory is [undefined behavior](#) because the pointer value is [indeterminate](#) and might be a [trap representation](#). When reading from or writing to freed memory does not cause a trap, it may corrupt the underlying data structures that manage the heap in a manner that can be exploited to execute arbitrary code. Alternatively, writing to memory after it has been freed might modify memory that has been reallocated.

Programmers should be wary when freeing memory in a loop or conditional statement; if coded incorrectly, these constructs can lead to double-free vulnerabilities. It is also a common error to misuse the `realloc()` function in a manner that results in double-free vulnerabilities. (See [MEM04-C. Beware of zero-length allocations](#).)

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM30-C	High	Likely	Medium	P18	L1

Related Guidelines

CERT C Secure Coding Standard	MEM01-C. Store a new value in pointers immediately after free()
SEI CERT C++ Coding Standard	MEM50-CPP. Do not access freed memory
ISO/IEC TR 24772:2013	Dangling References to Stack Frames [DCM] Dangling Reference to Heap [XYK]
ISO/IEC TS 17961	Accessing freed memory [accfree] Freeing memory multiple times [dblfree]
MISRA C:2012	Rule 18.6 (required)
MITRE CWE	CWE-415, Double Free CWE-416, Use After Free

Bibliography

[ISO/IEC 9899:2011]	7.22.3, "Memory Management Functions"
[Kernighan 1988]	Section 7.8.5, "Storage Management"
[OWASP Freed Memory]	
[MIT 2005]	
[Seacord 2013b]	Chapter 4, "Dynamic Memory Management"
[Viega 2005]	Section 5.2.19, "Using Freed Memory"
[VU#623332]	
[xorl 2009]	CVE-2009-1364: LibWMF Pointer Use after free()

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/vAE>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
resources	Configuration of memory types and allocate/release functions for them in addition to what <code>iranalysis.config</code> provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)

Possible Messages

Name	Message
double_free	Dynamic memory released here was already released earlier
possible_double_free	Dynamic memory released here possibly already released earlier
possible_use_after_free	Dynamic memory possibly used after it was previously released
use_after_free	Dynamic memory used after it was previously released

CertC++-MEM31

Free dynamically allocated memory when no longer needed.

Input: IR

Source languages: C++

Note: Rule requires `CONFIG.Run_IRAnalysis_Checks = True`.

Details

Before the lifetime of the last pointer that stores the return value of a call to a standard memory allocation function has ended, it must be matched by a call to `free()` with that pointer value.

Noncompliant Code Example

In this noncompliant example, the object allocated by the call to `malloc()` is not freed before the end of the lifetime of the last pointer `text_buffer` referring to the object:

```
#include <stdlib.h>

enum { BUFFER_SIZE = 32 };

int f(void)
{
    char *text_buffer = (char *)malloc(BUFFER_SIZE);
    if (text_buffer == NULL) {
        return -1;
    }
    return 0;
}
```

Compliant Solution

In this compliant solution, the pointer is deallocated with a call to `free()`:

```
#include <stdlib.h>

enum { BUFFER_SIZE = 32 };

int f(void) {
    char *text_buffer = (char *)malloc(BUFFER_SIZE);
    if (text_buffer == NULL) {
        return -1;
    }

    free(text_buffer);
    return 0;
}
```

Exceptions

MEM31-C-EX1: Allocated memory does not need to be freed if it is assigned to a pointer with static storage duration whose lifetime is the entire execution of a program. The following code example illustrates a pointer that stores the return value from `malloc()` in a `static` variable:

```
#include <stdlib.h>

enum { BUFFER_SIZE = 32 };

int f(void) {
    static char *text_buffer = NULL;
    if (text_buffer == NULL) {
        text_buffer = (char *)malloc(BUFFER_SIZE);
        if (text_buffer == NULL) {
            return -1;
        }
    }
    return 0;
}
```

Risk Assessment

Failing to free memory can result in the exhaustion of system memory resources, which can lead to a [denial-of-service attack](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM31-C	Medium	Probable	Medium	P8	L2

Related Guidelines

ISO/IEC TR 24772:2013	Memory Leak [XYL]
ISO/IEC TS 17961	Failing to close files or free dynamic memory when they are no longer needed [fclose]
MITRE CWE	CWE-401 , Improper Release of Memory Before Removing Last Reference ("Memory Leak")

Bibliography

[ISO/IEC 9899:2011]	Subclause 7.22.3, "Memory Management Functions"
-------------------------------------	---

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/vQE>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
resources	Configuration of memory types and allocate/release functions for them in addition to what <code>iranalysis.config</code> provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)

Possible Messages

Name	Message
memory_leak	Call allocates leaking memory
possible_memory_leak	Call allocates possibly leaking memory

CertC++-MEM34

Only free memory allocated dynamically.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

The C Standard, Annex J [[ISO/IEC 9899:2011](#)], states that the behavior of a program is undefined when

The pointer argument to the `free` or `realloc` function does not match a pointer earlier returned by a memory management function, or the space has been deallocated by a call to `free` or `realloc`.

See also [undefined behavior 179](#).

Freeing memory that is not allocated dynamically can result in heap corruption and other serious errors. Do not call `free()` on a pointer other than one returned by a standard memory allocation function, such as `malloc()`, `calloc()`, `realloc()`, or `aligned_alloc()`.

A similar situation arises when `realloc()` is supplied a pointer to non-dynamically allocated memory. The `realloc()` function is used to resize a block of dynamic memory. If `realloc()` is supplied a pointer to memory not allocated by a standard memory allocation function, the behavior is undefined. One consequence is that the program may [terminate abnormally](#).

This rule does not apply to null pointers. The C Standard guarantees that if `free()` is passed a null pointer, no action occurs.

Noncompliant Code Example

This noncompliant code example sets `c_str` to reference either dynamically allocated memory or a statically allocated string literal depending on the value of `argc`. In either case, `c_str` is passed as an argument to `free()`. If anything other than dynamically allocated memory is referenced by `c_str`, the call to `free(c_str)` is erroneous.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum { MAX_ALLOCATION = 1000 };

int main(int argc, const char *argv[]) {
    char *c_str = NULL;
    size_t len;

    if (argc == 2) {
        len = strlen(argv[1]) + 1;
        if (len > MAX_ALLOCATION) {
            /* Handle error */
        }
        c_str = (char *)malloc(len);
        if (c_str == NULL) {
            /* Handle error */
        }
        strcpy(c_str, argv[1]);
    } else {
        c_str = "usage: $>a.exe [string]";
        printf("%s\n", c_str);
    }
    free(c_str);
    return 0;
}
```

Compliant Solution

This compliant solution eliminates the possibility of `c_str` referencing memory that is not allocated dynamically when passed to `free()`:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum { MAX_ALLOCATION = 1000 };

int main(int argc, const char *argv[]) {
    char *c_str = NULL;
    size_t len;

    if (argc == 2) {
        len = strlen(argv[1]) + 1;
        if (len > MAX_ALLOCATION) {
            /* Handle error */
        }
        c_str = (char *)malloc(len);
    }
}
```

```

if (c_str == NULL) {
    /* Handle error */
    strcpy(c_str, argv[1]);
} else {
    printf("%s\n", "usage: $>a.exe [string]");
    return EXIT_FAILURE;
}
free(c_str);
return 0;
}

```

Noncompliant Code Example (`realloc()`)

In this noncompliant example, the pointer parameter to `realloc()`, `buf`, does not refer to dynamically allocated memory:

```

#include <stdlib.h>

enum { BUFSIZE = 256 };

void f(void) {
    char buf[BUFSIZE];
    char *p = (char *)realloc(buf, 2 * BUFSIZE);
    if (p == NULL) {
        /* Handle error */
    }
}

```

Compliant Solution (`realloc()`)

In this compliant solution, `buf` refers to dynamically allocated memory:

```

#include <stdlib.h>

enum { BUFSIZE = 256 };

void f(void) {
    char *buf = (char *)malloc(BUFSIZE * sizeof(char));
    char *p = (char *)realloc(buf, 2 * BUFSIZE);
    if (p == NULL) {
        /* Handle error */
    }
}

```

Note that `realloc()` will behave properly even if `malloc()` failed, because when given a null pointer, `realloc()` behaves like a call to `malloc()`.

Risk Assessment

The consequences of this error depend on the [implementation](#), but they range from nothing to arbitrary code execution if that memory is reused by `malloc()`.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM34-C	High	Likely	Medium	P18	L1

Related Guidelines

CERT C Secure Coding Standard	MEM31-C. Free dynamically allocated memory when no longer needed
SEI CERT C++ Coding Standard	MEM51-CPP. Properly deallocate dynamically allocated resources
ISO/IEC TS 17961	Reallocating or freeing memory that was not dynamically allocated [xfree]
MITRE CWE	CWE-590, Free of Memory Not on the Heap

Bibliography

[ISO/IEC 9899:2011]	Subclause J.2, "Undefined Behavior"
[Seacord 2013b]	Chapter 4, "Dynamic Memory Management"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/wQE>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)

Possible Messages

Name	Message
possible_stack_free	{ } possibly released by call to { } is a stack object
stack_free	{ } released by call to { } is a stack object

CertC++-MEM35

Allocate sufficient memory for an object.

Input: IR

Source languages: C++

Details

The types of integer expressions used as size arguments to `malloc()`, `calloc()`, `realloc()`, or `aligned_alloc()` must have sufficient range to represent the size of the objects to be stored. If size arguments are incorrect or can be manipulated by an attacker, then a buffer overflow may occur. Incorrect size arguments, inadequate range checking, integer overflow, or truncation can result in the allocation of an inadequately sized buffer.

Typically, the amount of memory to allocate will be the size of the type of object to allocate. When allocating space for an array, the size of the object will be multiplied by the bounds of the array. When allocating space for a structure containing a flexible array member, the size of the array member must be added to the size of the structure. (See [MEM33-C. Allocate and copy structures containing a flexible array member dynamically](#).) Use the correct type of the object when computing the size of memory to allocate.

[STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator](#) is a specific instance of this rule.

Noncompliant Code Example (Pointer)

In this noncompliant code example, inadequate space is allocated for a `struct tm` object because the size of the pointer is being used to determine the size of the pointed-to object:

```
#include <stdlib.h>
#include <time.h>

struct tm *make_tm(int year, int mon, int day, int hour,
                   int min, int sec) {
    struct tm *tmb;
    tmb = (struct tm *)malloc(sizeof(tmb));
    if (tmb == NULL) {
        return NULL;
    }
    *tmb = (struct tm) {
        .tm_sec = sec, .tm_min = min, .tm_hour = hour,
        .tm_mday = day, .tm_mon = mon, .tm_year = year
    };
    return tmb;
}
```

Compliant Solution (Pointer)

In this compliant solution, the correct amount of memory is allocated for the `struct tm` object. When allocating space for a single object, passing the (dereferenced) pointer type to the `sizeof` operator is a simple way to allocate sufficient memory. Because the `sizeof` operator does not evaluate its operand, dereferencing an uninitialized or null pointer in this context is well-defined behavior.

```
#include <stdlib.h>
#include <time.h>

struct tm *make_tm(int year, int mon, int day, int hour,
                   int min, int sec) {
    struct tm *tmb;
    tmb = (struct tm *)malloc(sizeof(*tmb));
    if (tmb == NULL) {
        return NULL;
    }
    *tmb = (struct tm) {
        .tm_sec = sec, .tm_min = min, .tm_hour = hour,
```

```

        .tm_mday = day, .tm_mon = mon, .tm_year = year
    };
    return tmb;
}

```

Noncompliant Code Example (Integer)

In this noncompliant code example, an array of `long` is allocated and assigned to `p`. The code attempts to check for unsigned integer overflow in compliance with [INT30-C. Ensure that unsigned integer operations do not wrap](#) and also ensures that `len` is not equal to zero. (See [MEM04-C. Beware of zero-length allocations](#).) However, because `sizeof(int)` is used to compute the size, and not `sizeof(long)`, an insufficient amount of memory can be allocated on implementations where `sizeof(long)` is larger than `sizeof(int)`, and filling the array can cause a heap buffer overflow.

```

#include <stdint.h>
#include <stdlib.h>

void function(size_t len) {
    long *p;
    if (len == 0 || len > SIZE_MAX / sizeof(long)) {
        /* Handle overflow */
    }
    p = (long *)malloc(len * sizeof(int));
    if (p == NULL) {
        /* Handle error */
    }
    free(p);
}

```

Compliant Solution (Integer)

This compliant solution uses `sizeof(long)` to correctly size the memory allocation:

```

#include <stdint.h>
#include <stdlib.h>

void function(size_t len) {
    long *p;
    if (len == 0 || len > SIZE_MAX / sizeof(long)) {
        /* Handle overflow */
    }
    p = (long *)malloc(len * sizeof(long));
    if (p == NULL) {
        /* Handle error */
    }
    free(p);
}

```

Compliant Solution (Integer)

Alternatively, `sizeof(*p)` can be used to properly size the allocation:

```

#include <stdint.h>
#include <stdlib.h>

void function(size_t len) {
    long *p;
    if (len == 0 || len > SIZE_MAX / sizeof(*p)) {
        /* Handle overflow */
    }
    p = (long *)malloc(len * sizeof(*p));
    if (p == NULL) {
        /* Handle error */
    }
    free(p);
}

```

Risk Assessment

Providing invalid size arguments to memory allocation functions can lead to buffer overflows and the execution of arbitrary code with the permissions of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM35-C	High	Probable	High	P6	L2

Related Guidelines

CERT C Secure Coding Standard	ARR01-C. Do not apply the sizeof operator to a pointer when taking the size of an array INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data INT32-C. Ensure that operations on signed integers do not result in overflow INT18-C. Evaluate integer expressions in a larger size before comparing or assigning to that size MEM04-C. Beware of zero-length allocations
ISO/IEC TR 24772:2013	Buffer Boundary Violation [Buffer Overflow] [HCB]
ISO/IEC TS 17961:2013	Taking the size of a pointer to determine the size of the pointed-to type [sizeofptr]
MITRE CWE	CWE-131 , Incorrect Calculation of Buffer Size CWE-190 , Integer Overflow or Wraparound CWE-467 , Use of <code>sizeof()</code> on a Pointer Type

Bibliography

[Coverity 2007]	
[Drepper 2006]	Section 2.1.1, "Respecting Memory Bounds"
[Seacord 2013]	Chapter 4, "Dynamic Memory Management" Chapter 5, "Integer Security"
[Viega 2005]	Section 5.6.8, "Use of <code>sizeof()</code> on a Pointer Type"
[xorl 2009]	CVE-2009-0587: Evolution Data Server Base64 Integer Overflows

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/2wE>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
wrong_malloc_size	Wrong allocation size: <code>sizeof</code> on {} used, but {} expected

CertC++-MEM36

Do not modify the alignment of objects by calling `realloc()`.

Input: IR

Source languages: C++

Note: Rule requires `CONFIG.Run_IRAnalysis_Checks = True`.

Details

Do not invoke `realloc()` to modify the size of allocated objects that have stricter alignment requirements than those guaranteed by `malloc()`. Storage allocated by a call to the standard `aligned_alloc()` function, for example, can have stricter than normal alignment requirements. The C standard requires only that a pointer returned by `realloc()` be suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement.

Noncompliant Code Example

This noncompliant code example returns a pointer to allocated memory that has been aligned to a 4096-byte boundary. If the `resize` argument to the `realloc()` function is larger than the object referenced by `ptr`, then `realloc()` will allocate new memory that is suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement but may not preserve the stricter alignment of the original object.

```
#include <stdlib.h>

void func(void) {
    size_t resize = 1024;
    size_t alignment = 1 << 12;
    int *ptr;
    int *ptr1;

    if (NULL == (ptr = (int *)aligned_alloc(alignment, sizeof(int)))) {
        /* Handle error */
    }

    if (NULL == (ptr1 = (int *)realloc(ptr, resize))) {
        /* Handle error */
    }
}
```

Implementation Details

When compiled with GCC 4.1.2 and run on the x86_64 Red Hat Linux platform, the following code produces the following output:

CODE

```
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    size_t size = 16;
    size_t resize = 1024;
    size_t align = 1 << 12;
    int *ptr;
    int *ptr1;

    if (posix_memalign((void **)&ptr, align, size) != 0) {
        exit(EXIT_FAILURE);
    }

    printf("memory aligned to %zu bytes\n", align);
    printf("ptr = %p\n\n", ptr);

    if ((ptr1 = (int *) realloc((int *)ptr, resize)) == NULL) {
        exit(EXIT_FAILURE);
    }

    puts("After realloc():\n");
    printf("ptr1 = %p\n", ptr1);

    free(ptr1);
    return 0;
}
```

OUTPUT

```
memory aligned to 4096 bytes
ptr = 0x1621b000

After realloc():
ptr1 = 0x1621a010
```

ptr1 is no longer aligned to 4096 bytes.

Compliant Solution

This compliant solution allocates `resize` bytes of new memory with the same alignment as the old memory, copies the original memory content, and then frees the old memory. This solution has [implementation-defined behavior](#) because it depends on whether extended alignments in excess of `_Alignof(max_align_t)` are supported and the contexts in which they are supported. If not supported, the behavior of this compliant solution is undefined.

```
#include <stdlib.h>
#include <string.h>

void func(void) {
    size_t resize = 1024;
    size_t alignment = 1 << 12;
    int *ptr;
    int *ptr1;

    if (NULL == (ptr = (int *)aligned_alloc(alignment,
                                             sizeof(int)))) {
        /* Handle error */
    }

    if (NULL == (ptr1 = (int *)aligned_alloc(alignment,
                                              resize))) {
        /* Handle error */
    }

    if (NULL == (memcpy(ptr1, ptr, sizeof(int)))) {
        /* Handle error */
    }

    free(ptr);
}
```

Compliant Solution (Windows)

Windows defines the `_aligned_malloc()` function to allocate memory on a specified alignment boundary. The `_aligned_realloc()` [MSDN] can be used to change the size of this memory. This compliant solution demonstrates one such usage:

```
#include <malloc.h>

void func(void) {
    size_t alignment = 1 << 12;
    int *ptr;
    int *ptr1;

    /* Original allocation */
    if (NULL == (ptr = (int *)_aligned_malloc(sizeof(int),
                                                alignment))) {
        /* Handle error */
    }

    /* Reallocation */
    if (NULL == (ptr1 = (int *)_aligned_realloc(ptr, 1024,
                                                alignment))) {
        _aligned_free(ptr);
        /* Handle error */
    }

    _aligned_free(ptr1);
}
```

The `size` and `alignment` arguments for `_aligned_malloc()` are provided in reverse order of the C Standard `aligned_alloc()` function.

Risk Assessment

Improper alignment can lead to arbitrary memory locations being accessed and written to.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
MEM36-C	Low	Probable	High	P2	L3

Bibliography

[ISO/IEC 9899:2011]	7.22.3.1, "The <code>aligned_alloc</code> Function"
[MSDN]	<code>aligned_malloc()</code>

Excerpt from SEI CERT C++ Coding Standard Wiki [https://www.securecoding.cert.org/confluence/x/4YEzAg], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
<code>aligned_allocators</code>	Functions that allocate aligned memory.	{'aligned_alloc', '_aligned_malloc', 'posix_memalign'}
<code>level</code>	Grouping of priorities into different levels	3
<code>likelihood</code>	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
<code>priority</code>	Priority based on the combination of severity, likelihood and remediation cost	2
<code>realloc_functions</code>	Functions that reallocate without allowing for aligned memory. The first argument must be the pointer to reallocate.	{'realloc',}
<code>recommendation</code>	Whether this check is classified as a recommendation or rule	False
<code>remediation_cost</code>	How expensive is it to comply with the rule?	high

Possible Messages

Name	Message
<code>realloc_aligned</code>	Do not modify the alignment of objects by calling <code>realloc()</code> .

CertC++-MEM50

Do not access freed memory.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

Evaluating a pointer—including dereferencing the pointer, using it as an operand of an arithmetic operation, type casting it, and using it as the right-hand side of an assignment—into memory that has been deallocated by a memory management function is [undefined behavior](#). Pointers to memory that has been deallocated are called *dangling pointers*. Accessing a dangling pointer can result in exploitable [vulnerabilities](#).

It is at the memory manager's discretion when to reallocate or recycle the freed memory. When memory is freed, all pointers into it become invalid, and its contents might either be returned to the operating system, making the freed space inaccessible, or remain intact and accessible. As a result, the data at the freed location can appear to be valid but change unexpectedly. Consequently, memory must not be written to or read from once it is freed.

Noncompliant Code Example (`new` and `delete`)

In this noncompliant code example, `s` is dereferenced after it has been deallocated. If this access results in a write-after-free, the [vulnerability](#) can be [exploited](#) to run arbitrary code with the permissions of the vulnerable process. Typically, dynamic memory allocations and deallocations are far removed, making it difficult to recognize and diagnose such problems.

```
#include <new>

struct S {
    void f();
};

void g() noexcept(false) {
    S *s = new S;
    // ...
    delete s;
    // ...
    s->f();
}
```

The function `g()` is marked `noexcept(false)` to comply with [MEM52-CPP. Detect and handle memory allocation errors](#).

Compliant Solution (`new` and `delete`)

In this compliant solution, the dynamically allocated memory is not deallocated until it is no longer required.

```
#include <new>

struct S {
    void f();
};

void g() noexcept(false) {
    S *s = new S;
    // ...
    s->f();
    delete s;
}
```

Compliant Solution (Automatic Storage Duration)

When possible, use automatic storage duration instead of dynamic storage duration. Since `s` is not required to live beyond the scope of `g()`, this compliant solution uses automatic storage duration to limit the lifetime of `s` to the scope of `g()`.

```
struct S {
    void f();
};

void g() {
    S s;
    // ...
    s.f();
}
```

Noncompliant Code Example (`std::unique_ptr`)

In the following noncompliant code example, the dynamically allocated memory managed by the `buff` object is accessed after it has been implicitly deallocated by the object's destructor.

```
#include <iostream>
#include <memory>
#include <cstring>

int main(int argc, const char *argv[]) {
    const char *s = "";
    if (argc > 1) {
        enum { BufferSize = 32 };
        try {
            std::unique_ptr<char[]> buff(new char[BufferSize]);
            std::memset(buff.get(), 0, BufferSize);
            // ...
            s = std::strncpy(buff.get(), argv[1], BufferSize - 1);
        } catch (std::bad_alloc &) {
            // Handle error
        }
    }
}
```

```
    std::cout << s << std::endl;
}
```

This code always creates a null-terminated byte string, despite its use of `strncpy()`, because it leaves the final `char` in the buffer set to 0.

Compliant Solution (`std::unique_ptr`)

In this compliant solution, the lifetime of the `buff` object extends past the point at which the memory managed by the object is accessed.

```
#include <iostream>
#include <memory>
#include <cstring>

int main(int argc, const char *argv[]) {
    std::unique_ptr<char[]> buff;
    const char *s = "";

    if (argc > 1) {
        enum { BufferSize = 32 };
        try {
            buff.reset(new char[BufferSize]);
            std::memset(buff.get(), 0, BufferSize);
            // ...
            s = std::strncpy(buff.get(), argv[1], BufferSize - 1);
        } catch (std::bad_alloc &) {
            // Handle error
        }
    }

    std::cout << s << std::endl;
}
```

Compliant Solution

In this compliant solution, a variable with automatic storage duration of type `std::string` is used in place of the `std::unique_ptr<char[]>`, which reduces the complexity and improves the security of the solution.

```
#include <iostream>
#include <string>

int main(int argc, const char *argv[]) {
    std::string str;

    if (argc > 1) {
        str = argv[1];
    }

    std::cout << str << std::endl;
}
```

Noncompliant Code Example (`std::string::c_str()`)

In this noncompliant code example, `std::string::c_str()` is being called on a temporary `std::string` object. The resulting pointer will point to released memory once the `std::string` object is destroyed at the end of the assignment expression, resulting in [undefined behavior](#) when accessing elements of that pointer.

```
#include <string>

std::string str_func();
void display_string(const char *);

void f() {
    const char *str = str_func().c_str();
    display_string(str); /* Undefined behavior */
}
```

Compliant solution (`std::string::c_str()`)

In this compliant solution, a local copy of the string returned by `str_func()` is made to ensure that string `str` will be valid when the call to `display_string()` is made.

```
#include <string>

std::string str_func();
void display_string(const char *s);

void f() {
    std::string str = str_func();
    const char *cstr = str.c_str();
    display_string(cstr); /* ok */
}
```

Noncompliant Code Example

In this noncompliant code example, an attempt is made to allocate zero bytes of memory through a call to `operator new()`. If this request succeeds, `operator new()` is required to return a non-null pointer value. However, according to the C++ Standard, [basic.stc.dynamic.allocation], paragraph 2 [[ISO/IEC 14882-2014](#)], attempting to dereference memory through such a pointer results in [undefined behavior](#).

```
#include <new>
```

```

void f() noexcept(false) {
    unsigned char *ptr = static_cast<unsigned char *>(::operator new(0));
    *ptr = 0;
    // ...
    ::operator delete(ptr);
}

```

Compliant Solution

The compliant solution depends on programmer intent. If the programmer intends to allocate a single `unsigned char` object, the compliant solution is to use `new` instead of a direct call to `operator new()`, as this compliant solution demonstrates.

```

void f() noexcept(false) {
    unsigned char *ptr = new unsigned char;
    *ptr = 0;
    // ...
    delete ptr;
}

```

Compliant Solution

If the programmer intends to allocate zero bytes of memory (perhaps to obtain a unique pointer value that cannot be reused by any other pointer in the program until it is properly released), then instead of attempting to dereference the resulting pointer, the recommended solution is to declare `ptr` as a `void *`, which cannot be dereferenced by a [conforming implementation](#).

```

#include <new>

void f() noexcept(false) {
    void *ptr = ::operator new(0);
    // ...
    ::operator delete(ptr);
}

```

Risk Assessment

Reading previously dynamically allocated memory after it has been deallocated can lead to [abnormal program termination](#) and [denial-of-service attacks](#). Writing memory that has been deallocated can lead to the execution of arbitrary code with the permissions of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM50-CPP	High	Likely	Medium	P18	L1

Related Guidelines

SEI CERT C++ Coding Standard	EXP54-CPP. Do not access an object outside of its lifetime MEM52-CPP. Detect and handle memory allocation errors
SEI CERT C Coding Standard	MEM30-C. Do not access freed memory
MITRE CWE	CWE-415 , Double Free CWE-416 , Use After Free

Bibliography

[ISO/IEC 14882-2014]	Subclause 3.7.4.1, "Allocation Functions" Subclause 3.7.4.2, "Deallocation Functions"
[Seacord 2013b]	Chapter 4, "Dynamic Memory Management"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)

Possible Messages

Name	Message
possible_use_after_free	Dynamic memory possibly used after it was previously released
use_after_free	Dynamic memory used after it was previously released

CertC++-MEM51

Properly deallocate dynamically allocated resources.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

The C programming language provides several ways to allocate memory, such as `std::malloc()`, `std::calloc()`, and `std::realloc()`, which can be used by a C++ program. However, the C programming language defines only a single way to free the allocated memory: `std::free()`. See [MEM31-C. Free dynamically allocated memory when no longer needed](#) and [MEM34-C. Only free memory allocated dynamically](#) for rules specifically regarding C allocation and deallocation requirements.

The C++ programming language adds additional ways to allocate memory, such as the operators `new`, `new[]`, and placement `new`, and [allocator objects](#). Unlike C, C++ provides multiple ways to free dynamically allocated memory, such as the operators `delete`, `delete[]`, and deallocation functions on allocator objects.

Do not call a deallocation function on anything other than `nullptr`, or a pointer returned by the corresponding allocation function described by the following.

Allocator	Deallocator
<code>global operator new()/new</code>	<code>global operator delete()/delete</code>
<code>global operator new[]()/new[]</code>	<code>global operator delete[]()/delete[]</code>
<code>class-specific operator new()/new</code>	<code>class-specific operator delete()/delete</code>
<code>class-specific operator new[]()/new[]</code>	<code>class-specific operator delete[]()/delete[]</code>
<code>placement operator new()</code>	N/A
<code>allocator<T>::allocate()</code>	<code>allocator<T>::deallocate()</code>
<code>std::malloc()</code> , <code>std::calloc()</code> , <code>std::realloc()</code>	<code>std::free()</code>
<code>std::get_temporary_buffer()</code>	<code>std::return_temporary_buffer()</code>

Passing a pointer value to an inappropriate deallocation function can result in [undefined behavior](#).

The C++ Standard, [expr.delete], paragraph 2 [[ISO/IEC 14882-2014](#)], in part, states the following:

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 [1.8] 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*. If not, the behavior is undefined.

Deallocating a pointer that is not allocated dynamically (including non-dynamic pointers returned from calls to placement `new()`) is undefined behavior because the pointer value was not obtained by an allocation function. Deallocating a pointer that has already been passed to a deallocation function is undefined behavior because the pointer value no longer points to memory that has been dynamically allocated.

When an operator such as `new` is called, it results in a call to an overloadable operator of the same name, such as `operator new()`. These overloadable functions can be called directly but carry the same restrictions as their operator counterparts. That is, calling `operator delete()` and passing a pointer parameter has the same constraints as calling the `delete` operator on that pointer. Further, the overloads are subject to scope resolution, so it is possible (but not permissible) to call a class-specific operator to allocate an object but a global operator to deallocate the object.

See [MEM53-CPP. Explicitly construct and destruct objects when manually managing object lifetime](#) for information on lifetime management of objects when using memory management functions other than the `new` and `delete` operators.

Noncompliant Code Example (placement `new()`)

In this noncompliant code example, the local variable `s1` is passed as the expression to the placement `new` operator. The resulting pointer of that call is then passed to `::operator delete()`, resulting in [undefined behavior](#) due to `::operator delete()` attempting to free memory that was not returned by `::operator new()`.

```
#include <iostream>

struct S {
    S() { std::cout << "S::S()" << std::endl; }
    ~S() { std::cout << "S::~S()" << std::endl; }
};

void f() {
    S s1;
    S *s2 = new (&s1) S;
    // ...
    delete s2;
}
```

Compliant Solution (placement `new()`)

This compliant solution removes the call to `::operator delete()`, allowing `s1` to be destroyed as a result of its normal object lifetime termination.

```
#include <iostream>

struct S {
    S() { std::cout << "S::S()" << std::endl; }
    ~S() { std::cout << "S::~S()" << std::endl; }
};

void f() {
    S s1;
    S *s2 = new (&s1) S;
    // ...
}
```

Noncompliant Code Example (Uninitialized `delete`)

In this noncompliant code example, two allocations are attempted within the same `try` block, and if either fails, the `catch` handler attempts to free resources that have been allocated. However, because the pointer variables have not been initialized to a known value, a failure to allocate memory for `i1` may result in passing `::operator delete()` a value (in `i2`) that was not previously returned by a call to `::operator new()`, resulting in [undefined behavior](#).

```
#include <new>

void f() {
    int *i1, *i2;
    try {
        i1 = new int;
        i2 = new int;
    } catch (std::bad_alloc &) {
        delete i1;
        delete i2;
    }
}
```

Compliant Solution (Uninitialized `delete`)

This compliant solution initializes both pointer values to `nullptr`, which is a valid value to pass to `::operator delete()`.

```
#include <new>

void f() {
    int *i1 = nullptr, *i2 = nullptr;
    try {
        i1 = new int;
        i2 = new int;
    } catch (std::bad_alloc &) {
        delete i1;
        delete i2;
    }
}
```

Noncompliant Code Example (Double-Free)

Once a pointer is passed to the proper deallocation function, that pointer value is invalidated. Passing the pointer to a deallocation function a second time when the pointer value has not been returned by a subsequent call to an allocation function results in an attempt to free memory that has not been allocated dynamically. The underlying data structures that manage the heap can become corrupted in a way that can introduce security [vulnerabilities](#) into a program.

These types of issues are called *double-free vulnerabilities*. In practice, double-free vulnerabilities can be [exploited](#) to execute arbitrary code.

In this noncompliant code example, the class `C` is given ownership of a `P *`, which is subsequently deleted by the class destructor. The C++ Standard, [class.copy], paragraph 7 [[ISO/IEC 14882-2014](#)], states the following:

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 (8.4). The latter case is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor.

Despite the presence of a user-declared destructor, `C` will have an implicitly defaulted copy constructor defined for it, and this defaulted copy constructor will copy the pointer value stored in `p`, resulting in a double-free: the first free happens when `g()` exits and the second free happens when `h()` exits.

```
struct P {};

class C {
    P *p;

public:
    C(P *p) : p(p) {}
    ~C() { delete p; }

    void f() {}

};

void g(C c) {
    c.f();
}

void h() {
    P *p = new P;
    C c(p);
    g(c);
}
```

Compliant Solution (Double-Free)

In this compliant solution, the copy constructor and copy assignment operator for `C` are explicitly deleted. This deletion would result in an [ill-formed](#) program with the definition of `g()` from the preceding noncompliant code example due to use of the deleted copy constructor. Consequently, `g()` was modified to accept its parameter by reference, removing the double-free.

```
struct P {};

class C {
    P *p;

public:
    C(P *p) : p(p) {}
    C(const C&) = delete;
    ~C() { delete p; }

    void operator=(const C&) = delete;

    void f() {}

};

void g(C &c) {
    c.f();
}

void h() {
    P *p = new P;
    C c(p);
    g(c);
}
```

Noncompliant Code Example (array `new[]`)

In the following noncompliant code example, an array is allocated with `array new[]` but is deallocated with a scalar `delete` call instead of an array `delete[]` call, resulting in [undefined behavior](#).

```
void f() {
    int *array = new int[10];
    // ...
    delete array;
}
```

Compliant Solution (array `new[]`)

In the compliant solution, the code is fixed by replacing the call to `delete` with a call to `delete []` to adhere to the correct pairing of memory allocation and deallocation functions.

```
void f() {
    int *array = new int[10];
    // ...
    delete[] array;
}
```

Noncompliant Code Example (`malloc()`)

In this noncompliant code example, the call to `malloc()` is mixed with a call to `delete`.

```
#include <cstdlib>
void f() {
    int *i = static_cast<int *>(std::malloc(sizeof(int)));
    // ...
    delete i;
}
```

This code does not violate [MEM53-CPP. Explicitly construct and destruct objects when manually managing object lifetime](#) because it complies with the MEM53-CPP-EX1 exception.

Implementation Details

Some implementations of `::operator new()` result in calling `std::malloc()`. On such implementations, the `::operator delete()` function is required to call `std::free()` to deallocate the pointer, and the noncompliant code example would behave in a well-defined manner. However, this is an [implementation detail](#) and should not be relied on—implementations are under no obligation to use underlying C memory management functions to implement C++ memory management operators.

Compliant Solution (`malloc()`)

In this compliant solution, the pointer allocated by `std::malloc()` is deallocated by a call to `std::free()` instead of `delete`.

```
#include <cstdlib>
void f() {
    int *i = static_cast<int *>(std::malloc(sizeof(int)));
    // ...
    std::free(i);
}
```

Noncompliant Code Example (`new`)

In this noncompliant code example, `std::free()` is called to deallocate memory that was allocated by `new`. A common side effect of the [undefined behavior](#) caused by using the incorrect deallocation function is that destructors will not be called for the object being deallocated by `std::free()`.

```
#include <cstdlib>
struct S {
    ~S();
};

void f() {
    S *s = new S();
    // ...
    std::free(s);
}
```

Additionally, this code violates [MEM53-CPP. Explicitly construct and destruct objects when manually managing object lifetime](#).

Compliant Solution (`new`)

In this compliant solution, the pointer allocated by `new` is deallocated by calling `delete` instead of `std::free()`.

```
struct S {
    ~S();
};

void f() {
    S *s = new S();
    // ...
    delete s;
}
```

Noncompliant Code Example (Class `new`)

In this noncompliant code example, the global `new` operator is overridden by a class-specific implementation of `operator new()`. When `new` is called, the class-specific override is selected, so `S::operator new()` is called. However, because the object is destroyed with a scoped `::delete` operator, the global `operator delete()` function is called instead of the class-specific implementation `S::operator delete()`, resulting in [undefined behavior](#).

```
#include <cstdlib>
#include <new>

struct S {
    static void *operator new(std::size_t size) noexcept(true) {
        return std::malloc(size);
    }

    static void operator delete(void *ptr) noexcept(true) {
        std::free(ptr);
    }
};

void f() {
    S *s = new S;
    ::delete s;
}
```

Compliant Solution (class `new`)

In this compliant solution, the scoped `::delete` call is replaced by a nonscoped `delete` call, resulting in `s::operator delete()` being called.

```

#include <cstdlib>
#include <new>

struct S {
    static void *operator new(std::size_t size) noexcept(true) {
        return std::malloc(size);
    }

    static void operator delete(void *ptr) noexcept(true) {
        std::free(ptr);
    }
};

void f() {
    S *s = new S;
    delete s;
}

```

Noncompliant Code Example (`std::unique_ptr`)

In this noncompliant code example, a `std::unique_ptr` is declared to hold a pointer to an object, but is direct-initialized with an array of objects. When the `std::unique_ptr` is destroyed, its default deleter calls `delete` instead of `delete[]`, resulting in undefined behavior.

```

#include <memory>

struct S {};

void f() {
    std::unique_ptr<S> s{new S[10]};
}

```

Compliant Solution (`std::unique_ptr`)

In this compliant solution, the `std::unique_ptr` is declared to hold an array of objects instead of a pointer to an object. Additionally, `std::make_unique()` is used to initialize the smart pointer.

```

#include <memory>

struct S {};

void f() {
    std::unique_ptr<S[]> s = std::make_unique<S[]>(10);
}

```

Use of `std::make_unique()` instead of direct initialization will emit a diagnostic if the resulting `std::unique_ptr` is not of the correct type. Had it been used in the noncompliant code example, the result would have been an ill-formed program instead of undefined behavior. It is best to use `std::make_unique()` instead of manual initialization by other means.

Noncompliant Code Example (`std::shared_ptr`)

In this noncompliant code example, a `std::shared_ptr` is declared to hold a pointer to an object, but is direct-initialized with an array of objects. As with `std::unique_ptr`, when the `std::shared_ptr` is destroyed, its default deleter calls `delete` instead of `delete[]`, resulting in undefined behavior.

```

#include <memory>

struct S {};

void f() {
    std::shared_ptr<S> s{new S[10]};
}

```

Compliant Solution (`std::shared_ptr`)

Unlike the compliant solution for `std::unique_ptr`, where `std::make_unique()` is called to create a unique pointer to an array, it is ill-formed to call `std::make_shared()` with an array type. Instead, this compliant solution manually specifies a custom deleter for the shared pointer type, ensuring that the underlying array is properly deleted.

```

#include <memory>

struct S {};

void f() {
    std::shared_ptr<S> s{new S[10], [](const S *ptr) { delete [] ptr; }};
}

```

Risk Assessment

Passing a pointer value to a deallocation function that was not previously obtained by the matching allocation function results in [undefined behavior](#), which can lead to exploitable [vulnerabilities](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM51-CPP	High	Likely	Medium	P18	L1

Related Guidelines

SEI CERT C++ Coding Standard	MEM53-CPP. Explicitly construct and destruct objects when manually managing object lifetime
SEI CERT C Coding Standard	MEM31-C. Free dynamically allocated memory when no longer needed MEM34-C. Only free memory allocated dynamically
MITRE CWE	CWE 590 , Free of Memory Not on the Heap CWE 415 , Double Free CWE 404 , Improper Resource Shutdown or Release CWE 762 , Mismatched Memory Management Routines

Bibliography

[Dowd 2007]	"Attacking <code>delete</code> and <code>delete []</code> in C++"
[Henricson 1997]	Rule 8.1, " <code>delete</code> should only be used with <code>new</code> " Rule 8.2, " <code>delete []</code> should only be used with <code>new []</code> "
[ISO/IEC 14882-2014]	Subclause 5.3.5, "Delete" Subclause 12.8, "Copying and Moving Class Objects" Subclause 18.6.1, "Storage Allocation and Deallocation" Subclause 20.7.11, "Temporary Buffers"
[Meyers 2005]	Item 16, "Use the Same Form in Corresponding Uses of <code>new</code> and <code>delete</code> "
[Seacord 2013]	Chapter 4, "Dynamic Memory Management"
[Viega 2005]	"Doubly Freeing Memory"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM51-CPP.+Properly+deallocate+dynamically+allocated+resources>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)

Possible Messages

Name	Message
possible_wrong_release	Resource possibly released using wrong function [allocation used {0}]
wrong_release	Resource released using wrong function [allocation used {0}]

CertC++-FI030

Exclude user input from format strings.

Input: IR

Source languages: C++

Details

Never call a formatted I/O function with a format string containing a [tainted value](#). An attacker who can fully or partially control the contents of a format string can crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location. Consequently, the attacker can execute arbitrary code with the permissions of the vulnerable process [[Seacord 2013b](#)]. Formatted output functions are particularly dangerous because many programmers are unaware of their capabilities. For example, formatted output functions can be used to write an integer value to a specified address using the `%n` conversion specifier.

Noncompliant Code Example

The `incorrect_password()` function in this noncompliant code example is called during identification and authentication to display an error message if the

specified user is not found or the password is incorrect. The function accepts the name of the user as a string referenced by `user`. This is an exemplar of [untrusted data](#) that originates from an unauthenticated user. The function constructs an error message that is then output to `stderr` using the C Standard `fprintf()` function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void incorrect_password(const char *user) {
    int ret;
    /* User names are restricted to 256 or fewer characters */
    static const char msg_format[] = "%s cannot be authenticated.\n";
    size_t len = strlen(user) + sizeof(msg_format);
    char *msg = (char *)malloc(len);
    if (msg == NULL) {
        /* Handle error */
    }
    ret = snprintf(msg, len, msg_format, user);
    if (ret < 0) {
        /* Handle error */
    } else if (ret >= len) {
        /* Handle truncated output */
    }
    fprintf(stderr, msg);
    free(msg);
}
```

The `incorrect_password()` function calculates the size of the message, allocates dynamic storage, and then constructs the message in the allocated memory using the `snprintf()` function. The addition operations are not checked for integer overflow because the string referenced by `user` is known to have a length of 256 or less. Because the `%s` characters are replaced by the string referenced by `user` in the call to `snprintf()`, the resulting string needs 1 byte less than is allocated. The `snprintf()` function is commonly used for messages that are displayed in multiple locations or messages that are difficult to build. However, the resulting code contains a format-string [vulnerability](#) because the `msg` includes untrusted user input and is passed as the format-string argument in the call to `fprintf()`.

Compliant Solution (`fputs()`)

This compliant solution fixes the problem by replacing the `fprintf()` call with a call to `fputs()`, which outputs `msg` directly to `stderr` without evaluating its contents:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void incorrect_password(const char *user) {
    int ret;
    /* User names are restricted to 256 or fewer characters */
    static const char msg_format[] = "%s cannot be authenticated.\n";
    size_t len = strlen(user) + sizeof(msg_format);
    char *msg = (char *)malloc(len);
    if (msg == NULL) {
        /* Handle error */
    }
    ret = snprintf(msg, len, msg_format, user);
    if (ret < 0) {
        /* Handle error */
    } else if (ret >= len) {
        /* Handle truncated output */
    }
    fputs(msg, stderr);
    free(msg);
}
```

Compliant Solution (`fprintf()`)

This compliant solution passes the untrusted user input as one of the variadic arguments to `fprintf()` and not as part of the format string, eliminating the possibility of a format-string vulnerability:

```
#include <stdio.h>

void incorrect_password(const char *user) {
    static const char msg_format[] = "%s cannot be authenticated.\n";
    fprintf(stderr, msg_format, user);
}
```

Noncompliant Code Example (POSIX)

This noncompliant code example is similar to the first noncompliant code example but uses the POSIX function `syslog()` [[IEEE Std 1003.1:2013](#)] instead of the `fprintf()` function. The `syslog()` function is also susceptible to format-string vulnerabilities.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <syslog.h>

void incorrect_password(const char *user) {
    int ret;
    /* User names are restricted to 256 or fewer characters */
    static const char msg_format[] = "%s cannot be authenticated.\n";
    size_t len = strlen(user) + sizeof(msg_format);
    char *msg = (char *)malloc(len);
    if (msg == NULL) {
        /* Handle error */
    }
```

```

ret = sprintf(msg, len, msg_format, user);
if (ret < 0) {
    /* Handle error */
} else if (ret >= len) {
    /* Handle truncated output */
}
syslog(LOG_INFO, msg);
free(msg);
}

```

The `syslog()` function first appeared in BSD 4.2 and is supported by Linux and other modern UNIX implementations. It is not available on Windows systems.

Compliant Solution (POSIX)

This compliant solution passes the untrusted user input as one of the variadic arguments to `syslog()` instead of including it in the format string:

```

#include <syslog.h>

void incorrect_password(const char *user) {
    static const char msg_format[] = "%s cannot be authenticated.\n";
    syslog(LOG_INFO, msg_format, user);
}

```

Risk Assessment

Failing to exclude user input from format specifiers may allow an attacker to crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location and consequently execute arbitrary code with the permissions of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO30-C	High	Likely	Medium	P18	L1

Related Guidelines

[Key here](#) (explains table format and definitions)

Taxonomy	Taxonomy item	Relationship
CERT Oracle Secure Coding Standard for Java	IDS06-J. Exclude unsanitized user input from format strings	Prior to 2018-01-12: CERT: Unspecified Relationship
CERT Perl Secure Coding Standard	IDS30-PL. Exclude user input from format strings	Prior to 2018-01-12: CERT: Unspecified Relationship
ISO/IEC TR 24772:2013	Injection [RST]	Prior to 2018-01-12: CERT: Unspecified Relationship
ISO/IEC TS 17961:2013	Including tainted or out-of-domain input in a format string [usrfmt]	Prior to 2018-01-12: CERT: Unspecified Relationship
CWE 2.11	CWE-134 , Uncontrolled Format String	2017-05-16: CERT: Exact
CWE 2.11	CWE-20 , Improper Input Validation	2017-05-17: CERT: Rule subset of CWE

Bibliography

[IEEE Std 1003.1:2013]	XSH, System Interfaces, <code>syslog</code>
[Seacord 2013b]	Chapter 6, "Formatted Output"
[Viega 2005]	Section 5.2.23, "Format String Problem"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/c/FIO30-C.+Exclude+user+input+from+format+strings>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
forbidden_one_argument_calls		['printf', 'wprintf']
forbidden_two_argument_calls		['fprintf', 'fwprintf', 'syslog']
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
tainted_string	Exclude user input from format strings.

CertC++-FI034

Distinguish between characters read from a file and EOF or WEOF.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

The `EOF` macro represents a negative value that is used to indicate that the file is exhausted and no data remains when reading data from a file. `EOF` is an example of an [in-band error indicator](#). In-band error indicators are problematic to work with, and the creation of new in-band-error indicators is discouraged by [ERR02-C. Avoid in-band error indicators](#).

The byte I/O functions `fgetc()`, `getc()`, and `getchar()` all read a character from a stream and return it as an `int`. (See [STR00-C. Represent characters using an appropriate type](#).) If the stream is at the end of the file, the end-of-file indicator for the stream is set and the function returns `EOF`. If a read error occurs, the error indicator for the stream is set and the function returns `EOF`. If these functions succeed, they cast the character returned into an `unsigned char`.

Because `EOF` is negative, it should not match any `unsigned char` value. However, this is only true for [implementations](#) where the `int` type is wider than `char`. On an implementation where `int` and `char` have the same width, a character-reading function can read and return a valid character that has the same bit-pattern as `EOF`. This could occur, for example, if an attacker inserted a value that looked like `EOF` into the file or data stream to alter the behavior of the program.

The C Standard requires only that the `int` type be able to represent a maximum value of +32767 and that a `char` type be no larger than an `int`. Although uncommon, this situation can result in the integer constant expression `EOF` being indistinguishable from a valid character; that is, `(int)(unsigned char)65535 == -1`. Consequently, failing to use `feof()` and `ferror()` to detect end-of-file and file errors can result in incorrectly identifying the `EOF` character on rare implementations where `sizeof(int) == sizeof(char)`.

This problem is much more common when reading wide characters. The `fgetwc()`, `getwc()`, and `getwchar()` functions return a value of type `wint_t`. This value can represent the next wide character read, or it can represent `WEOF`, which indicates end-of-file for wide character streams. On most implementations, the `wchar_t` type has the same width as `wint_t`, and these functions can return a character indistinguishable from `WEOF`.

In the UTF-16 character set, `0xFFFF` is guaranteed not to be a character, which allows `WEOF` to be represented as the value `-1`. Similarly, all UTF-32 characters are positive when viewed as a signed 32-bit integer. All widely used character sets are designed with at least one value that does not represent a character. Consequently, it would require a custom character set designed without consideration of the C programming language for this problem to occur with wide characters or with ordinary characters that are as wide as `int`.

The C Standard `feof()` and `ferror()` functions are not subject to the problems associated with character and integer sizes and should be used to verify end-of-file and file errors for susceptible implementations [[Kettlewell 2002](#)]. Calling both functions on each iteration of a loop adds significant overhead, so a good strategy is to temporarily trust `EOF` and `WEOF` within the loop but verify them with `feof()` and `ferror()` following the loop.

Noncompliant Code Example

This noncompliant code example loops while the character `c` is not `EOF`:

```
#include <stdio.h>

void func(void) {
    int c;

    do {
        c = getchar();
    } while (c != EOF);
}
```

Although `EOF` is guaranteed to be negative and distinct from the value of any unsigned character, it is not guaranteed to be different from any such value when converted to an `int`. Consequently, when `int` has the same width as `char`, this loop may terminate prematurely.

Compliant Solution (Portable)

This compliant solution uses `feof()` to test for end-of-file and `ferror()` to test for errors:

```
#include <stdio.h>

void func(void) {
    int c;

    do {
        c = getchar();
    } while (c != EOF);
    if (feof(stdin)) {
        /* Handle end of file */
    } else if (ferror(stdin)) {
        /* Handle file error */
    } else {
        /* Received a character that resembles EOF; handle error */
    }
}
```

Noncompliant Code Example (Nonportable)

This noncompliant code example uses an assertion to ensure that the code is executed only on architectures where `int` is wider than `char` and `EOF` is guaranteed not to be a valid character value. However, this code example is noncompliant because the variable `c` is declared as a `char` rather than an `int`, making it possible for a valid character value to compare equal to the value of the `EOF` macro when `char` is signed because of sign extension:

```
#include <assert.h>
#include <limits.h>
#include <stdio.h>

void func(void) {
    char c;
    static_assert(UCHAR_MAX < UINT_MAX, "FIO34-C violation");

    do {
        c = getchar();
    } while (c != EOF);
}
```

Assuming that a `char` is a signed 8-bit type and an `int` is a 32-bit type, if `getchar()` returns the character value '`\xFF`' (decimal 255), it will be interpreted as `EOF` because this value is sign-extended to `0xFFFFFFFF` (the value of `EOF`) to perform the comparison. (See [STR34-C. Cast characters to unsigned char before converting to larger integer sizes.](#))

Compliant Solution (Nonportable)

This compliant solution declares `c` to be an `int`. Consequently, the loop will terminate only when the file is exhausted.

```
#include <assert.h>
#include <stdio.h>
#include <limits.h>

void func(void) {
    int c;
    static_assert(UCHAR_MAX < UINT_MAX, "FIO34-C violation");

    do {
        c = getchar();
    } while (c != EOF);
}
```

Noncompliant Code Example (Wide Characters)

In this noncompliant example, the result of the call to the C standard library function `getwc()` is stored into a variable of type `wchar_t` and is subsequently compared with `WEOF`:

```
#include <stddef.h>
#include <stdio.h>
#include <wchar.h>

enum { BUFFER_SIZE = 32 };

void g(void) {
    wchar_t buf[BUFFER_SIZE];
    wchar_t wc;
    size_t i = 0;

    while ((wc = getwc(stdin)) != L'\n' && wc != WEOF) {
        if (i < (BUFFER_SIZE - 1)) {
            buf[i++] = wc;
        }
        buf[i] = L'\0';
    }
}
```

This code suffers from two problems. First, the value returned by `getwc()` is immediately converted to `wchar_t` before being compared with `WEOF`. Second,

there is no check to ensure that `wint_t` is wider than `wchar_t`. Both of these problems make it possible for an attacker to terminate the loop prematurely by supplying the wide-character value matching `WEOF` in the file.

Compliant Solution (Portable)

This compliant solution declares `c` to be a `wint_t` to match the integer type returned by `getwc()`. Furthermore, it does not rely on `WEOF` to determine end-of-file definitively.

```
#include <stddef.h>
#include <stdio.h>
#include <wchar.h>

enum {BUFFER_SIZE = 32 }

void g(void) {
    wchar_t buf[BUFFER_SIZE];
    wint_t wc;
    size_t i = 0;

    while ((wc = getwc(stdin)) != L'\n' && wc != WEOF) {
        if (i < BUFFER_SIZE - 1) {
            buf[i++] = wc;
        }
    }

    if (feof(stdin) || ferror(stdin)) {
        buf[i] = L'\0';
    } else {
        /* Received a wide character that resembles WEOF; handle error */
    }
}
```

Exceptions

F1034-C-EX1: A number of C functions do not return characters but can return `EOF` as a status code. These functions include `fclose()`, `fflush()`, `fputs()`, `fscanf()`, `puts()`, `scanf()`, `sscanf()`, `vfscanf()`, and `vscanf()`. These return values can be compared to `EOF` without validating the result.

Risk Assessment

Incorrectly assuming characters from a file cannot match `EOF` or `WEOF` has resulted in significant vulnerabilities, including command injection attacks. [See the [*CA-1996-22](#) advisory.]

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
F1034-C	High	Probable	Medium	P12	L1

Related Guidelines

CERT C Secure Coding Standard	STR00-C. Represent characters using an appropriate type INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data
CERT Oracle Secure Coding Standard for Java	F1008-J. Use an int to capture the return value of methods that read a character or byte
ISO/IEC TS 17961:2013	Using character values that are indistinguishable from EOF [chreof]

Bibliography

[Kettlewell 2002]	Section 1.2, "<stdio.h> and Character Types"
[NIST 2006]	SAMATE Reference Dataset Test Case ID 000-000-088
[Summit 2005]	Question 12.2

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/dwGKBw>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
functions_under_test	Function which comparison with EOF/WEOF might be unsafe	['fgetc', 'getc', 'getchar', 'fgetwc', 'getwc', 'getwchar']
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
limit_header_files	Header files which declare following constants: UCHAR_MAX, UINT_MAX, WCHAR_MAX, WINT_MAX.	['limits.h']
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
unsafe_eof	Distinguish between characters read from a file and EOF or WEOF.

CertC++-FI037

Do not assume that fgets() or fgetws() returns a nonempty string when successful.

Input: IR

Source languages: C++

Details

Errors can occur when incorrect assumptions are made about the type of data being read. These assumptions may be violated, for example, when binary data has been read from a file instead of text from a user's terminal or the output of a process is piped to `stdin`. (See [FI014-C. Understand the difference between text mode and binary mode with file streams](#).) On some systems, it may also be possible to input a null byte (as well as other binary codes) from the keyboard.

Subclause 7.21.7.2 of the C Standard [[ISO/IEC 9899:2011](#)] says,

The `fgets` function returns `s` if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned.

The wide-character function `fgetws()` has the same behavior. Therefore, if `fgets()` or `fgetws()` returns a non-null pointer, it is safe to assume that the array contains data. However, it is erroneous to assume that the array contains a nonempty string because the data may contain null characters.

Noncompliant Code Example

This noncompliant code example attempts to remove the trailing newline (`\n`) from an input line. The `fgets()` function is typically used to read a newline-terminated line of input from a stream. It takes a size parameter for the destination buffer and copies, at most, `size - 1` characters from a stream to a character array.

```
#include <stdio.h>
#include <string.h>

enum { BUFFER_SIZE = 1024 };

void func(void) {
    char buf[BUFFER_SIZE];

    if (fgets(buf, sizeof(buf), stdin) == NULL) {
        /* Handle error */
    }
    buf[strlen(buf) - 1] = '\0';
}
```

The `strlen()` function computes the length of a string by determining the number of characters that precede the terminating null character. A problem occurs if the first character read from the input by `fgets()` happens to be a null character. This may occur, for example, if a binary data file is read by the `fgets()` call [[Lai 2006](#)]. If the first character in `buf` is a null character, `strlen(buf)` returns 0, the expression `strlen(buf) - 1` wraps around to a large positive value, and a write-outside-array-bounds error occurs.

Compliant Solution

This compliant solution uses `strchr()` to replace the newline character in the string if it exists:

```
#include <stdio.h>
#include <string.h>

enum { BUFFER_SIZE = 1024 };
```

```

void func(void) {
    char buf[BUFFER_SIZE];
    char *p;

    if (fgets(buf, sizeof(buf), stdin)) {
        p = strchr(buf, '\n');
        if (p) {
            *p = '\0';
        }
    } else {
        /* Handle error */
    }
}

```

Risk Assessment

Incorrectly assuming that character data has been read can result in an out-of-bounds memory write or other flawed logic.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
F1037-C	High	Probable	Medium	P12	L1

Related Guidelines

CERT C Secure Coding Standard	F1014-C. Understand the difference between text mode and binary mode with file streams F1020-C. Avoid unintentional truncation when using fgets() or fgetws()
MITRE CWE	CWE-119 , Improper Restriction of Operations within the Bounds of a Memory Buffer CWE-123 , Write-what-where Condition CWE-125 , Out-of-bounds Read CWE-241 , Improper Handling of Unexpected Data Type

Bibliography

[ISO/IEC 9899:2011]	Subclause 7.21.7.2, "The fgets Function" Subclause 7.29.3.2, "The fgetws Function"
[Lai 2006]	
[Seacord 2013]	Chapter 2, "Strings"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/dh>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
strlen_functions	Functions which test for empty strings.	['strlen']

Possible Messages

Name	Message
empty_test	Do not assume that fgets() or fgetws() returns a nonempty string when successful.

CertC++-F1038

Do not copy a FILE object.

Input: IR

Source languages: C++

Details

According to the C Standard, 7.21.3, paragraph 6 [[ISO/IEC 9899:2011](#)],

The address of the `FILE` object used to control a stream may be significant; a copy of a `FILE` object need not serve in place of the original.

Consequently, do not copy a `FILE` object.

Noncompliant Code Example

This noncompliant code example can fail because a by-value copy of `stdout` is being used in the call to `fputs()`:

```
#include <stdio.h>

int main(void) {
    FILE my_stdout = *stdout;
    if (fputs("Hello, World!\n", &my_stdout) == EOF) {
        /* Handle error */
    }
    return 0;
}
```

When compiled under Microsoft Visual Studio 2013 and run on Windows, this noncompliant example results in an "access violation" at runtime.

Compliant Solution

In this compliant solution, a copy of the `stdout` pointer to the `FILE` object is used in the call to `fputs()`:

```
#include <stdio.h>

int main(void) {
    FILE *my_stdout = stdout;
    if (fputs("Hello, World!\n", my_stdout) == EOF) {
        /* Handle error */
    }
    return 0;
}
```

Risk Assessment

Using a copy of a `FILE` object in place of the original may result in a crash, which can be used in a [denial-of-service attack](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO38-C	Low	Probable	Medium	P4	L3

Related Guidelines

ISO/IEC TS 17961:2013	Copying a <code>FILE</code> object [filecpy]
---------------------------------------	--

Bibliography

ISO/IEC 9899:2011	7.21.3, "Files"
-----------------------------------	-----------------

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/wAw>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
file_pointer_dereference	A pointer to a <code>FILE</code> object shall not be dereferenced

CertC++-FIO47

Use valid format strings.

Input: IR

Source languages: C++

Details

The formatted output functions (`fprintf()` and related functions) convert, format, and print their arguments under control of a *format string*. The C Standard, 7.21.6.1, paragraph 3 [[ISO/IEC 9899:2011](#)], specifies

The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

Each *conversion specification* is introduced by the % character followed (in order) by

- Zero or more *flags* (in any order), which modify the meaning of the conversion specification
- An optional minimum field *width*
- An optional *precision* that gives the minimum number of digits to appear for certain conversion specifiers
- An optional *length modifier* that specifies the size of the argument
- A *conversion specifier character* that indicates the type of conversion to be applied

Common mistakes in creating format strings include

- Providing an incorrect number of arguments for the format string
- Using invalid conversion specifiers
- Using a flag character that is incompatible with the conversion specifier
- Using a length modifier that is incompatible with the conversion specifier
- Mismatching the argument type and conversion specifier
- Using an argument of type other than `int` for *width* or *precision*

The following table summarizes the compliance of various conversion specifications. The first column contains one or more conversion specifier characters. The next four columns consider the combination of the specifier characters with the various flags (the apostrophe ['], -, +, the space character, #, and 0). The next eight columns consider the combination of the specifier characters with the various length modifiers (h, hh, l, ll, j, z, t, and L).

Valid combinations are marked with a type name; arguments matched with the conversion specification are interpreted as that type. For example, an argument matched with the specifier %hd is interpreted as a `short`, so short appears in the cell where d and h intersect. The last column denotes the expected types of arguments matched with the original specifier characters.

Valid and meaningful combinations are marked by the ✓ symbol (save for the length modifier columns, as described previously). Valid combinations that have no effect are labeled N/E. Using a combination marked by the ✘ symbol, using a specification not represented in the table, or using an argument of an unexpected type is [undefined behavior](#). (See undefined behaviors [153](#), [155](#), [157](#), [158](#), [161](#), and [162](#).)

Conversion Specifier Character	<i>XSI</i>	- + SPACE	#	0	h	hh	l	ll	j	z	t	L	Argument Type
d, i	✓	✓	✗	✓	short	signed char	long	long long	intmax_t	size_t	ptrdiff_t	✗	Signed integer
o	✗	✓	✓	✓	unsigned short	unsigned char	unsigned long	unsigned long long	uintmax_t	size_t	ptrdiff_t	✗	Unsigned integer
u	✓	✓	✗	✓	unsigned short	unsigned char	unsigned long	unsigned long long	uintmax_t	size_t	ptrdiff_t	✗	Unsigned integer
x, X	✗	✓	✓	✓	unsigned short	unsigned char	unsigned long	unsigned long long	uintmax_t	size_t	ptrdiff_t	✗	Unsigned integer
f, F	✓	✓	✓	✓	✗	✗	N/E	N/E	✗	✗	✗	long double	double or long double
e, E	✗	✓	✓	✓	✗	✗	N/E	N/E	✗	✗	✗	long double	double or long double
g, G	✓	✓	✓	✓	✗	✗	N/E	N/E	✗	✗	✗	long double	double or long double
a, A	✓	✓	✓	✓	✗	✗	N/E	N/E	✗	✗	✗	long double	double or long double
c	✗	✓	✗	✗	✗	✗	wint_t	✗	✗	✗	✗	✗	int or wint_t
s	✗	✓	✗	✗	✗	✗	NTWS	✗	✗	✗	✗	✗	NTBS or NTWS
p	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	void*
n	✗	✓	✗	✗	short*	char*	long*	long long*	intmax_t*	size_t*	ptrdiff_t*	✗	Pointer to integer
c XSI	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	wint_t
s XSI	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	NTWS
%	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	None

SPACE: The space (" ") character

N/E: No effect

NTBS: `char*` argument pointing to a null-terminated character string

NTWS: `wchar_t*` argument pointing to a null-terminated wide character string

XSI: [ISO/IEC 9945-2003](#) XSI extension

The formatted input functions (`fscanf()` and related functions) use similarly specified format strings and impose similar restrictions on their format strings and arguments.

Do not supply an unknown or invalid conversion specification or an invalid combination of flag character, precision, length modifier, or conversion specifier to a formatted IO function. Likewise, do not provide a number or type of argument that does not match the argument type of the conversion specifier used in the format string.

Format strings are usually string literals specified at the call site, but they need not be. However, they should not contain [tainted values](#). (See [F1030-C. Exclude user input from format strings](#) for more information.)

Noncompliant Code Example

Mismatches between arguments and conversion specifications may result in [undefined behavior](#). Compilers may diagnose type mismatches in formatted output function invocations. In this noncompliant code example, the `error_type` argument to `printf()` is incorrectly matched with the `s` specifier rather than with the `a` specifier. Likewise, the `error_msg` argument is incorrectly matched with the `d` specifier instead of the `s` specifier. These usages result in [undefined behavior](#). One possible result of this invocation is that `printf()` will interpret the `error_type` argument as a pointer and try to read a string from the address that `error_type` contains, possibly resulting in an access violation.

```
#include <stdio.h>

void func(void) {
    const char *error_msg = "Resource not available to user.";
    int error_type = 3;
    /* ... */
}
```

```

    printf("Error (type %s): %d\n", error_type, error_msg);
    /* ... */
}

```

Compliant Solution

This compliant solution ensures that the arguments to the `printf()` function match their respective conversion specifications:

```
#include <stdio.h>

void func(void) {
    const char *error_msg = "Resource not available to user.";
    int error_type = 3;
    /* ... */
    printf("Error (type %d): %s\n", error_type, error_msg);

    /* ... */
}
```

Risk Assessment

Incorrectly specified format strings can result in memory corruption or [abnormal program termination](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FI047-C	High	Unlikely	Medium	P6	L2

Related Guidelines

SEI CERT C++ Coding Standard	FI000-CPP. Take care when creating format strings
ISO/IEC TS 17961:2013	Using invalid format strings [invfmtstr]
MITRE CWE	CWE-686 , Function Call with Incorrect Argument Type

Bibliography

[ISO/IEC 9899:2011]	Subclause 7.21.6.1, "The <code>fprintf</code> Function"
-------------------------------------	---

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/wQA1>], Copyright [C] 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
allow_extra_args	Whether to allow additional arguments that are not used by the format string.	False
allow_gnu_extensions	Whether to allow the GNU extensions to format specifications.	True
allow_unknown_specs	Whether to allow unknown format specifications. It may be necessary to set this option when using implementation-specific extensions. Arguments are not checked when the format string contains unknown format specifications.	False
functions	A dictionary mapping the names of the functions to check, to a triple {function_kind, fmt_param_index, arg_start_index} where function_kind is either 'printf' or 'scanf', fmt_param_index is the index of the format-string parameter, and arg_start_index is the index of the first variadic argument.	dict(...)
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
arg_type_mismatch	{ } expects argument of type '{ }', but argument { } has type '{ }'
buffer_too_small	{ } may write up to { } characters to buffer of size { }. (disabled)
invalid_conversion	Invalid or non-standard conversion specification
matching_arg_expected	{ } expects a matching '{ }' argument
precision_for_conversion	Precision must not be used with %{ } conversion specifier
too_many_args	Too many arguments for format.
unknown_buffer_size	Potential buffer overflow: { } used with buffer of unknown size. (disabled)
unlimited_read	Potential buffer overflow: { } has no limit on amount of characters read. (disabled)
unsupported_assignment_suppression	%n does not support assignment suppression
unsupported_field_width	%n does not support field width
unsupported_flags	%n does not support flags
unsupported_flags_modifiers	Cannot use any flags or modifiers with '%%'
unsupported_hash	%{ } does not support the '#' flag
unsupported_i_flag	%{ } does not support the 'l' flag
unsupported_length_modifier	%{ } does not support the '{ }' length modifier
unsupported_tick	%{ } does not support the "" flag
unsupported_zero	%{ } does not support the '0' flag

CertC++-ERR30

Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

The value of `errno` is initialized to zero at program startup, but it is never subsequently set to zero by any C standard library function. The value of `errno` may be set to nonzero by a C standard library function call whether or not there is an error, provided the use of `errno` is not documented in the description of the function. It is meaningful for a program to inspect the contents of `errno` only after an error might have occurred. More precisely, `errno` is meaningful only after a library function that sets `errno` on error has returned an error code.

According to Question 20.4 of C-FAQ [[Summit 2005](#)].

In general, you should detect errors by checking return values, and use `errno` only to distinguish among the various causes of an error, such as "File not found" or "Permission denied." (Typically, you use `perror` or `strerror` to print these discriminating error messages.) It's only necessary to detect errors with `errno` when a function does not have a unique, unambiguous, out-of-band error return (that is, because all of its possible return values are valid; one example is `atoi` [sic]). In these cases (and in these cases only; check the documentation to be sure whether a function allows this), you can detect errors by setting `errno` to 0, calling the function, and then testing `errno`. (Setting `errno` to 0 first is important, as no library function ever does that for you.)

Note that `atoi()` is not required to set the value of `errno`.

Library functions fall into the following categories:

- Those that set `errno` and return and [out-of-band error indicator](#)
- Those that set `errno` and return and [in-band error indicator](#)
- Those that do not promise to set `errno`
- Those with differing standards documentation

Library Functions that Set `errno` and Return an Out-of-Band Error Indicator

The C Standard specifies that the functions listed in the following table set `errno` and return an [out-of-band error indicator](#). That is, their return value on error can never be returned by a successful call.

A program may set and check `errno` for these library functions but is not required to do so. The program should not check the value of `errno` without first verifying that the function returned an error indicator. For example, `errno` should not be checked after calling `signal()` without first ensuring that `signal()` actually returned `SIG_ERR`.

Functions That Set `errno` and Return an Out-of-Band Error Indicator

Function Name	Return Value	<code>errno</code> Value
<code>fgetpos()</code> , <code>fsetpos()</code>	-1L	Positive
<code>mbrtowc()</code> , <code>mbsrtowcs()</code>	(<code>size_t</code>) (-1)	EILSEQ
<code>signal()</code>	SIG_ERR	Positive
<code>wcrtomb()</code> , <code>wcsrtombs()</code>	(<code>size_t</code>) (-1)	EILSEQ
<code>mbrtoc16()</code> , <code>mbrtoc32()</code>	(<code>size_t</code>) (-1)	EILSEQ
<code>c16rtomb()</code> , <code>cr32rtomb()</code>	(<code>size_t</code>) (-1)	EILSEQ

Library Functions that Set `errno` and Return an In-Band Error Indicator

The C Standard specifies that the functions listed in the following table set `errno` and return an [in-band error indicator](#). That is, the return value when an error occurs is also a valid return value for successful calls. For example, the `strtoul()` function returns `ULONG_MAX` and sets `errno` to `ERANGE` if an error occurs. Because `ULONG_MAX` is a valid return value, `errno` must be used to check whether an error actually occurred. A program that uses `errno` for error checking must set it to 0 before calling one of these library functions and then inspect `errno` before a subsequent library function call.

The `fgetwc()` and `fputwc()` functions return `WEOF` in multiple cases, only one of which results in setting `errno`. The string conversion functions will return the maximum or minimum representable value and set `errno` to `ERANGE` if the converted value cannot be represented by the data type. However, if the conversion cannot happen because the input is invalid, the function will return 0, and the output pointer parameter will be assigned the value of the input pointer parameter, provided the output parameter is non-null.

Functions that Set `errno` and Return an In-Band Error Indicator

Function Name	Return Value	<code>errno</code> Value
<code>fgetwc()</code> , <code>fputwc()</code>	<code>WEOF</code>	EILSEQ
<code>strtol()</code> , <code>wcstol()</code>	<code>LONG_MIN</code> OR <code>LONG_MAX</code>	ERANGE
<code>strtoll()</code> , <code>wcstoll()</code>	<code>LLONG_MIN</code> OR <code>LLONG_MAX</code>	ERANGE
<code>strtoul()</code> , <code>wcstoul()</code>	<code>ULONG_MAX</code>	ERANGE
<code>strtoull()</code> , <code>wcstoull()</code>	<code>ULLONG_MAX</code>	ERANGE
<code>strtoumax()</code> , <code>wcstoumax()</code>	<code>UINTMAX_MAX</code>	ERANGE
<code>strtod()</code> , <code>wcstod()</code>	0 OR <code>±HUGE_VAL</code>	ERANGE
<code>strtof()</code> , <code>wcstof()</code>	0 OR <code>±HUGE_VALF</code>	ERANGE
<code>strtold()</code> , <code>wcstold()</code>	0 OR <code>±HUGE_VALL</code>	ERANGE
<code>strtoimax()</code> , <code>wcstoimax()</code>	<code>INTMAX_MIN</code> , <code>INTMAX_MAX</code>	ERANGE

Library Functions that Do Not Promise to Set `errno`

The C Standard fails to document the behavior of `errno` for some functions. For example, the `setlocale()` function normally returns a null pointer in the event of an error, but no guarantees are made about setting `errno`.

After calling one of these functions, a program should not rely solely on the value of `errno` to determine if an error occurred. The function might have altered `errno`, but this does not ensure that `errno` will properly indicate an error condition.

Library Functions with Differing Standards Documentation

Some functions behave differently regarding `errno` in various standards. The `fopen()` function is one such example. When `fopen()` encounters an error, it returns a null pointer. The C Standard makes no mention of `errno` when describing `fopen()`. However, POSIX.1 declares that when `fopen()` encounters an error, it returns a null pointer and sets `errno` to a value indicating the error [[IEEE Std 1003.1-2013](#)]. The implication is that a program conforming to C but not to POSIX (such as a Windows program) should not check `errno` after calling `fopen()`, but a POSIX program may check `errno` if `fopen()` returns a null pointer.

Library Functions and `errno`

The following uses of `errno` are documented in the C Standard:

- Functions defined in `<complex.h>` may set `errno` but are not required to.
- For numeric conversion functions in the `strtod`, `strtol`, `wcstod`, and `wcstol` families, if the correct result is outside the range of representable values, an appropriate minimum or maximum value is returned and the value `ERANGE` is stored in `errno`. For floating-point conversion functions in the `strtod` and `wcstod` families, if an underflow occurs, whether `errno` acquires the value `ERANGE` is [implementation-defined](#). If the conversion fails, 0 is returned

- and `errno` is not set.
- The numeric conversion function `atof()` and those in the `atoi` family "need not affect the value of" `errno`.
- For mathematical functions in `<math.h>`, if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, on a domain error, `errno` acquires the value `EDOM`; on an overflow with default rounding or if the mathematical result is an exact infinity from finite arguments, `errno` acquires the value `ERANGE`; and on an underflow, whether `errno` acquires the value `ERANGE` is implementation-defined.
- If a request made by calling `signal()` cannot be honored, a value of `SIG_ERR` is returned and a positive value is stored in `errno`.
- The byte I/O functions, wide-character I/O functions, and multibyte conversion functions store the value of the macro `EILSEQ` in `errno` if and only if an encoding error occurs.
- On failure, `fgetpos()` and `fsetpos()` return nonzero and store an [implementation-defined](#) positive value in `errno`.
- On failure, `ftell()` returns `-1L` and stores an [implementation-defined](#) positive value in `errno`.
- The `perror()` function maps the error number in `errno` to a message and writes it to `stderr`.

The POSIX.1 standard defines the use of `errno` by many more functions (including the C standard library function). POSIX also has a small set of functions that are exceptions to the rule. These functions have no return value reserved to indicate an error, but they still set `errno` on error. To detect an error, an application must set `errno` to 0 before calling the function and check whether it is nonzero after the call. Affected functions include `strcoll()`, `strxfrm()`, `strerror()`, `wcscoll()`, `wcsxfrm()`, and `fwide()`. The C Standard allows these functions to set `errno` to a nonzero value on success. Consequently, this type of error checking should be performed only on POSIX systems.

Noncompliant Code Example (`strtoul()`)

This noncompliant code example fails to set `errno` to 0 before invoking `strtoul()`. If an error occurs, `strtoul()` returns a valid value (`ULONG_MAX`), so `errno` is the only means of determining if `strtoul()` ran successfully.

```
#include <errno.h>
#include <limits.h>
#include <stdlib.h>

void func(const char *c_str) {
    unsigned long number;
    char *endptr;

    number = strtoul(c_str, &endptr, 0);
    if (endptr == c_str || (number == ULONG_MAX
                           && errno == ERANGE)) {
        /* Handle error */
    } else {
        /* Computation succeeded */
    }
}
```

Any error detected in this manner may have occurred earlier in the program or may not represent an actual error.

Compliant Solution (`strtoul()`)

This compliant solution sets `errno` to 0 before the call to `strtoul()` and inspects `errno` after the call:

```
#include <errno.h>
#include <limits.h>
#include <stdlib.h>

void func(const char *c_str) {
    unsigned long number;
    char *endptr;

    errno = 0;
    number = strtoul(c_str, &endptr, 0);
    if (endptr == c_str || (number == ULONG_MAX
                           && errno == ERANGE)) {
        /* Handle error */
    } else {
        /* Computation succeeded */
    }
}
```

Noncompliant Code Example (`fopen()`)

This noncompliant code example may fail to diagnose errors because `fopen()` might not set `errno` even if an error occurs:

```
#include <errno.h>
#include <stdio.h>

void func(const char *filename) {
    FILE *fileptr;

    errno = 0;
    fileptr = fopen(filename, "rb");
    if (errno != 0) {
        /* Handle error */
    }
}
```

Compliant Solution (`fopen()`, C)

The C Standard makes no mention of `errno` when describing `fopen()`. In this compliant solution, the results of the call to `fopen()` are used to determine failure and `errno` is not checked:

```
#include <stdio.h>
```

```

void func(const char *filename) {
    FILE *fileptr = fopen(filename, "rb");
    if (fileptr == NULL) {
        /* An error occurred in fopen() */
    }
}

```

Compliant Solution (`fopen()`, POSIX)

In this compliant solution, `errno` is checked only after an error has already been detected by another means:

```

#include <errno.h>
#include <stdio.h>

void func(const char *filename) {
    FILE *fileptr;

    errno = 0;
    fileptr = fopen(filename, "rb");
    if (fileptr == NULL) {
        /*
         * An error occurred in fopen(); now it's valid
         * to examine errno.
         */
        perror(filename);
    }
}

```

Risk Assessment

The improper use of `errno` may result in failing to detect an error condition or in incorrectly identifying an error condition when none exists.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR30-C	Medium	Probable	Medium	P8	L2

Related Guidelines

CERT C Secure Coding Standard	EXP12-C. Do not ignore values returned by functions
ISO/IEC TS 17961:2013	Incorrectly setting and using <code>errno</code> [<code>inverrno</code>]
MITRE CWE	CWE-456 , Missing Initialization of a Variable

Bibliography

[Brainbell.com]	Macros and Miscellaneous Pitfalls
[Horton 1990]	Section 11, p. 168 Section 14, p. 254
[IEEE Std 1003.1-2013]	XSH, System Interfaces, <code>fopen</code>
[Koenig 1989]	Section 5.4, p. 73
[Summit 2005]	

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/KwBL>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
errno_clearers		[]
errno_readers		['perror']
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
standard_posix		False
variant_cert		True

Possible Messages

Name	Message
misplaced_errno_check	errno should only be tested after a call to an errno-setting function.
missing_error_handling	The program should not check the value of errno without first verifying that the function returned an error indicator.
uncleared_errno	errno should be set to zero before calling an errno-setting function.

CertC++-ERR32

Do not rely on indeterminate values of errno.

Input: IR

Source languages: C++

Details

According to the C Standard [ISO/IEC 9899:2011], the behavior of a program is [undefined](#) when

the value of `errno` is referred to after a signal occurred other than as the result of calling the `abort` or `raise` function and the corresponding signal handler obtained a `SIG_ERR` return from a call to the `signal` function.

See [undefined behavior 133](#).

A signal handler is allowed to call `signal()`; if that fails, `signal()` returns `SIG_ERR` and sets `errno` to a positive value. However, if the event that caused a signal was external (not the result of the program calling `abort()` or `raise()`), the only functions the signal handler may call are `_Exit()` or `abort()`, or it may call `signal()` on the signal currently being handled; if `signal()` fails, the value of `errno` is [indeterminate](#).

This rule is also a special case of [SIG31-C. Do not access shared objects in signal handlers](#). The object designated by `errno` is of static storage duration and is not a volatile `sig_atomic_t`. As a result, performing any action that would require `errno` to be set would normally cause [undefined behavior](#). The C Standard, 7.14.1.1, paragraph 5, makes a special exception for `errno` in this case, allowing `errno` to take on an indeterminate value but specifying that there is no other [undefined behavior](#). This special exception makes it possible to call `signal()` from within a signal handler without risking [undefined behavior](#), but the handler, and any code executed after the handler returns, must not depend on the value of `errno` being meaningful.

Noncompliant Code Example

The `handler()` function in this noncompliant code example attempts to restore default handling for the signal indicated by `signum`. If the request to set the signal to default can be honored, the `signal()` function returns the value of the signal handler for the most recent successful call to the `signal()` function for the specified signal. Otherwise, a value of `SIG_ERR` is returned and a positive value is stored in `errno`. Unfortunately, the value of `errno` is indeterminate because the `handler()` function is called when an external signal is raised, so any attempt to read `errno` (for example, by the `perror()` function) is [undefined behavior](#):

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

typedef void (*pfv)(int);

void handler(int signum) {
    pfv old_handler = signal(signum, SIG_DFL);
    if (old_handler == SIG_ERR) {
        perror("SIGINT handler"); /* Undefined behavior */
        /* Handle error */
    }
}

int main(void) {
    pfv old_handler = signal(SIGINT, handler);
    if (old_handler == SIG_ERR) {
```

```

    perror("SIGINT handler");
    /* Handle error */
}

/* Main code loop */

return EXIT_SUCCESS;
}

```

The call to `perror()` from `handler()` also violates [SIG30-C. Call only asynchronous-safe functions within signal handlers](#).

Compliant Solution

This compliant solution does not reference `errno` and does not return from the signal handler if the `signal()` call fails:

```

#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

typedef void (*pfv)(int);

void handler(int signum) {
    pfv old_handler = signal(signum, SIG_DFL);
    if (old_handler == SIG_ERR) {
        abort();
    }
}

int main(void) {
    pfv old_handler = signal(SIGINT, handler);
    if (old_handler == SIG_ERR) {
        perror("SIGINT handler");
        /* Handle error */
    }

    /* Main code loop */

    return EXIT_SUCCESS;
}

```

Noncompliant Code Example (POSIX)

POSIX is less restrictive than C about what applications can do in signal handlers. It has a long list of [asynchronous-safe](#) functions that can be called. (See [SIG30-C. Call only asynchronous-safe functions within signal handlers](#).) Many of these functions set `errno` on error, which can lead to a signal handler being executed between a call to a failed function and the subsequent inspection of `errno`. Consequently, the value inspected is not the one set by that function but the one set by a function call in the signal handler. POSIX applications can avoid this problem by ensuring that signal handlers containing code that might alter `errno`; always save the value of `errno` on entry and restore it before returning.

The signal handler in this noncompliant code example alters the value of `errno`. As a result, it can cause incorrect error handling if executed between a failed function call and the subsequent inspection of `errno`:

```

#include <signal.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>

void reaper(int signum) {
    errno = 0;
    for (;;) {
        int rc = waitpid(-1, NULL, WNOHANG);
        if ((0 == rc) || (-1 == rc && EINTR != errno)) {
            break;
        }
    }
    if (ECHILD != errno) {
        /* Handle error */
    }
}

int main(void) {
    struct sigaction act;
    act.sa_handler = reaper;
    act.sa_flags = 0;
    if (sigemptyset(&act.sa_mask) != 0) {
        /* Handle error */
    }
    if (sigaction(SIGCHLD, &act, NULL) != 0) {
        /* Handle error */
    }

    /* ... */

    return EXIT_SUCCESS;
}

```

Compliant Solution (POSIX)

This compliant solution saves and restores the value of `errno` in the signal handler:

```

#include <signal.h>
#include <stdlib.h>
#include <errno.h>

```

```

#include <sys/wait.h>

void reaper(int signum) {
    errno_t save_errno = errno;
    errno = 0;
    for (;;) {
        int rc = waitpid(-1, NULL, WNOHANG);
        if ((0 == rc) || (-1 == rc && EINTR != errno)) {
            break;
        }
    }
    if (ECHILD != errno) {
        /* Handle error */
    }
    errno = save_errno;
}

int main(void) {
    struct sigaction act;
    act.sa_handler = reaper;
    act.sa_flags = 0;
    if (sigemptyset(&act.sa_mask) != 0) {
        /* Handle error */
    }
    if (sigaction(SIGCHLD, &act, NULL) != 0) {
        /* Handle error */
    }

    /* ... */

    return EXIT_SUCCESS;
}

```

Risk Assessment

Referencing indeterminate values of `errno` is [undefined behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR32-C	Low	Unlikely	Low	P3	L3

Related Guidelines

CERT C Secure Coding Standard	SIG30-C. Call only asynchronous-safe functions within signal handlers SIG31-C. Do not access shared objects in signal handlers
---	---

Bibliography

[ISO/IEC 9899:2011]	Subclause 7.14.1.1, "The <code>signal</code> Function"
-------------------------------------	--

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/NABL>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
<code>errno_readers</code>		[' perror']
<code>level</code>	Grouping of priorities into different levels	3
<code>likelihood</code>	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
<code>priority</code>	Priority based on the combination of severity, likelihood and remediation cost	3
<code>recommendation</code>	Whether this check is classified as a recommendation or rule	False
<code>remediation_cost</code>	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
<code>errno_referred_in_signal_handler</code>	Do not rely on <code>errno</code> value in signal handler.

CertC++-ERR33

Detect and handle standard library errors.

Input: IR

Source languages: C++

Details

The majority of the standard library functions, including I/O functions and memory allocation functions, return either a valid value or a value of the correct return type that indicates an error (for example, -1 or a null pointer). Assuming that all calls to such functions will succeed and failing to check the return value for an indication of an error is a dangerous practice that may lead to [unexpected](#) or [undefined behavior](#) when an error occurs. It is essential that programs detect and appropriately handle all errors in accordance with an error-handling policy, as discussed in [ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy](#).

The successful completion or failure of each of the standard library functions listed in the following table shall be determined either by comparing the function's return value with the value listed in the column labeled "Error Return" or by calling one of the library functions mentioned in the footnotes.

Standard Library Functions

Function	Successful Return	Error Return
aligned_alloc()	Pointer to space	NULL
asctime_s()	0	Nonzero
at_quick_exit()	0	Nonzero
atexit()	0	Nonzero
bsearch()	Pointer to matching element	NULL
bsearch_s()	Pointer to matching element	NULL
btowc()	Converted wide character	WEOF
c16rtomb()	Number of bytes	(size_t) (-1)
c32rtomb()	Number of bytes	(size_t) (-1)
calloc()	Pointer to space	NULL
clock()	Processor time	(clock_t) (-1)
cond_broadcast()	thrd_success	thrd_error
cond_init()	thrd_success	thrd_nomem OR thrd_error
cond_signal()	thrd_success	thrd_error
cond_timedwait()	thrd_success	thrd_timeout OR thrd_error
cond_wait()	thrd_success	thrd_error
ctime_s()	0	Nonzero
fclose()	0	EOF (negative)
fflush()	0	EOF (negative)
fgetc()	Character read	EOF ¹
fgetpos()	0	Nonzero, errno > 0
fgets()	Pointer to string	NULL
fgetwc()	Wide character read	WEOF ¹
fopen()	Pointer to stream	NULL
fopen_s()	0	Nonzero
fprintf()	Number of characters (nonnegative)	Negative
fprintf_s()	Number of characters (nonnegative)	Negative
fputc()	Character written	EOF ²
fputs()	Nonnegative	EOF (negative)
fputwc()	Wide character written	WEOF
fputws()	Nonnegative	EOF (negative)

<code>fread()</code>	Elements read	Elements read
<code>freopen()</code>	Pointer to stream	NULL
<code>freopen_s()</code>	0	Nonzero
<code>fscanf()</code>	Number of conversions (nonnegative)	<code>EOF</code> (negative)
<code>fscanf_s()</code>	Number of conversions (nonnegative)	<code>EOF</code> (negative)
<code>fseek()</code>	0	Nonzero
<code>fsetpos()</code>	0	Nonzero, <code>errno > 0</code>
<code>ftell()</code>	File position	<code>-1L, errno > 0</code>
<code>fwprintf()</code>	Number of wide characters (nonnegative)	Negative
<code>fwprintf_s()</code>	Number of wide characters (nonnegative)	Negative
<code>fwrite()</code>	Elements written	Elements written
<code>fwscanf()</code>	Number of conversions (nonnegative)	<code>EOF</code> (negative)
<code>fwscanf_s()</code>	Number of conversions (nonnegative)	<code>EOF</code> (negative)
<code>getc()</code>	Character read	<code>EOF</code> ¹
<code>getchar()</code>	Character read	<code>EOF</code> ¹
<code>getenv()</code>	Pointer to string	NULL
<code>getenv_s()</code>	Pointer to string	NULL
<code>gets_s()</code>	Pointer to string	NULL
<code>getwc()</code>	Wide character read	<code>WEOF</code>
<code>getwchar()</code>	Wide character read	<code>WEOF</code>
<code>gmtime()</code>	Pointer to broken-down time	NULL
<code>gmtime_s()</code>	Pointer to broken-down time	NULL
<code>localtime()</code>	Pointer to broken-down time	NULL
<code>localtime_s()</code>	Pointer to broken-down time	NULL
<code>malloc()</code>	Pointer to space	NULL
<code>mblen(), s != NULL</code>	Number of bytes	-1
<code>mbrlen(), s != NULL</code>	Number of bytes or status	<code>(size_t) (-1)</code>
<code>mbrtoc16()</code>	Number of bytes or status	<code>(size_t) (-1), errno == EILSEQ</code>
<code>mbrtoc32()</code>	Number of bytes or status	<code>(size_t) (-1), errno == EILSEQ</code>
<code>mbrtowc(), s != NULL</code>	Number of bytes or status	<code>(size_t) (-1), errno == EILSEQ</code>
<code>mbsrtowcs()</code>	Number of non-null elements	<code>(size_t) (-1), errno == EILSEQ</code>
<code>mbsrtowcs_s()</code>	0	Nonzero
<code>mbstowcs()</code>	Number of non-null elements	<code>(size_t) (-1)</code>
<code>mbstowcs_s()</code>	0	Nonzero
<code>mbtowc(), s != NULL</code>	Number of bytes	-1
<code>memchr()</code>	Pointer to located character	NULL
<code>mktime()</code>	Calendar time	<code>(time_t) (-1)</code>
<code>mtx_init()</code>	<code>thrd_success</code>	<code>thrd_error</code>

<code>mtx_lock()</code>	<code>thrd_success</code>	<code>thrd_error</code>
<code>mtx_timedlock()</code>	<code>thrd_success</code>	<code>thrd_timedout OR thrd_error</code>
<code>mtx_trylock()</code>	<code>thrd_success</code>	<code>thrd_busy OR thrd_error</code>
<code>mtx_unlock()</code>	<code>thrd_success</code>	<code>thrd_error</code>
<code>printf_s()</code>	Number of characters (nonnegative)	Negative
<code>putc()</code>	Character written	<code>EOF</code> ²
<code>putwc()</code>	Wide character written	<code>WEOF</code>
<code>raise()</code>	0	Nonzero
<code>realloc()</code>	Pointer to space	<code>NULL</code>
<code>remove()</code>	0	Nonzero
<code>rename()</code>	0	Nonzero
<code>setlocale()</code>	Pointer to string	<code>NULL</code>
<code>setvbuf()</code>	0	Nonzero
<code>scanf()</code>	Number of conversions (nonnegative)	<code>EOF</code> (negative)
<code>scanf_s()</code>	Number of conversions (nonnegative)	<code>EOF</code> (negative)
<code>signal()</code>	Pointer to previous function	<code>SIG_ERR, errno > 0</code>
<code>snprintf()</code>	Number of characters that would be written (nonnegative)	Negative
<code>snprintf_s()</code>	Number of characters that would be written (nonnegative)	Negative
<code>sprintf()</code>	Number of non-null characters written	Negative
<code>sprintf_s()</code>	Number of non-null characters written	Negative
<code>sscanf()</code>	Number of conversions (nonnegative)	<code>EOF</code> (negative)
<code>sscanf_s()</code>	Number of conversions (nonnegative)	<code>EOF</code> (negative)
<code>strchr()</code>	Pointer to located character	<code>NULL</code>
<code>strerror_s()</code>	0	Nonzero
<code>strftime()</code>	Number of non-null characters	0
<code>strupr()</code>	Pointer to located character	<code>NULL</code>
<code> strrchr()</code>	Pointer to located character	<code>NULL</code>
<code>strstr()</code>	Pointer to located string	<code>NULL</code>
<code>strtod()</code>	Converted value	<code>0, errno == ERANGE</code>
<code>strtof()</code>	Converted value	<code>0, errno == ERANGE</code>
<code>strtoimax()</code>	Converted value	<code>INTMAX_MAX OR INTMAX_MIN, errno == ERANGE</code>
<code>strtok()</code>	Pointer to first character of a token	<code>NULL</code>
<code>strtok_s()</code>	Pointer to first character of a token	<code>NULL</code>
<code>strtol()</code>	Converted value	<code>LONG_MAX OR LONG_MIN, errno == ERANGE</code>
<code>strtold()</code>	Converted value	<code>0, errno == ERANGE</code>
<code>strtoll()</code>	Converted value	<code>LLONG_MAX OR LLONG_MIN, errno == ERANGE</code>
<code>strtoumax()</code>	Converted value	<code>UINTMAX_MAX, errno == ERANGE</code>
<code>strtoul()</code>	Converted value	<code>ULONG_MAX, errno == ERANGE</code>

<code>strtoull()</code>	Converted value	<code>ULLONG_MAX, errno == ERANGE</code>
<code>strxfrm()</code>	Length of transformed string	<code>>= n</code>
<code>swprintf()</code>	Number of non-null wide characters	Negative
<code>swprintf_s()</code>	Number of non-null wide characters	Negative
<code>swscanf()</code>	Number of conversions {nonnegative}	<code>EOF {negative}</code>
<code>swscanf_s()</code>	Number of conversions {nonnegative}	<code>EOF {negative}</code>
<code>thrd_create()</code>	<code>thrd_success</code>	<code>thrd_nomem OR thrd_error</code>
<code>thrd_detach()</code>	<code>thrd_success</code>	<code>thrd_error</code>
<code>thrd_join()</code>	<code>thrd_success</code>	<code>thrd_error</code>
<code>thrd_sleep()</code>	0	Negative
<code>time()</code>	Calendar time	<code>(time_t) (-1)</code>
<code>timespec_get()</code>	Base	0
<code>tmpfile()</code>	Pointer to stream	<code>NULL</code>
<code>tmpfile_s()</code>	0	Nonzero
<code>tmpnam()</code>	Non-null pointer	<code>NULL</code>
<code>tmpnam_s()</code>	0	Nonzero
<code>tss_create()</code>	<code>thrd_success</code>	<code>thrd_error</code>
<code>tss_get()</code>	Value of thread-specific storage	0
<code>tss_set()</code>	<code>thrd_success</code>	<code>thrd_error</code>
<code>ungetc()</code>	Character pushed back	<code>EOF {see below}</code>
<code>ungetwc()</code>	Character pushed back	<code>WEOF</code>
<code>vfprintf()</code>	Number of characters {nonnegative}	Negative
<code>vfprintf_s()</code>	Number of characters {nonnegative}	Negative
<code>vfscanf()</code>	Number of conversions {nonnegative}	<code>EOF {negative}</code>
<code>vfscanf_s()</code>	Number of conversions {nonnegative}	<code>EOF {negative}</code>
<code>vfwprintf()</code>	Number of wide characters {nonnegative}	Negative
<code>vfwprintf_s()</code>	Number of wide characters {nonnegative}	Negative
<code>vfwscanf()</code>	Number of conversions {nonnegative}	<code>EOF {negative}</code>
<code>vfwscanf_s()</code>	Number of conversions {nonnegative}	<code>EOF {negative}</code>
<code>vprintf_s()</code>	Number of characters {nonnegative}	Negative
<code>vscanf()</code>	Number of conversions {nonnegative}	<code>EOF {negative}</code>
<code>vscanf_s()</code>	Number of conversions {nonnegative}	<code>EOF {negative}</code>
<code>vsnprintf()</code>	Number of characters that would be written {nonnegative}	Negative
<code>vsnprintf_s()</code>	Number of characters that would be written {nonnegative}	Negative
<code>vsprintf()</code>	Number of non-null characters {nonnegative}	Negative
<code>vsprintf_s()</code>	Number of non-null characters {nonnegative}	Negative
<code>vsscanf()</code>	Number of conversions {nonnegative}	<code>EOF {negative}</code>
<code>vsscanf_s()</code>	Number of conversions {nonnegative}	<code>EOF {negative}</code>

vswprintf()	Number of non-null wide characters	Negative
vswprintf_s()	Number of non-null wide characters	Negative
vswscanf()	Number of conversions (nonnegative)	EOF (negative)
vswscanf_s()	Number of conversions (nonnegative)	EOF (negative)
vwprintf_s()	Number of wide characters (nonnegative)	Negative
vwscanf()	Number of conversions (nonnegative)	EOF (negative)
vwscanf_s()	Number of conversions (nonnegative)	EOF (negative)
wcrtomb()	Number of bytes stored	(size_t) (-1)
wcschr()	Pointer to located wide character	NULL
wcsftime()	Number of non-null wide characters	0
wcspbrk()	Pointer to located wide character	NULL
wcsrchr()	Pointer to located wide character	NULL
wcsrtombs()	Number of non-null bytes	(size_t) (-1), errno == EILSEQ
wcsrtombs_s()	0	Nonzero
wcsstr()	Pointer to located wide string	NULL
wcstod()	Converted value	0, errno == ERANGE
wcstof()	Converted value	0, errno == ERANGE
wcstoiimax()	Converted value	INTMAX_MAX OR INTMAX_MIN, errno == ERANGE
wcstok()	Pointer to first wide character of a token	NULL
wcstok_s()	Pointer to first wide character of a token	NULL
wcstol()	Converted value	LONG_MAX OR LONG_MIN, errno == ERANGE
wcstold()	Converted value	0, errno == ERANGE
wcstoll()	Converted value	LLONG_MAX OR LLONG_MIN, errno == ERANGE
wcstombs()	Number of non-null bytes	(size_t) (-1)
wcstombs_s()	0	Nonzero
wcstoumax()	Converted value	UINTMAX_MAX, errno == ERANGE
wcstoul()	Converted value	ULONG_MAX, errno == ERANGE
wcstoull()	Converted value	ULLONG_MAX, errno == ERANGE
wcsxfrm()	Length of transformed wide string	>= n
wctob()	Converted character	EOF
wctomb(), s != NULL	Number of bytes stored	-1
wctomb_s(), s != NULL	Number of bytes stored	-1
wctrans()	Valid argument to towctrans	0
wctype()	Valid argument to iswctype	0
wmemchr()	Pointer to located wide character	NULL
wprintf_s()	Number of wide characters (nonnegative)	Negative
wscanf()	Number of conversions (nonnegative)	EOF (negative)
wscanf_s()	Number of conversions (nonnegative)	EOF (negative)

Note: According to F1035-C, Use feof() and ferror() to detect end-of-file and file errors when sizeof(int) == sizeof(char), callers should verify end-of-file and file

errors for the functions in this table as follows:

¹ By calling `ferror()` and `feof()`

² By calling `ferror()`

The `ungetc()` function does not set the error indicator even when it fails, so it is not possible to check for errors reliably unless it is known that the argument is not equal to `EOF`. The C Standard [ISO/IEC 9899:2011] states that "one character of pushback is guaranteed," so this should not be an issue if, at most, one character is ever pushed back before reading again. (See [F1013-C. Never push back anything other than one read character](#).)

Noncompliant Code Example (`setlocale()`)

In this noncompliant code example, the function `utf8_to_wcs()` attempts to convert a sequence of UTF-8 characters to wide characters. It first invokes `setlocale()` to set the global locale to the implementation-defined `en_US.UTF-8` but does not check for failure. The `setlocale()` function will fail by returning a null pointer, for example, when the locale is not installed. The function may fail for other reasons as well, such as the lack of resources. Depending on the sequence of characters pointed to by `utf8`, the subsequent call to `mbstowcs()` may fail or result in the function storing an unexpected sequence of wide characters in the supplied buffer `wcs`.

```
#include <locale.h>
#include <stdlib.h>

int utf8_to_wcs(wchar_t *wcs, size_t n, const char *utf8,
                size_t *size) {
    if (NULL == size) {
        return -1;
    }
    setlocale(LC_CTYPE, "en_US.UTF-8");
    *size = mbstowcs(wcs, utf8, n);
    return 0;
}
```

Compliant Solution (`setlocale()`)

This compliant solution checks the value returned by `setlocale()` and avoids calling `mbstowcs()` if the function fails. The function also takes care to restore the locale to its initial setting before returning control to the caller.

```
#include <locale.h>
#include <stdlib.h>

int utf8_to_wcs(wchar_t *wcs, size_t n, const char *utf8,
                size_t *size) {
    if (NULL == size) {
        return -1;
    }
    const char *save = setlocale(LC_CTYPE, "en_US.UTF-8");
    if (NULL == save) {
        return -1;
    }

    *size = mbstowcs(wcs, utf8, n);
    if (NULL == setlocale(LC_CTYPE, save)) {
        return -1;
    }
    return 0;
}
```

Noncompliant Code Example (`calloc()`)

In this noncompliant code example, `temp_num`, `tmp2`, and `num_of_records` are derived from a [tainted source](#). Consequently, an attacker can easily cause `calloc()` to fail by providing a large value for `num_of_records`.

```
#include <stdlib.h>
#include <string.h>

enum { SIG_DESC_SIZE = 32 };

typedef struct {
    char sig_desc[SIG_DESC_SIZE];
} signal_info;

void func(size_t num_of_records, size_t temp_num,
          const char *tmp2, size_t tmp2_size_bytes) {
    signal_info *start = (signal_info *)calloc(num_of_records,
                                                sizeof(signal_info));

    if (tmp2 == NULL) {
        /* Handle error */
    } else if (temp_num > num_of_records) {
        /* Handle error */
    } else if (tmp2_size_bytes < SIG_DESC_SIZE) {
        /* Handle error */
    }

    signal_info *point = start + temp_num - 1;
    memcpy(point->sig_desc, tmp2, SIG_DESC_SIZE);
    point->sig_desc[SIG_DESC_SIZE - 1] = '\0';
    /* ... */
    free(start);
}
```

When `calloc()` fails, it returns a null pointer that is assigned to `start`. If `start` is null, an attacker can provide a value for `temp_num` that, when scaled by

`sizeof(signal_info)`, references a writable address to which control is eventually transferred. The contents of the string referenced by `tmp2` can then be used to overwrite the address, resulting in an arbitrary code execution [vulnerability](#).

Compliant Solution (`calloc()`)

To correct this error, ensure the pointer returned by `calloc()` is not null:

```
#include <stdlib.h>
#include <string.h>

enum { SIG_DESC_SIZE = 32 };

typedef struct {
    char sig_desc[SIG_DESC_SIZE];
} signal_info;

void func(size_t num_of_records, size_t temp_num,
          const char *tmp2, size_t tmp2_size_bytes) {
    signal_info *start = (signal_info *)calloc(num_of_records,
                                                sizeof(signal_info));

    if (start == NULL) {
        /* Handle allocation error */
    } else if (tmp2 == NULL) {
        /* Handle error */
    } else if (temp_num > num_of_records) {
        /* Handle error */
    } else if (tmp2_size_bytes < SIG_DESC_SIZE) {
        /* Handle error */
    }

    signal_info *point = start + temp_num - 1;
    memcpy(point->sig_desc, tmp2, SIG_DESC_SIZE);
    point->sig_desc[SIG_DESC_SIZE - 1] = '\0';
    /* ... */
    free(start);
}
```

Noncompliant Code Example (`realloc()`)

This noncompliant code example calls `realloc()` to resize the memory referred to by `p`. However, if `realloc()` fails, it returns a null pointer and the connection between the original block of memory and `p` is lost, resulting in a memory leak.

```
#include <stdlib.h>

void *p;
void func(size_t new_size) {
    if (new_size == 0) {
        /* Handle error */
    }
    p = realloc(p, new_size);
    if (p == NULL) {
        /* Handle error */
    }
}
```

This code example complies with [MEM04-C. Do not perform zero-length allocations](#).

Compliant Solution (`realloc()`)

In this compliant solution, the result of `realloc()` is assigned to the temporary pointer `q` and validated before it is assigned to the original pointer `p`:

```
#include <stdlib.h>

void *p;
void func(size_t new_size) {
    void *q;

    if (new_size == 0) {
        /* Handle error */
    }

    q = realloc(p, new_size);
    if (q == NULL) {
        /* Handle error */
    } else {
        p = q;
    }
}
```

Noncompliant Code Example (`fseek()`)

In this noncompliant code example, the `fseek()` function is used to set the file position to a location `offset` in the file referred to by `file` prior to reading a sequence of bytes from the file. However, if an I/O error occurs during the seek operation, the subsequent read will fill the buffer with the wrong contents.

```
#include <stdio.h>

size_t read_at(FILE *file, long offset,
               void *buf, size_t nbytes) {
    fseek(file, offset, SEEK_SET);
    return fread(buf, 1, nbytes, file);
}
```

Compliant Solution (`fseek()`)

According to the C Standard, the `fseek()` function returns a nonzero value to indicate that an error occurred. This compliant solution tests for this condition before reading from a file to eliminate the chance of operating on the wrong portion of the file if `fseek()` fails:

```
#include <stdio.h>

size_t read_at(FILE *file, long offset,
                void *buf, size_t nbytes) {
    if (fseek(file, offset, SEEK_SET) != 0) {
        /* Indicate error to caller */
        return 0;
    }
    return fread(buf, 1, nbytes, file);
}
```

Noncompliant Code Example (`snprintf()`)

In this noncompliant code example, `snprintf()` is assumed to succeed. However, if the call fails (for example, because of insufficient memory, as described in GNU libc bug [441945](#)), the subsequent call to `log_message()` has [undefined behavior](#) because the character buffer is uninitialized and need not be null-terminated.

```
#include <stdio.h>

extern void log_message(const char *);

void f(int i, int width, int prec) {
    char buf[40];
    snprintf(buf, sizeof(buf), "i = %.*i", width, prec, i);
    log_message(buf);
    /* ... */
}
```

Compliant Solution (`snprintf()`)

This compliant solution does not assume that `snprintf()` will succeed regardless of its arguments. It tests the return value of `snprintf()` before subsequently using the formatted buffer. This compliant solution also treats the case where the static buffer is not large enough for `snprintf()` to append the terminating null character as an error.

```
#include <stdio.h>
#include <string.h>

extern void log_message(const char *);

void f(int i, int width, int prec) {
    char buf[40];
    int n;
    n = snprintf(buf, sizeof(buf), "i = %.*i", width, prec, i);
    if (n < 0 || n >= sizeof(buf)) {
        /* Handle snprintf() error */
        strcpy(buf, "unknown error");
    }
    log_message(buf);
}
```

Compliant Solution (`snprintf(null)`)

If unknown, the length of the formatted string can be discovered by invoking `snprintf()` with a null buffer pointer to determine the size required for the output, then dynamically allocating a buffer of sufficient size, and finally calling `snprintf()` again to format the output into the dynamically allocated buffer. Even with this approach, the success of all calls still needs to be tested, and any errors must be appropriately handled. A possible optimization is to first attempt to format the string into a reasonably small buffer allocated on the stack and, only when the buffer turns out to be too small, dynamically allocate one of a sufficient size:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern void log_message(const char *);

void f(int i, int width, int prec) {
    char buffer[20];
    char *buf = buffer;
    int n = sizeof(buffer);
    const char fmt[] = "i = %.*i";

    n = snprintf(buf, n, fmt, width, prec, i);
    if (n < 0) {
        /* Handle snprintf() error */
        strcpy(buffer, "unknown error");
        goto write_log;
    }

    if (n < sizeof(buffer)) {
        goto write_log;
    }

    buf = (char *)malloc(n + 1);
    if (NULL == buf) {

```

```

/* Handle malloc() error */
strcpy(buffer, "unknown error");
goto write_log;
}

n = sprintf(buf, n, fmt, width, prec, i);
if (n < 0) {
    /* Handle sprintf() error */
    strcpy(buffer, "unknown error");
}

write_log:
log_message(buf);

if (buf != buffer) {
    free(buf);
}
}

```

This solution uses the `goto` statement, as suggested in [MEM12-C. Consider using a goto chain when leaving a function on error when using and releasing resources](#).

Exceptions

ERR33-C-EX1: It is acceptable to ignore the return value of a function that cannot fail, or a function whose return value is inconsequential, or when an error condition need not be diagnosed. The function's results should be explicitly cast to `void` to signify programmer intent. Return values from the functions in the following table do not need to be checked because their historical use has overwhelmingly omitted error checking and the consequences are not relevant to security.

Functions for which Return Values Need Not Be Checked

Function	Successful Return	Error Return
<code>putchar()</code>	Character written	<code>EOF</code>
<code>putwchar()</code>	Wide character written	<code>WEOF</code>
<code>puts()</code>	Nonnegative	<code>EOF</code> (negative)
<code>printf()</code> , <code>vprintf()</code>	Number of characters (nonnegative)	Negative
<code>wprintf()</code> , <code>vwprintf()</code>	Number of wide characters (nonnegative)	Negative
<code>kill_dependency()</code>	The input parameter	NA
<code>memcpy()</code> , <code>wmemcpy()</code>	The destination input parameter	NA
<code>memmove()</code> , <code>wmemmove()</code>	The destination input parameter	NA
<code>strncpy()</code> , <code>wcsncpy()</code>	The destination input parameter	NA
<code>strcat()</code> , <code>wcsconcat()</code>	The destination input parameter	NA
<code>strncat()</code> , <code>wcsncat()</code>	The destination input parameter	NA
<code>memset()</code> , <code>wmemset()</code>	The destination input parameter	NA

Risk Assessment

Failing to detect error conditions can lead to unpredictable results, including [abnormal program termination](#) and [denial-of-service attacks](#) or, in some situations, could even allow an attacker to run arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR33-C	High	Likely	Medium	P18	L1

Related Guidelines

CERT C Secure Coding Standard	ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy EXP34-C. Do not dereference null pointers FI013-C. Never push back anything other than one read character MEM04-C. Do not perform zero-length allocations MEM12-C. Consider using a goto chain when leaving a function on error when using and releasing resources
SEI CERT C++ Coding Standard	ERR10-CPP. Check for error conditions FI004-CPP. Detect and handle input and output errors
ISO/IEC TS 17961:2013	Failing to detect and handle standard library errors [liberr]
MITRE CWE	CWE-252 , Unchecked Return Value CWE-253 , Incorrect Check of Function Return Value CWE-390 , Detection of Error Condition without Action CWE-391 , Unchecked Error Condition CWE-476 , NULL Pointer Dereference

Bibliography

[DHS 2006]	Handle All Errors Safely
[Henricson 1997]	Recommendation 12.1, "Check for All Errors Reported from Functions"
[ISO/IEC 9899:2011]	Subclause 7.21.7.10, "The <code>ungetc</code> Function"
[VU#159523]	

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/w4C4Ag>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
allowed_functions		frozenset(['strncat', 'memmove', 'memset', 'strcat', 'strcpy', 'memcpy', 'strncpy'])
errno_headers	Header names were 'errno', 'EILSEQ' and 'ERANGE' might be located.	['errno.h', 'errno-base.h']
function_argument_lookup	Function name to argument position mapping where argument and return value are forbidden to share the same variable.	dict{...}
functions	Allows to declare function names for which a check must exist. The check is expressed as an IR pattern.	dict{...}
inspect_template_instances	Whether calls in template instances should be reported.	False
known_check_functions	Collection of functions which are known to test return values of functions under test.	[]
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
stdio_headers	Header names were 'EOF' might be located.	['stdio.h', 'libio.h']
threads_headers	Header names were thread related macros and functions might be located.	['threads.h']
wchar_headers	Header names were 'WEOF' might be located.	['wchar.h', 'corecrt_wctype.h', '_wctype.h', '_types.h']

Possible Messages

Name	Message
discarded_return_with_entity	Return value of function discarded.
discarded_return_without_entity	Return value of function discarded.
possible_leak	Possible memory leak in function error case when overriding a pointer.
unhandled_return_value	Return value of function call not properly checked.

CertC++-ERR34

Detect errors when converting a string to a number.

Input: IR

Source languages: C++

Details

The process of parsing an integer or floating-point number from a string can produce many errors. The string might not contain a number. It might contain a number of the correct type that is out of range (such as an integer that is larger than `INT_MAX`). The string may also contain extra information after the number, which may or may not be useful after the conversion. These error conditions must be detected and addressed when a string-to-number conversion is performed using a C Standard Library function.

The `strtol()`, `strtoll()`, `strtoimax()`, `strtoul()`, `strtoull()`, `strtoimax()`, `strtof()`, `strtod()`, and `strtold()` functions convert the initial portion of a null-terminated byte string to a `long int`, `long long int`, `intmax_t`, `unsigned long int`, `unsigned long long int`, `uintmax_t`, `float`, `double`, and `long double` representation, respectively.

Use one of the C Standard Library `strto*` functions to parse an integer or floating-point number from a string. These functions provide more robust error handling than alternative solutions. Also, use the `strtol()` function to convert to a smaller signed integer type such as `signed int`, `signed short`, and `signed char`, testing the result against the range limits for that type. Likewise, use the `strtoul()` function to convert to a smaller unsigned integer type such as `unsigned int`, `unsigned short`, and `unsigned char`, and test the result against the range limits for that type. These range tests do nothing if the smaller type happens to have the same size and representation for a particular implementation.

Noncompliant Code Example (`atoi()`)

This noncompliant code example converts the string token stored in the `buff` to a signed integer value using the `atoi()` function:

```
#include <stdlib.h>

void func(const char *buff) {
    int si;

    if (buff) {
        si = atoi(buff);
    } else {
        /* Handle error */
    }
}
```

The `atoi()`, `atol()`, `atoll()`, and `atof()` functions convert the initial portion of a string token to `int`, `long int`, `long long int`, and `double` representation, respectively. Except for the behavior on error, they are equivalent to

<code>atoi: (int)strtol(nptr, (char **)NULL, 10)</code>
<code>atol: strtol(nptr, (char **)NULL, 10)</code>
<code>atoll: strtoll(nptr, (char **)NULL, 10)</code>
<code>atof: strtod(nptr, (char **)NULL)</code>

Unfortunately, `atoi()` and related functions lack a mechanism for reporting errors for invalid values. Specifically, these functions:

- do not need to set `errno` on an error;
- have undefined behavior if the value of the result cannot be represented;
- return 0 (or 0.0) if the string does not represent an integer (or decimal), which is indistinguishable from a correctly formatted, zero-denoting input string.

Noncompliant Example (`sscanf()`)

This noncompliant example uses the `sscanf()` function to convert a string token to an integer. The `sscanf()` function has the same limitations as `atoi()`:

```
#include <stdio.h>

void func(const char *buff) {
    int matches;
    int si;

    if (buff) {
        matches = sscanf(buff, "%d", &si);
        if (matches != 1) {
            /* Handle error */
        }
    } else {
        /* Handle error */
    }
}
```

The `sscanf()` function returns the number of input items successfully matched and assigned, which can be fewer than provided for, or even 0 in the event of an early matching failure. However, `sscanf()` fails to report the other errors reported by `strtol()`, such as numeric overflow.

Compliant Solution (`strtol()`)

The `strtol()`, `strtoll()`, `strtoimax()`, `strtoul()`, `strtoull()`, `strtoumax()`, `strtof()`, `strtod()`, and `strtold()` functions convert a null-terminated byte string to `long`, `int`, `long long`, `intmax_t`, `unsigned long`, `unsigned long long`, `uintmax_t`, `float`, `double`, and `long double` representation, respectively.

This compliant solution uses `strtol()` to convert a string token to an integer and ensures that the value is in the range of `int`:

```
#include <errno.h>
#include <limits.h>
#include <stdlib.h>
#include <stdio.h>

void func(const char *buff) {
    char *end;
    int si;

    errno = 0;

    const long sl = strtol(buff, &end, 10);

    if (end == buff) {
        fprintf(stderr, "%s: not a decimal number\n", buff);
    } else if ('\0' != *end) {
        fprintf(stderr, "%s: extra characters at end of input: %s\n", buff, end);
    } else if ((LONG_MIN == sl || LONG_MAX == sl) && ERANGE == errno) {
        fprintf(stderr, "%s out of range of type long\n", buff);
    } else if (sl > INT_MAX) {
        fprintf(stderr, "%ld greater than INT_MAX\n", sl);
    } else if (sl < INT_MIN) {
        fprintf(stderr, "%ld less than INT_MIN\n", sl);
    } else {
        si = (int)sl;

        /* Process si */
    }
}
```

Risk Assessment

It is rare for a violation of this rule to result in a security [vulnerability](#) unless it occurs in security-sensitive code. However, violations of this rule can easily result in lost or misinterpreted data.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
ERR34-C	Medium	Unlikely	Medium	P4	L3

Related Guidelines

SEI CERT C++ Coding Standard	INT06-CPP. Use strtol() or a related function to convert a string token to an integer
MITRE CWE	CWE-676 , Use of potentially dangerous function CWE-20 , Insufficient input validation

Bibliography

[ISO/IEC 9899:2011]	Subclause 7.22.1, "Numeric conversion functions" Subclause 7.21.6, "Formatted input/output functions"
[Klein 2002]	

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/6AQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
scanf_functions		dict(...)
symbol_header	Name of the header file of which the symbols should not be used.	['stdlib', 'stdio']
symbols	Names of symbols which are forbidden.	['atof', 'atoi', 'atol', 'atoll']
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.
scanf_conversion_to_number	Potential numeric overflow: do not use functions of scanf() family to convert a string to number.

CertC++-ERR51

Handle all exceptions.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

When an exception is thrown, control is transferred to the nearest handler with a type that matches the type of the exception thrown. If no matching handler is directly found within the handlers for a try block in which the exception is thrown, the search for a matching handler continues to dynamically search for handlers in the surrounding try blocks of the same thread. The C++ Standard, [except.handle], paragraph 9 [[ISO/IEC 14882-2014](#)], states the following:

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.

The default terminate handler called by `std::terminate()` calls `std::abort()`, which [abnormally terminates](#) the process. When `std::abort()` is called, or if the [implementation](#) does not unwind the stack prior to calling `std::terminate()`, destructors for objects may not be called and external resources can be left in an indeterminate state. Abnormal process termination is the typical vector for [denial-of-service](#) attacks. For more information on implicitly calling `std::terminate()`, see [ERR50-CPP. Do not abruptly terminate the program](#).

All exceptions thrown by an application must be caught by a matching exception handler. Even if the exception cannot be gracefully recovered from, using the matching exception handler ensures that the stack will be properly unwound and provides an opportunity to gracefully manage external resources before terminating the process.

As per [ERR50-CPP-EX1](#), a program that encounters an unrecoverable exception may explicitly catch the exception and terminate, but it may not allow the exception to remain uncaught. One possible solution to comply with this rule, as well as with ERR50-CPP, is for the `main()` function to catch all exceptions. While this does not generally allow the application to recover from the exception gracefully, it does allow the application to terminate in a controlled fashion.

Noncompliant Code Example

In this noncompliant code example, neither `f()` nor `main()` catch exceptions thrown by `throwing_func()`. Because no matching handler can be found for the exception thrown, `std::terminate()` is called.

```
void throwing_func() noexcept(false);
void f() {
    throwing_func();
}
int main() {
    f();
}
```

Compliant Solution

In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the `main()` function and allows for

graceful management of external resources.

```
void throwing_func() noexcept(false);

void f() {
    throwing_func();
}

int main() {
    try {
        f();
    } catch (...) {
        // Handle error
    }
}
```

Noncompliant Code Example

In this noncompliant code example, the thread entry point function `thread_start()` does not catch exceptions thrown by `throwing_func()`. If the initial thread function exits because an exception is thrown, `std::terminate()` is called.

```
#include <thread>

void throwing_func() noexcept(false);

void thread_start() {
    throwing_func();
}

void f() {
    std::thread t(thread_start);
    t.join();
}
```

Compliant Solution

In this compliant solution, the `thread_start()` handles all exceptions and does not rethrow, allowing the thread to terminate normally.

```
#include <thread>

void throwing_func() noexcept(false);

void thread_start(void) {
    try {
        throwing_func();
    } catch (...) {
        // Handle error
    }
}

void f() {
    std::thread t(thread_start);
    t.join();
}
```

Risk Assessment

Allowing the application to [abnormally terminate](#) can lead to resources not being freed, closed, and so on. It is frequently a vector for [denial-of-service attacks](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR51-CPP	Low	Probable	Medium	P4	L3

Related Guidelines

This rule is a subset of [ERR50-CPP. Do not abruptly terminate the program.](#)

MITRE CWE	CWE-754 , Improper Check for Unusual or Exceptional Conditions
---------------------------	--

Bibliography

[ISO/IEC 14882-2014]	Subclause 15.1, "Throwing an Exception" Subclause 15.3, "Handling an Exception" Subclause 15.5.1, "The <code>std::terminate()</code> Function"
[MISRA 2008]	Rule 15-3-2 (Advisory) Rule 15-3-4 (Required)

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
inspect_at_exit_handlers	Whether to also inspect at_exit() handlers-functions.	True
inspect_thread_main	Whether to also inspect thread main functions.	True
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
exception_escaping_main	Uncaught exception escaping from main or additional entry point

CertC++-ERR52

Do not use `setjmp()` or `longjmp()`.

Input: IR

Source languages: C++

Details

The C standard library facilities `setjmp()` and `longjmp()` can be used to simulate throwing and catching exceptions. However, these facilities bypass automatic resource management and can result in [undefined behavior](#), commonly including resource leaks and [denial-of-service attacks](#).

The C++ Standard, [support.runtime], paragraph 4 [[ISO/IEC 14882-2014](#)], states the following:

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.

Do not call `setjmp()` or `longjmp()`; their usage can be replaced by more standard idioms such as `throw` expressions and `catch` statements.

Noncompliant Code Example

If a `throw` expression would cause a nontrivial destructor to be invoked, then calling `longjmp()` in the same context will result in [undefined behavior](#). In the following noncompliant code example, the call to `longjmp()` occurs in a context with a local `Counter` object. Since this object's destructor is nontrivial, undefined behavior results.

```
#include <csetjmp>
#include <iostream>

static jmp_buf env;

struct Counter {
    static int instances;
    Counter() { ++instances; }
    ~Counter() { --instances; }
};

int Counter::instances = 0;

void f() {
    Counter c;
    std::cout << "f(): Instances: " << Counter::instances << std::endl;
    std::longjmp(env, 1);
}

int main() {
    std::cout << "Before setjmp(): Instances: " << Counter::instances << std::endl;
    if (setjmp(env) == 0) {
        f();
    } else {
        std::cout << "From longjmp(): Instances: " << Counter::instances << std::endl;
    }
    std::cout << "After longjmp(): Instances: " << Counter::instances << std::endl;
}
```

Implementation Details

The above code produces the following results when compiled with [Clang 3.8](#) for Linux, demonstrating that the program, on this platform, fails to destroy the local `Counter` instance when the execution of `f()` is terminated. This is permissible as the behavior is undefined.

```
Before setjmp(): Instances: 0
f(): Instances: 1
From longjmp(): Instances: 1
```

After longjmp(): Instances: 1

Compliant Solution

This compliant solution replaces the calls to `setjmp()` and `longjmp()` with a `throw` expression and a `catch` statement.

```
#include <iostream>

struct Counter {
    static int instances;
    Counter() { ++instances; }
    ~Counter() { --instances; }
};

int Counter::instances = 0;

void f() {
    Counter c;
    std::cout << "f(): Instances: " << Counter::instances << std::endl;
    throw "Exception";
}

int main() {
    std::cout << "Before throw: Instances: " << Counter::instances << std::endl;
    try {
        f();
    } catch (const char *E) {
        std::cout << "From catch: Instances: " << Counter::instances << std::endl;
    }
    std::cout << "After catch: Instances: " << Counter::instances << std::endl;
}
```

This solution produces the following output.

```
Before throw: Instances: 0
f(): Instances: 1
From catch: Instances: 0
After catch: Instances: 0
```

Risk Assessment

Using `setjmp()` and `longjmp()` could lead to a [denial-of-service attack](#) due to resources not being properly destroyed.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR52-CPP	Low	Probable	Medium	P4	L3

Bibliography

[Henricson 1997]	Rule 13.3, Do not use <code>setjmp()</code> and <code>longjmp()</code>
[ISO/IEC 14882-2014]	Subclause 18.10, "Other Runtime Support"

Excerpt from SEI CERT C++ Coding Standard Wiki [\[https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046492\]](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046492), Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
jump_functions	Forbidden jump functions.	['setjmp', 'longjmp']
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
jump	Do not use <code>setjmp()</code> or <code>longjmp()</code> .

CertC++-ERR53

Do not reference base classes or class data members in a constructor or destructor function-block handler.

Input: IR

Source languages: C++

Details

When an exception is caught by a *function-try-block* handler in a constructor, any fully constructed base classes and class members of the object are destroyed prior to entering the handler [ISO/IEC 14882-2014]. Similarly, when an exception is caught by a *function-try-block* handler in a destructor, all base classes and nonvariant class members of the objects are destroyed prior to entering the handler. Because of this behavior, the C++ Standard, [except.handle], paragraph 10, states the following:

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.

Do not reference base classes or class data members in a constructor or destructor *function-try-block* handler. Doing so results in [undefined behavior](#).

Noncompliant Code Example

In this noncompliant code example, the constructor for class C handles exceptions with a *function-try-block*. However, it generates [undefined behavior](#) by inspecting its member field str.

```
#include <string>

class C {
    std::string str;

public:
    C(const std::string &s) try : str(s) {
        // ...
    } catch (...) {
        if (!str.empty())
            // ...
    }
};
```

Compliant Solution

In this compliant solution, the handler inspects the constructor parameter rather than the class data member, thereby avoiding [undefined behavior](#).

```
#include <string>

class C {
    std::string str;

public:
    C(const std::string &s) try : str(s) {
        // ...
    } catch (...) {
        if (!s.empty())
            // ...
    }
};
```

Risk Assessment

Accessing nonstatic data in a constructor's exception handler or a destructor's exception handler leads to [undefined behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR53-CPP	Low	Unlikely	Medium	P2	L3

Related Guidelines

[[MISRA 2008](#)] Rule 15-3-3 [Required]

Bibliography

[[ISO/IEC 14882-2014](#)] Subclause 15.3, "Handling an Exception"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR53-CPP.+Do+not+reference+base+classes+or+class+data+members+in+a+constructor+or+destructor+function-try-block+handler>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
handler_uses_field	Handler of a function-try-block shall not reference non-static members from this class or its bases

CertC++-ERR54

Catch handlers should order their parameter types from most derived to least derived.

Input: IR

Source languages: C++

Details

The C++ Standard, [except.handle], paragraph 4 [[ISO/IEC 14882-2014](#)], states the following:

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.

Consequently, if two handlers catch exceptions that are derived from the same base class (such as `std::exception`), the most derived exception must come first.

Noncompliant Code Example

In this noncompliant code example, the first handler catches all exceptions of class `B`, as well as exceptions of class `D`, since they are also of class `B`. Consequently, the second handler does not catch any exceptions.

```
// Classes used for exception handling
class B {};
class D : public B {};

void f() {
    try {
        // ...
    } catch (B &b) {
        // ...
    } catch (D &d) {
        // ...
    }
}
```

Compliant Solution

In this compliant solution, the first handler catches all exceptions of class `D`, and the second handler catches all the other exceptions of class `B`.

```
// Classes used for exception handling
class B {};
class D : public B {};

void f() {
    try {
        // ...
    } catch (D &d) {
        // ...
    } catch (B &b) {
        // ...
    }
}
```

Risk Assessment

Exception handlers with inverted priorities cause unexpected control flow when an exception of the derived type occurs.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR54-CPP	Medium	Likely	Low	P18	L1

Related Guidelines

Bibliography

[ISO/IEC 14882-2014]	Subclause 15.3, "Handling an Exception"
----------------------	---

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR54-CPP.+Catch+handlers+should+order+their+parameter+types+from+most+derived+to+least+derived>], Copyright [C] 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
catch_all_not_last	Catch-all shall occur as last handler.
wrong_catch_order	Catch handlers in wrong order.

CertC++-ERR55

Honor exception specifications.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

The C++ Standard, [except.spec], paragraph 8 [ISO/IEC 14882-2014], states the following:

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 (15.3) for an exception of type E .

If a function throws an exception other than one allowed by its *exception-specification*, it can lead to an *implementation-defined* termination of the program ([except.spec], paragraph 9).

If a function declared with a *dynamic-exception-specification* throws an exception of a type that would not match the *exception-specification*, the function `std::unexpected()` is called. The behavior of this function can be overridden but, by default, causes an exception of `std::bad_exception` to be thrown. Unless `std::bad_exception` is listed in the *exception-specification*, the function `std::terminate()` will be called.

Similarly, if a function declared with a *noexcept-specification* throws an exception of a type that would cause the *noexcept-specification* to evaluate to `false`, the function `std::terminate()` will be called.

Calling `std::terminate()` leads to implementation-defined termination of the program. To prevent *abnormal termination* of the program, any function that declares an *exception-specification* should restrict itself, as well as any functions it calls, to throwing only allowed exceptions.

Noncompliant Code Example

In this noncompliant code example, the second function claims to throw only `Exception1`, but it may also throw `Exception2`.

```
#include <exception>

class Exception1 : public std::exception {};
class Exception2 : public std::exception {};

void foo() {
    throw Exception2(); // Okay because foo() promises nothing about exceptions
}

void bar() throw (Exception1) {
    foo(); // Bad because foo() can throw Exception2
}
```

Compliant Solution

This compliant solution catches the exceptions thrown by `foo()`.

```
#include <exception>

class Exception1 : public std::exception {};
class Exception2 : public std::exception {};

void foo() {
    throw Exception2{}; // Okay because foo() promises nothing about exceptions
}

void bar() throw (Exception1) {
    try {
        foo();
    } catch (Exception2 e) {
        // Handle error without rethrowing it
    }
}
```

Compliant Solution

This compliant solution declares a dynamic *exception-specification* for `bar()`, which covers all of the exceptions that can be thrown from it.

```
#include <exception>

class Exception1 : public std::exception {};
class Exception2 : public std::exception {};

void foo() {
    throw Exception2{}; // Okay because foo() promises nothing about exceptions
}

void bar() throw (Exception1, Exception2) {
    foo();
}
```

Noncompliant Code Example

In this noncompliant code example, a function is declared as nonthrowing, but it is possible for `std::vector::resize()` to throw an exception when the requested memory cannot be allocated.

```
#include <cstddef>
#include <vector>

void f(std::vector<int> &v, size_t s) noexcept(true) {
    v.resize(s); // May throw
}
```

Compliant Solution

In this compliant solution, the function's *noexcept-specification* is removed, signifying that the function allows all exceptions.

```
#include <cstddef>
#include <vector>

void f(std::vector<int> &v, size_t s) {
    v.resize(s); // May throw, but that is okay
}
```

Implementation Details

Some vendors provide language extensions for specifying whether or not a function throws. For instance, [Microsoft Visual Studio](#) provides `__declspec(nothrow)`, and [Clang](#) supports `__attribute__((nothrow))`. Currently, the vendors do not document the behavior of specifying a nonthrowing function using these extensions. Throwing from a function declared with one of these language extensions is presumed to be [undefined behavior](#).

Risk Assessment

Throwing unexpected exceptions disrupts control flow and can cause premature termination and [denial of service](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR55-CPP	Low	Likely	Low	P9	L2

Related Guidelines

[SEI CERT C++ Coding Standard](#) | [ERR50-CPP. Do not abruptly terminate the program](#)

Bibliography

[GNU 2016]	"Declaring Attributes of Functions"
[ISO/IEC 14882-2014]	Subclause 15.4, "Exception Specifications"
[MSDN 2016]	" <code>nothrow</code> [C++]"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR55-CPP.+Honor+exception+specifications>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
<code>generate_violation_path</code>	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
<code>ignore_constructor_destructor</code>	Whether to ignore escaping exceptions from constructors and destructors.	False
<code>ignore_unkown_routines</code>	Whether to ignore extern or only declared routines.	False
<code>level</code>	Grouping of priorities into different levels	2
<code>likelihood</code>	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
<code>priority</code>	Priority based on the combination of severity, likelihood and remediation cost	9
<code>recommendation</code>	Whether this check is classified as a recommendation or rule	False
<code>remediation_cost</code>	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
<code>exception_specificationViolation</code>	Exception violates function's exception-specification.

CertC++-ERR58

Handle all exceptions thrown before `main()` begins executing.

Input: IR

Source languages: C++

Note: Rule requires `CONFIG.Run_IRAnalysis_Checks = True`.

Details

Not all exceptions can be caught, even with careful use of *function-try-blocks*. The C++ Standard, [except.handle], paragraph 13 [[ISO/IEC 14882-2014](#)], states the following:

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.

When declaring an object with static or thread storage duration, and that object is not declared within a function block scope, the type's constructor must be declared `noexcept` and must comply with [ERR55-CPP. Honor exception specifications](#). Additionally, the initializer for such a declaration, if any, must not throw an uncaught exception (including from any implicitly constructed objects that are created as a part of the initialization). If an uncaught exception is thrown before `main()` is executed, or if an uncaught exception is thrown after `main()` has finished executing, there are no further opportunities to handle the exception and it results in implementation-defined behavior. (See [ERR50-CPP. Do not abruptly terminate the program](#) for further details.)

For more information on exception specifications of destructors, see [DCL57-CPP. Do not let exceptions escape from destructors or deallocation functions](#).

Noncompliant Code Example

In this noncompliant example, the constructor for `s` may throw an exception that is not caught when `globals` is constructed during program startup.

```
struct S {
    S() noexcept(false);
};

static S globals;
```

Compliant Solution

This compliant solution makes `globals` into a local variable with static storage duration, allowing any exceptions thrown during object construction to be caught because the constructor for `s` will be executed the first time the function `globals()` is called rather than at program startup. This solution does require the programmer to modify source code so that previous uses of `globals` are replaced by a function call to `globals()`.

```

struct S {
    S() noexcept(false);
};

S &globals() {
    try {
        static S s;
        return s;
    } catch (...) {
        // Handle error, perhaps by logging it and gracefully terminating the application.
    }
    // Unreachable.
}

```

Noncompliant Code Example

In this noncompliant example, the constructor of `global` may throw an exception during program startup. (The `std::string` constructor, which accepts a `const char *` and a default allocator object, is not marked `noexcept` and consequently allows all exceptions.) This exception is not caught by the *function-try-block* on `main()`, resulting in a call to `std::terminate()` and [abnormal program termination](#).

```

#include <string>

static const std::string global("...");

int main()
try {
    // ...
} catch(...) {
    // IMPORTANT: Will not catch exceptions thrown
    // from the constructor of global
}

```

Compliant Solution

Compliant code must prevent exceptions from escaping during program startup and termination. This compliant solution avoids defining a `std::string` at global namespace scope and instead uses a `static const char *`.

```

static const char *global = "...";

int main() {
    // ...
}

```

Compliant Solution

This compliant solution introduces a class derived from `std::string` with a constructor that catches all exceptions with a function try block and terminates the application in accordance with [ERR50-CPP-EX1](#) in [ERR50-CPP. Do not abruptly terminate the program](#) in the event any exceptions are thrown. Because no exceptions can escape the constructor, it is marked `noexcept` and the class type is permissible to use in the declaration or initialization of a static global variable.

For brevity, the full interface for such a type is not described.

```

#include <exception>
#include <string>

namespace my {
    struct string : std::string {
        explicit string(const char *msg,
                        const std::string::allocator_type &alloc = std::string::allocator_type{}) noexcept
            try : std::string(msg, alloc) {} catch(...) {
                extern void log_message(const char *) noexcept;
                log_message("std::string constructor threw an exception");
                std::terminate();
            }
            // ...
    };
}

static const my::string global("...");

int main() {
    // ...
}

```

Noncompliant Code Example

In this noncompliant example, an exception may be thrown by the initializer for the static global variable `i`.

```

extern int f() noexcept(false);
int i = f();

int main() {
    // ...
}

```

Compliant Solution

This compliant solution wraps the call to `f()` with a helper function that catches all exceptions and terminates the program in conformance with [ERR50-CPP-EX1](#) of [ERR50-CPP. Do not abruptly terminate the program](#).

```
#include <exception>
```

```

int f_helper() noexcept {
    try {
        extern int f() noexcept(false);
        return f();
    } catch (...) {
        extern void log_message(const char *) noexcept;
        log_message("f() threw an exception");
        std::terminate();
    }
    // Unreachable.
}

int i = f_helper();

int main() {
    // ...
}

```

Risk Assessment

Throwing an exception that cannot be caught results in [abnormal program termination](#) and can lead to [denial-of-service attacks](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR58-CPP	Low	Likely	Low	P9	L2

Related Guidelines

This rule is a subset of [ERR50-CPP. Do not abruptly terminate the program](#)

SEI CERT C++ Coding Standard	DCL57-CPP. Do not let exceptions escape from destructors or deallocation functions ERR55-CPP. Honor exception specifications
--	---

Bibliography

ISO/IEC 14882-2014	Subclause 15.4, "Exception Specifications"
Sutter 2000	Item 8, "Writing Exception-Safe Code—Part 1"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR58-CPP.+Handle+all+exceptions+thrown+before+main%28%29+begins+executing>], Copyright [C] 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	9
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low
report_only_one_exception_per_function	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
report_only_uncaught	Whether the check shall report all throws or just those not caught.	False

Possible Messages

Name	Message
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionRaisedInInitialization	Exception raised in initialization or finalization

CertC++-ERR61

Catch exceptions by lvalue reference.

Input: IR

Details

When an exception is thrown, the value of the object in the throw expression is used to initialize an anonymous temporary object called the *exception object*. The type of this exception object is used to transfer control to the nearest catch handler, which contains an exception declaration with a matching type. The C++ Standard, [except.handle], paragraph 16 [[ISO/IEC 14882-2014](#)], in part, states the following:

- 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 from the corresponding base class subobject of the exception object;
 - otherwise, the variable is copy-initialized from the exception object.

Because the variable declared by the *exception-declaration* is copy-initialized, it is possible to *slice* the exception object as part of the copy operation, losing valuable exception information and leading to incorrect error recovery. For more information about object slicing, see [OOP51-CPP. Do not slice derived objects](#). Further, if the copy constructor of the exception object throws an exception, the copy initialization of the *exception-declaration* object results in undefined behavior. (See [ERR60-CPP. Exception objects must be nothrow copy constructible](#) for more information.)

Always catch exceptions by *lvalue* reference unless the type is a trivial type. For reference, the C++ Standard, [basic.types], paragraph 9 [[ISO/IEC 14882-2014](#)], defines trivial types as the following:

Arithmetic types, enumeration types, pointer types, pointer to member types, `std::nullptr_t`, and cv-qualified versions of these types are collectively called *scalar types*.... Scalar types, trivial class types, arrays of such types and cv-qualified versions of these types are collectively called *trivial types*.

The C++ Standard, [class], paragraph 6, defines trivial class types as the following:

- A *trivially copyable class* is a class that:
- has no non-trivial copy constructors,
 - has no non-trivial move constructors,
 - has no non-trivial copy assignment operators,
 - has no non-trivial move assignment operators, and
 - has a trivial destructor.

A *trivial class* is a class that has a default constructor, 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*]

Noncompliant Code Example

In this noncompliant code example, an object of type *s* is used to initialize the exception object that is later caught by an *exception-declaration* of type `std::exception`. The *exception-declaration* matches the exception object type, so the variable *e* is copy-initialized from the exception object, resulting in the exception object being sliced. Consequently, the output of this noncompliant code example is the implementation-defined value returned from calling `std::exception::what()` instead of "My custom exception".

```
#include <exception>
#include <iostream>

struct S : std::exception {
    const char *what() const noexcept override {
        return "My custom exception";
    }
};

void f() {
    try {
        throw S();
    } catch (std::exception e) {
        std::cout << e.what() << std::endl;
    }
}
```

Compliant Solution

In this compliant solution, the variable declared by the *exception-declaration* is an lvalue reference. The call to `what()` results in executing `s::what()` instead of `std::exception::what()`.

```
#include <exception>
#include <iostream>

struct S : std::exception {
    const char *what() const noexcept override {
        return "My custom exception";
    }
};

void f() {
    try {
        throw S();
    } catch (std::exception &e) {
        std::cout << e.what() << std::endl;
    }
}
```

Risk Assessment

Object slicing can result in abnormal program execution. This generally is not a problem for exceptions, but it can lead to unexpected behavior depending on the assumptions made by the exception handler.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR61-CPP	Low	Unlikely	Low	P3	L3

Related Guidelines

This rule is a subset of [OOP51-CPP. Do not slice derived objects.](#)

SEI CERT C++ Coding Standard	ERR60-CPP. Exception objects must be nothrow copy constructible.
--	--

Bibliography

[ISO/IEC 14882-2014]	Subclause 3.9, "Types" Clause 9, "Classes" Subclause 15.1, "Throwing an Exception" Subclause 15.3, "Handling an Exception"
[MISRA 2008]	Rule 15-3-5

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR61-CPP.+Catch+exceptions+by+lvalue+reference>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
only_class_types	Whether all types should be caught by reference or only class types. If set to True, non-class types (e.g. primitive types) may be caught by value.	False
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
catch_without_reference	Exceptions shall be caught by reference.

CertC++-ERR62

Detect errors when converting a string to a number.

Input: IR

Source languages: C++

Details

The process of parsing an integer or floating-point number from a string can produce many errors. The string might not contain a number. It might contain a number of the correct type that is out of range (such as an integer that is larger than `INT_MAX`). The string may also contain extra information after the number, which may or may not be useful after the conversion. These error conditions must be detected and addressed when a string-to-number conversion is performed using a formatted input stream such as `std::istream` or the locale facet `num_get<>`.

When calling a formatted input stream function like `istream::operator>>()`, information about conversion errors is queried through the `basic_ios::good()`, `basic_ios::bad()`, and `basic_ios::fail()` inherited member functions or through exception handling if it is enabled on the stream object.

When calling `num_get<>::get()`, information about conversion errors is returned to the caller through the `ios_base::iostate&` argument. The C++ Standard, section [facet.num.get.virtuals], paragraph 3 [[ISO/IEC 14882-2014](#)], in part, states the following:

If the conversion function fails to convert the entire field, or if the field represents a value outside the range of representable values, `ios_base::failbit` is assigned to `err`.

Always explicitly check the error state of a conversion from string to a numeric value (or handle the related exception, if applicable) instead of assuming the conversion results in a valid value. This rule is in addition to [ERR34-C. Detect errors when converting a string to a number](#), which bans the use of conversion functions that do not perform conversion validation such as `std::atoi()` and `std::scanf()` from the C Standard Library.

Noncompliant Code Example

In this noncompliant code example, multiple numeric values are converted from the standard input stream. However, if the text received from the standard input stream cannot be converted into a numeric value that can be represented by an `int`, the resulting value stored into the variables `i` and `j` may be unexpected.

```
#include <iostream>

void f() {
    int i, j;
    std::cin >> i >> j;
    // ...
}
```

For instance, if the text `12345678901234567890` is the first converted value read from the standard input stream, then `i` will have the value `std::numeric_limits<int>::max()` (per [facet.num.get.virtuals] paragraph 3), and `j` will be uninitialized (per [istream.formatted.arithmetic] paragraph 3). If the text `abcdefg` is the first converted value read from the standard input stream, then `i` will have the value `0` and `j` will remain uninitialized.

Compliant Solution

In this compliant solution, exceptions are enabled so that any conversion failure results in an exception being thrown. However, this approach cannot distinguish between which values are valid and which values are invalid and must assume that all values are invalid. Both the `badbit` and `failbit` flags are set to ensure that conversion errors as well as loss of integrity with the stream are treated as exceptions.

```
#include <iostream>

void f() {
    int i, j;

    std::cin.exceptions(std::istream::failbit | std::istream::badbit);
    try {
        std::cin >> i >> j;
        // ...
    } catch (std::istream::failure &E) {
        // Handle error
    }
}
```

Compliant Solution

In this compliant solution, each converted value read from the standard input stream is tested for validity before reading the next value in the sequence, allowing error recovery on a per-value basis. It checks `std::istream::fail()` to see if the failure bit was set due to a conversion failure or whether the bad bit was set due to a loss of integrity with the stream object. If a failure condition is encountered, it is cleared on the input stream and then characters are read and discarded until a `' '` (space) character occurs. The error handling in this case only works if a space character is what delimits the two numeric values to be converted.

```
#include <iostream>
#include <limits>

void f() {
    int i;
    std::cin >> i;
    if (std::cin.fail()) {
        // Handle failure to convert the value.
        std::cin.clear();
        std::cin.ignore(std::numeric_limits::max(), ' ');
    }

    int j;
    std::cin >> j;
    if (std::cin.fail()) {
        std::cin.clear();
        std::cin.ignore(std::numeric_limits::max(), ' ');
    }

    // ...
}
```

Risk Assessment

It is rare for a violation of this rule to result in a security [vulnerability](#) unless it occurs in security-sensitive code. However, violations of this rule can easily result in lost or misinterpreted data.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR62-CPP	Medium	Unlikely	Medium	P4	L3

Related Guidelines

SEI CERT C Coding Standard	ERR34-C. Detect errors when converting a string to a number
MITRE CWE	CWE-676 , Use of potentially dangerous function CWE-20 , Insufficient input validation

Bibliography

[ISO/IEC 9899:1999]	Subclause 7.22.1, "Numeric conversion functions" Subclause 7.21.6, "Formatted input/output functions"
[ISO/IEC 14882-2014]	Subclause 22.4.2.1.1, "num_get members" Subclause 27.7.2.2, "Formatted input functions"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR62-CPP.+Detect+errors+when+converting+a+string+to+a+number>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
check_functions	Fully qualified names of functions to check the state of the input stream.	['std::basic_ios::fail', 'std::ios_base::fail']
functions_to_check	Fully qualified names of functions after which a call to one of the check functions is required	['std::basic_istream::operator>>']
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
missing_basic_ios_fail	Use basic_ios::fail() after reading from input streams.

CertC++-OOP50

Do not invoke virtual functions from constructors or destructors.

Input: IR

Source languages: C++

Details

Virtual functions allow for the choice of member function calls to be determined at run time based on the dynamic type of the object that the member function is being called on. This convention supports object-oriented programming practices commonly associated with object inheritance and function overriding. When calling a nonvirtual member function or when using a class member access expression to denote a call, the specified function is called. Otherwise, a virtual function call is made to the final overrider in the dynamic type of the object expression.

However, during the construction and destruction of an object, the rules for virtual method dispatch on that object are restricted. The C++ Standard, [class.cdtor], paragraph 4 [\[ISO/IEC 14882-2014\]](#), states the following:

Member functions, including virtual functions, can be called during construction or destruction. 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 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.

Do not directly or indirectly invoke a virtual function from a constructor or destructor that attempts to call into the object under construction or destruction. Because the order of construction starts with base classes and moves to more derived classes, attempting to call a derived class function from a base class under construction is dangerous. The derived class has not had the opportunity to initialize its resources, which is why calling a virtual function from a constructor does not result in a call to a function in a more derived class. Similarly, an object is destroyed in reverse order from construction, so attempting to call a function in a more derived class from a destructor may access resources that have already been released.

Noncompliant Code Example

In this noncompliant code example, the base class attempts to seize and release an object's resources through calls to virtual functions from the constructor and destructor. However, the `B::B()` constructor calls `B::seize()` rather than `D::seize()`. Likewise, the `B::~B()` destructor calls `B::release()` rather than `D::release()`.

```
struct B {
    B() { seize(); }
    virtual ~B() { release(); }
protected:
```

```

virtual void seize();
virtual void release();
};

struct D : B {
    virtual ~D() = default;

protected:
    void seize() override {
        B::seize();
        // Get derived resources...
    }

    void release() override {
        // Release derived resources...
        B::release();
    }
};

```

The result of running this code is that no derived class resources will be seized or released during the initialization and destruction of object of type `D`. At the time of the call to `seize()` from `B::B()`, the `D` constructor has not been entered, and the behavior of the under-construction object will be to invoke `B::seize()` rather than `D::seize()`. A similar situation occurs for the call to `release()` in the base class destructor. If the functions `seize()` and `release()` were declared to be pure virtual functions, the result would be [undefined behavior](#).

Compliant Solution

In this compliant solution, the constructors and destructors call a nonvirtual, private member function (suffixed with `_mine`) instead of calling a virtual function. The result is that each class is responsible for seizing and releasing its own resources.

```

class B {
    void seize_mine();
    void release_mine();

public:
    B() { seize_mine(); }
    virtual ~B() { release_mine(); }

protected:
    virtual void seize() { seize_mine(); }
    virtual void release() { release_mine(); }
};

class D : public B {
    void seize_mine();
    void release_mine();

public:
    D() { seize_mine(); }
    virtual ~D() { release_mine(); }

protected:
    void seize() override {
        B::seize();
        seize_mine();
    }

    void release() override {
        release_mine();
        B::release();
    }
};

```

Exceptions

OOP50-CPP-EX1: Because valid use cases exist that involve calling (non-pure) virtual functions from the constructor of a class, it is permissible to call the virtual function with an explicitly qualified ID. The qualified ID signifies to code maintainers that the expected behavior is for the class under construction or destruction to be the final overrider for the function call.

```

struct A {
    A() {
        // f();      // WRONG!
        A::f();    // Okay
    }
    virtual void f();
};

```

OOP50-CPP-EX2: It is permissible to call a virtual function that has the `final` *virt-specifier* from a constructor or destructor, as in this example.

```

struct A {
    A();
    virtual void f();
};

struct B : A {
    B() : A() {
        f(); // Okay
    }
    void f() override final;
};

```

Similarly, it is permissible to call a virtual function from a constructor or destructor of a class that has the `final` *class-virt-specifier*, as in this example.

```

struct A {
    A();
    virtual void f();
};

```

```

};

struct B final : A {
    B() : A() {
        f(); // Okay
    }
    void f() override;
};

```

In either case, `f()` must be the final overrider, guaranteeing consistent behavior of the function being called.

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
OOP50-CPP	Low	Unlikely	Medium	P2	L3

Bibliography

[Dewhurst 2002]	Gotcha #75, "Calling Virtual Functions in Constructors and Destructors"
[ISO/IEC 14882-2014]	Subclause 5.5, "Pointer-to-Member Operators"
[Lockheed Martin 2005]	AV Rule 71.1, "A class's virtual functions shall not be invoked from its destructor or any of its constructors"
[Sutter 2004]	Item 49, "Avoid Calling Virtual Functions in Constructors and Destructors"

Excerpt from SEI CERT C++ Coding Standard Wiki <https://wiki.sei.cmu.edu/confluence/display/cplusplus/OOP50-CPP.+Do+not+invoke+virtual+functions+from+constructors+or+destructors>, Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
constructor_using_virtual_call	Virtual call used in constructor/destructor.

CertC++-OOP52

Do not delete a polymorphic object without a virtual destructor.

Input: IR

Source languages: C++

Details

The C++ Standard, [expr.delete], paragraph 3 [\[ISO/IEC 14882-2014\]](#), states the following:

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.

Do not delete an object of derived class type through a pointer to its base class type that has a non-virtual destructor. Instead, the base class should be defined with a virtual destructor. Deleting an object through a pointer to a type without a virtual destructor results in [undefined behavior](#).

Noncompliant Code Example

In this noncompliant example, `b` is a polymorphic pointer type whose static type is `Base *` and whose dynamic type is `Derived *`. When `b` is deleted, it results in [undefined behavior](#) because `Base` does not have a virtual destructor. The C++ Standard, [class.dtor], paragraph 4 [\[ISO/IEC 14882-2014\]](#), states the following:

If a class has no user-declared destructor, a destructor is implicitly declared as defaulted. An implicitly declared destructor is an `inline`

```
public member of its class.
```

The implicitly declared destructor is not declared as `virtual` even in the presence of other `virtual` functions.

```
struct Base {  
    virtual void f();  
};  
  
struct Derived : Base {};  
  
void f() {  
    Base *b = new Derived();  
    // ...  
    delete b;  
}
```

Noncompliant Code Example

In this noncompliant example, the explicit pointer operations have been replaced with a smart pointer object, demonstrating that smart pointers suffer from the same problem as other pointers. Because the default deleter for `std::unique_ptr` calls `delete` on the internal pointer value, the resulting behavior is identical to the previous noncompliant example.

```
#include <memory>  
  
struct Base {  
    virtual void f();  
};  
  
struct Derived : Base {};  
  
void f() {  
    std::unique_ptr<Base> b = std::make_unique<Derived>();  
}
```

Compliant Solution

In this compliant solution, the destructor for `Base` has an explicitly declared `virtual` destructor, ensuring that the polymorphic delete operation results in well-defined behavior.

```
struct Base {  
    virtual ~Base() = default;  
    virtual void f();  
};  
  
struct Derived : Base {};  
  
void f() {  
    Base *b = new Derived();  
    // ...  
    delete b;  
}
```

Exceptions

OOP52-CPP:EX0: Deleting a polymorphic object without a `virtual` destructor is permitted if the object is referenced by a pointer to its class, rather than via a pointer to a class it inherits from.

```
class Base {  
public:  
    // ...  
    virtual void AddRef() = 0;  
    virtual void Destroy() = 0;  
};  
  
class Derived final : public Base {  
public:  
    // ...  
    virtual void AddRef() { /* ... */ }  
    virtual void Destroy() { delete this; }  
private:  
    ~Derived() {}  
};
```

Note that if `Derived` were not marked as `final`, then `delete this` could actually reference a subclass of `Derived`, violating this rule.

Risk Assessment

Attempting to destruct a polymorphic object that does not have a `virtual` destructor declared results in [undefined behavior](#). In practice, potential consequences include [abnormal program termination](#) and memory leaks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
OOP52-CPP	Low	Likely	Low	P9	L2

Related Guidelines

Bibliography

[ISO/IEC 14882-2014]	Subclause 5.3.5, "Delete" Subclause 12.4, "Destructors"
[Stroustrup 2006]	"Why Are Destructors Not Virtual by Default?"

Excerpt from SEI CERT C++ Coding Standard Wiki <https://wiki.sei.cmu.edu/confluence/display/cplusplus/OOP52-CPP.+Do+not+delete+a+polymorphic+object+without+a+virtual+destructor>, Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	9
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
base_class_destructor	Destructor of a base class shall be public virtual, public override or protected non-virtual.
non_final	If a public destructor of a class is non-virtual, then the class should be declared final.

CertC++-OOP53

Write constructor member initializers in the canonical order.

Input: IR

Source languages: C++

Details

The member initializer list for a class constructor allows members to be initialized to specified values and for base class constructors to be called with specific arguments. However, the order in which initialization occurs is fixed and does not depend on the order written in the member initializer list. The C++ Standard, [class.base.init], paragraph 11 [\[ISO/IEC 14882-2014\]](#), states the following:

In a non-delegating constructor, initialization proceeds in the following order:

- First, and only for the constructor of the most derived class, 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]

Consequently, the order in which member initializers appear in the member initializer list is irrelevant. The order in which members are initialized, including base class initialization, is determined by the declaration order of the class member variables or the base class specifier list. Writing member initializers other than in canonical order can result in [undefined behavior](#), such as reading uninitialized memory.

Always write member initializers in a constructor in the canonical order: first, direct base classes in the order in which they appear in the *base-specifier-list* for the class, then nonstatic data members in the order in which they are declared in the class definition.

Noncompliant Code Example

In this noncompliant code example, the member initializer list for `c::c()` attempts to initialize `someVal` first and then to initialize `dependsOnSomeVal` to a value dependent on `someVal`. Because the declaration order of the member variables does not match the member initializer order, attempting to read the value of `someVal` results in an [unspecified value](#) being stored into `dependsOnSomeVal`.

```
class C {
    int dependsOnSomeVal;
    int someVal;
```

```

public:
    C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}
};

```

Compliant Solution

This compliant solution changes the declaration order of the class member variables so that the dependency can be ordered properly in the constructor's member initializer list.

```

class C {
    int someVal;
    int dependsOnSomeVal;

public:
    C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}
};

```

It is reasonable for initializers to depend on previously initialized values.

Noncompliant Code Example

In this noncompliant code example, the derived class, `D`, attempts to initialize the base class, `B1`, with a value obtained from the base class, `B2`. However, because `B1` is initialized before `B2` due to the declaration order in the base class specifier list, the resulting behavior is [undefined](#).

```

class B1 {
    int val;

public:
    B1(int val) : val(val) {}

class B2 {
    int otherVal;

public:
    B2(int otherVal) : otherVal(otherVal) {}
    int get_other_val() const { return otherVal; }

class D : B1, B2 {
public:
    D(int a) : B2(a), B1(get_other_val()) {}
};

```

Compliant Solution

This compliant solution initializes both base classes using the same value from the constructor's parameter list instead of relying on the initialization order of the base classes.

```

class B1 {
    int val;

public:
    B1(int val) : val(val) {}

class B2 {
    int otherVal;

public:
    B2(int otherVal) : otherVal(otherVal) {}

class D : B1, B2 {
public:
    D(int a) : B1(a), B2(a) {}
};

```

Exceptions

OOP53-CPP-EX0: Constructors that do not use member initializers do not violate this rule.

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
OOP53-CPP	Medium	Unlikely	Medium	P4	L3

Bibliography

[ISO/IEC 14882-2014]	Subclause 12.6.2, "Initializing Bases and Members"
[Lockheed Martin 2005]	AV Rule 75, Members of the initialization list shall be listed in the order in which they are declared in the class

Excerpt from SEI CERT C++ Coding Standard Wiki [\[https://wiki.sei.cmu.edu/confluence/display/cplusplus/OOP53-CPP.+Write+constructor+member+initializers+in+the+canonical+order\]](https://wiki.sei.cmu.edu/confluence/display/cplusplus/OOP53-CPP.+Write+constructor+member+initializers+in+the+canonical+order), Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#)

for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
reported_messages	If provided, only messages of these types are reported.	[1719]
reported_severities	List of severities to display.	('error', 'warning', 'remark')
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

CertC++-CON40

Do not refer to an atomic variable twice in an expression.

Input: IR

Source languages: C++

Details

A consistent locking policy guarantees that multiple threads cannot simultaneously access or modify shared data. Atomic variables eliminate the need for locks by guaranteeing thread safety when certain operations are performed on them. The thread-safe operations on atomic variables are specified in the C Standard, subclauses 7.17.7 and 7.17.8 [[ISO/IEC 9899:2011](#)]. While atomic operations can be combined, combined operations do not provide the thread safety provided by individual atomic operations.

Every time an atomic variable appears on the left side of an assignment operator, including a compound assignment operator such as *=, an atomic write is performed on the variable. The use of the increment (++) or decrement (--) operators on an atomic variable constitutes an atomic read-and-write operation and is consequently thread-safe. Any reference of an atomic variable anywhere else in an expression indicates a distinct atomic read on the variable.

If the same atomic variable appears twice in an expression, then two atomic reads, or an atomic read and an atomic write, are required. Such a pair of atomic operations is not thread-safe, as another thread can modify the atomic variable between the two operations. Consequently, an atomic variable must not be referenced twice in the same expression.

Noncompliant Code Example [atomic_bool]

This noncompliant code example declares a shared `atomic_bool flag` variable and provides a `toggle_flag()` method that negates the current value of `flag`:

```
#include <stdatomic.h>
#include <stdbool.h>

static atomic_bool flag = ATOMIC_VAR_INIT(false);

void init_flag(void) {
    atomic_init(&flag, false);
}

void toggle_flag(void) {
    bool temp_flag = atomic_load(&flag);
    temp_flag = !temp_flag;
    atomic_store(&flag, temp_flag);
}

bool get_flag(void) {
    return atomic_load(&flag);
}
```

Execution of this code may result in unexpected behavior because the value of `flag` is read, negated, and written back. This occurs even though the read and write are both atomic.

Consider, for example, two threads that call `toggle_flag()`. The expected effect of toggling `flag` twice is that it is restored to its original value. However, the scenario in the following table leaves `flag` in the incorrect state.

`toggle_flag()` without Compare-and-Exchange

Time	flag	Thread	Action
1	true	t_1	Reads the current value of flag, which is true, into a cache
2	true	t_2	Reads the current value of flag, which is still true, into a different cache
3	true	t_1	Toggles the temporary variable in the cache to false
4	true	t_2	Toggles the temporary variable in the different cache to false
5	false	t_1	Writes the cache variable's value to flag
6	false	t_2	Writes the different cache variable's value to flag

As a result, the effect of the call by t_2 is not reflected in flag; the program behaves as if toggle_flag() was called only once, not twice.

Compliant Solution (`atomic_compare_exchange_weak()`)

This compliant solution uses a compare-and-exchange to guarantee that the correct value is stored in flag. All updates are visible to other threads. The call to atomic_compare_exchange_weak() is in a loop in conformance with [CON41-C. Wrap functions that can fail spuriously in a loop](#).

```
#include <stdatomic.h>
#include <stdbool.h>

static atomic_bool flag = ATOMIC_VAR_INIT(false);

void init_flag(void) {
    atomic_init(&flag, false);
}

void toggle_flag(void) {
    bool old_flag = atomic_load(&flag);
    bool new_flag;
    do {
        new_flag = !old_flag;
    } while (!atomic_compare_exchange_weak(&flag, &old_flag, new_flag));
}

bool get_flag(void) {
    return atomic_load(&flag);
}
```

An alternative solution is to use the `atomic_flag` data type for managing Boolean values atomically. However, `atomic_flag` does not support a toggle operation.

Compliant Solution (Compound Assignment)

This compliant solution uses the `^=` assignment operation to toggle flag. This operation is guaranteed to be atomic, according to the C Standard, 6.5.16.2, paragraph 3. This operation performs a bitwise-exclusive-or between its arguments, but for Boolean arguments, this is equivalent to negation.

```
#include <stdatomic.h>
#include <stdbool.h>

static atomic_bool flag = ATOMIC_VAR_INIT(false);

void toggle_flag(void) {
    flag ^= 1;
}

bool get_flag(void) {
    return flag;
}
```

An alternative solution is to use a mutex to protect the atomic operation, but this solution loses the performance benefits of atomic variables.

Noncompliant Code Example

This noncompliant code example takes an atomic global variable n and computes $n + (n - 1) + (n - 2) + \dots + 1$, using the formula $n * (n + 1) / 2$:

```
#include <stdatomic.h>

atomic_int n = ATOMIC_VAR_INIT(0);

int compute_sum(void) {
    return n * (n + 1) / 2;
}
```

The value of n may change between the two atomic reads of n in the expression, yielding an incorrect result.

Compliant Solution

This compliant solution passes the atomic variable as a function argument, forcing the variable to be copied and guaranteeing a correct result. Note that the

function's formal parameter need not be atomic, and the atomic variable can still be passed as an actual argument.

```
#include <stdatomic.h>

int compute_sum(int n) {
    return n * (n + 1) / 2;
}
```

Risk Assessment

When operations on atomic variables are assumed to be atomic, but are not atomic, surprising data races can occur, leading to corrupted data and invalid control flow.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON40-C	Medium	Probable	Medium	P8	L2

Related Guidelines

MITRE CWE	CWE-366 , Race Condition within a Thread CWE-413 , Improper Resource Locking CWE-567 , Unsynchronized Access to Shared Data in a Multithreaded Context CWE-667 , Improper Locking
-----------	--

Bibliography

[ISO/IEC 9899:2011]	6.5.16.2, "Compound Assignment" 7.17, "Atomics"
-------------------------------------	--

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/fwBnBw>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
statements_to_look_ahead	Specifies after how many statements of an atomic_load an atomic_store may appear. Default is 3.	3

Possible Messages

Name	Message
atomics_used_twice	Do not refer to an atomic variable twice in an expression.
unsafe_atomic_load_store	'atomic_store' after an 'atomic_load' is unsafe in multi-threaded environments.

CertC++-CON52

Prevent data races when accessing bit-fields from multiple threads.

Input: IR

Source languages: C++

Details

When accessing a bit-field, a thread may inadvertently access a separate bit-field in adjacent memory. This is because compilers are required to store multiple adjacent bit-fields in one storage unit whenever they fit. Consequently, data races may exist not just on a bit-field accessed by multiple threads but also on other bit-fields sharing the same byte or word. The problem is difficult to diagnose because it may not be obvious that the same memory location is being modified by multiple threads.

One approach for preventing data races in concurrent programming is to use a mutex. When properly observed by all threads, a mutex can provide safe and

secure access to a shared object. However, mutexes provide no guarantees with regard to other objects that might be accessed when the mutex is not controlled by the accessing thread. Unfortunately, there is no portable way to determine which adjacent bit-fields may be stored along with the desired bit-field.

Another approach is to insert a non-bit-field member between any two bit-fields to ensure that each bit-field is the only one accessed within its storage unit. This technique effectively guarantees that no two bit-fields are accessed simultaneously.

Noncompliant Code Example {bit-field}

Adjacent bit-fields may be stored in a single memory location. Consequently, modifying adjacent bit-fields in different threads is [undefined behavior](#), as shown in this noncompliant code example.

```
struct MultiThreadedFlags {
    unsigned int flag1 : 2;
    unsigned int flag2 : 2;
};

MultiThreadedFlags flags;

void thread1() {
    flags.flag1 = 1;
}

void thread2() {
    flags.flag2 = 2;
}
```

For example, the following instruction sequence is possible.

```
Thread 1: register 0 = flags
Thread 1: register 0 &= ~mask(flag1)
Thread 2: register 0 = flags
Thread 2: register 0 &= ~mask(flag2)
Thread 1: register 0 |= 1 << shift(flag1)
Thread 1: flags = register 0
Thread 2: register 0 |= 2 << shift(flag2)
Thread 2: flags = register 0
```

Compliant Solution {bit-field, C++11 and later, mutex}

This compliant solution protects all accesses of the flags with a mutex, thereby preventing any data races.

```
#include <mutex>

struct MultiThreadedFlags {
    unsigned int flag1 : 2;
    unsigned int flag2 : 2;
};

struct MtfMutex {
    MultiThreadedFlags s;
    std::mutex mutex;
};

MtfMutex flags;

void thread1() {
    std::lock_guard<std::mutex> lk(flags.mutex);
    flags.s.flag1 = 1;
}

void thread2() {
    std::lock_guard<std::mutex> lk(flags.mutex);
    flags.s.flag2 = 2;
}
```

Compliant Solution {C++11}

In this compliant solution, two threads simultaneously modify two distinct non-bit-field members of a structure. Because the members occupy different bytes in memory, no concurrency protection is required.

```
struct MultiThreadedFlags {
    unsigned char flag1;
    unsigned char flag2;
};

MultiThreadedFlags flags;

void thread1() {
    flags.flag1 = 1;
}

void thread2() {
    flags.flag2 = 2;
}
```

Unlike earlier versions of the standard, C++11 and later explicitly define a memory location and provide the following note in [intro.memory] paragraph 4 [[ISO/IEC 14882-2014](#)]:

[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*]

It is almost certain that `flag1` and `flag2` are stored in the same word. Using a compiler that conforms to earlier versions of the standard, if both assignments occur on a thread-scheduling interleaving that ends with both stores occurring after one another, it is possible that only one of the flags will be set as intended, and the other flag will contain its previous value because both members are represented by the same word, which is the smallest unit the processor can work on. Before the changes made to the C++ Standard for C++11, there were no guarantees that these flags could be modified concurrently.

Risk Assessment

Although the race window is narrow, an assignment or an expression can evaluate improperly because of misinterpreted data resulting in a corrupted running state or unintended information disclosure.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON52-CPP	Medium	Probable	Medium	P8	L2

Related Guidelines

SEI CERT C Coding Standard	CON32-C. Prevent data races when accessing bit-fields from multiple threads
--	---

Bibliography

[ISO/IEC 14882-2014]	Subclause 1.7, "The C++ memory model"
--------------------------------------	---------------------------------------

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/CON52-CPP+Prevent+data+races+when+accessing+bit-fields+from+multiple+threads>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
access_kinds	Access kinds (e.g. Reading_Operand_Interface, Writing_Operand_Interface, Address_Operand_Interface).	['Reading_Operand_Interface', 'Writing_Operand_Interface']
allow_c11_atomics	If set, don't report races on C11 atomic variables.	True
allow_volatile_sig_atomic_t	If set, don't report races on variables of type "volatile sig_atomic_t".	False
debug_output	Option to provide diagnostic output.	False
enter_critical_functions	List of function names to enter a critical region.	[]
enter_critical_macros	List of macro names to enter a critical region (macros must expand to asm() statement).	[]
excluded_routines	List of functions that should be excluded from check.	[]
excluded_subgraphs	List of entry functions to subgraphs that should be excluded as subgraph from check.	[]
exit_critical_functions	List of function names to exit a critical region.	[]
exit_critical_macros	List of macro names to exit a critical region (macros must expand to asm() statement).	[]
inspect_pointers	Whether pointer targets should be inspected to detect more global variable uses.	False
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
nested_critical_regions	If set to True, critical regions nest; if set to False, a single exit-critical-region terminates all open critical regions.	True
output_safe_accesses	When enabled, outputs not only unsafe variable accesses, but also the safe ones.	False
partitions	Dict with partition name as key and dict as value which may contain keys 'entries' and/or 'vectors' with lists of entry points or vector table variables respectively. If special partition '*IRQ*' to configure interrupt handlers is missing, all functions not reached by any of the other options are treated as interrupt handlers.	dict(...)
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium
report_cfg_based_critical_region_issues	Report unbalanced lock/unlock pairs within a routine. This has the same intention, but is slightly less strict than the purely syntactic check performed by the rule Parallelism-IncorrectCriticalSection.	False
show_identical_access	When enabled, outputs variable accesses of same kind (i.e., R/R and W/W).	True
show_object_number	Option for debugging (shows internal node numbers).	False
treat_types_as_atomic	List of type-patterns. A type-pattern is either a regular expression of a type name, or a triple of {min. alignment, max. size, type name-regex}. Each of the triple's components may be None. None is interpreted as general wildcard.	[]

Possible Messages

Name	Message
data-race-on-bitfields	Prevent data races when accessing bit-fields from multiple threads.
multiple_lock_add	Lock is acquired while it is already locked.
removed_nonexisting_lock	Lock is released, although it is not currently locked.
unbalanced_locks_path	Different control flow paths have different sets of locks.
unbalanced_locks_routine	Routine may return with different lock set than it is entered with {{in_set} vs {out_set}}.

CertC++-ENV30

Do not modify the object referenced by the return value of certain functions.

Input: IR

Source languages: C++

Details

Some functions return a pointer to an object that cannot be modified without causing [undefined behavior](#). These functions include `getenv()`, `setlocale()`, `localeconv()`, `asctime()`, and `strerror()`. In such cases, the function call results must be treated as being `const`-qualified.

The C Standard, 7.22.4.6, paragraph 4 [[ISO/IEC 9899:2011](#)], defines `getenv()` as follows:

The `getenv` function returns a pointer to a string associated with the matched list member. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `getenv` function. If the specified name cannot be found, a null pointer is returned.

If the string returned by `getenv()` must be altered, a local copy should be created. Altering the string returned by `getenv()` is [undefined behavior](#). (See [undefined behavior 184](#).)

Similarly, subclause 7.11.1.1, paragraph 8 [[ISO/IEC 9899:2011](#)], defines `setlocale()` as follows:

The pointer to string returned by the `setlocale` function is such that a subsequent call with that string value and its associated category will restore that part of the program's locale. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `setlocale` function.

And subclause 7.11.2.1, paragraph 8 [[ISO/IEC 9899:2011](#)], defines `localeconv()` as follows:

The `localeconv` function returns a pointer to the filled-in object. The structure pointed to by the return value shall not be modified by the program, but may be overwritten by a subsequent call to the `localeconv` function. In addition, calls to the `setlocale` function with categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC` may overwrite the contents of the structure.

Altering the string returned by `setlocale()` or the structure returned by `localeconv()` are [undefined behaviors](#). (See [undefined behaviors 120](#) and [121](#).) Furthermore, the C Standard imposes no requirements on the contents of the string by `setlocale()`. Consequently, no assumptions can be made as to the string's internal contents or structure.

Finally, subclause 7.24.6.2, paragraph 4 [[ISO/IEC 9899:2011](#)], states

The `strerror` function returns a pointer to the string, the contents of which are locale-specific. The array pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `strerror` function.

Altering the string returned by `strerror()` is [undefined behavior](#). (See [undefined behavior 184](#).)

Noncompliant Code Example (`getenv()`)

This noncompliant code example modifies the string returned by `getenv()` by replacing all double quotation marks ("") with underscores (_):

```
#include <stdlib.h>

void trstr(char *c_str, char orig, char rep) {
    while (*c_str != '\0') {
        if (*c_str == orig) {
            *c_str = rep;
        }
        ++c_str;
    }
}

void func(void) {
    char *env = getenv("TEST_ENV");
    if (env == NULL) {
        /* Handle error */
    }
    trstr(env, "'", '_');
}
```

Compliant Solution (`getenv()`) (Environment Not Modified)

If the programmer does not intend to modify the environment, this compliant solution demonstrates how to modify a copy of the return value:

```
#include <stdlib.h>
```

```

#include <string.h>

void trstr(char *c_str, char orig, char rep) {
    while (*c_str != '\0') {
        if (*c_str == orig) {
            *c_str = rep;
        }
        ++c_str;
    }
}

void func(void) {
    const char *env;
    char *copy_of_env;

    env = getenv("TEST_ENV");
    if (env == NULL) {
        /* Handle error */
    }

    copy_of_env = (char *)malloc(strlen(env) + 1);
    if (copy_of_env == NULL) {
        /* Handle error */
    }

    strcpy(copy_of_env, env);
    trstr(copy_of_env, "'", '_');
    /* ... */
    free(copy_of_env);
}

```

Compliant Solution {getenv()} {Modifying the Environment in POSIX}

If the programmer's intent is to modify the environment, this compliant solution, which saves the altered string back into the environment by using the POSIX `setenv()` and `strdup()` functions, can be used:

```

#include <stdlib.h>
#include <string.h>

void trstr(char *c_str, char orig, char rep) {
    while (*c_str != '\0') {
        if (*c_str == orig) {
            *c_str = rep;
        }
        ++c_str;
    }
}

void func(void) {
    const char *env;
    char *copy_of_env;

    env = getenv("TEST_ENV");
    if (env == NULL) {
        /* Handle error */
    }

    copy_of_env = strdup(env);
    if (copy_of_env == NULL) {
        /* Handle error */
    }

    trstr(copy_of_env, "'", '_');

    if (setenv("TEST_ENV", copy_of_env, 1) != 0) {
        /* Handle error */
    }
    /* ... */
    free(copy_of_env);
}

```

Noncompliant Code Example {localeconv()}

In this noncompliant example, the object returned by `localeconv()` is directly modified:

```

#include <locale.h>

void f2(void) {
    struct lconv *conv = localeconv();

    if ('\0' == conv->decimal_point[0]) {
        conv->decimal_point = ".";
    }
}

```

Compliant Solution {localeconv()} {Copy}

This compliant solution modifies a copy of the object returned by `localeconv()`:

```

#include <locale.h>
#include <stdlib.h>
#include <string.h>

void f2(void) {

```

```

const struct lconv *conv = localeconv();
if (conv == NULL) {
    /* Handle error */
}

struct lconv *copy_of_conv = (struct lconv *)malloc(
    sizeof(struct lconv));
if (copy_of_conv == NULL) {
    /* Handle error */
}

memcpy(copy_of_conv, conv, sizeof(struct lconv));

if ('\0' == copy_of_conv->decimal_point[0]) {
    copy_of_conv->decimal_point = ".";
}
/* ... */
free(copy_of_conv);
}

```

Risk Assessment

Modifying the object pointed to by the return value of `getenv()`, `setlocale()`, `localeconv()`, `asctime()`, or `strerror()` is [undefined behavior](#). Even if the modification succeeds, the modified object can be overwritten by a subsequent call to the same function.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ENV30-C	Low	Probable	Medium	P4	L3

Related Guidelines

ISO/IEC TS 17961:2013	Modifying the string returned by <code>getenv</code> , <code>localeconv</code> , <code>setlocale</code> , and <code>strerror</code> [libmod]
---------------------------------------	--

Bibliography

[IEEE Std 1003.1:2013]	XSH, System Interfaces, <code>getenv</code> XSH, System Interfaces, <code>setlocale</code> XSH, System Interfaces, <code>localeconv</code>
[ISO/IEC 9899:2011]	7.11.1.1, "The <code>setlocale</code> Function" 7.11.2.1, "The <code>localeconv</code> Function" 7.22.4.6, "The <code>getenv</code> Function" 7.24.6.2, "The <code>strerror</code> Function"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/XgA1>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
funcs		{'getenv', 'setlocale', 'localeconv', 'asctime', 'strerror'}
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	4
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
<code>nonconst_system_pointer_retrievals</code>	Return value should be assigned to a pointer to const-qualified type.
<code>string_of_system_pointer_modified</code>	Return value of call to system function should be considered const, including strings referenced by it

CertC++-ENV32

All exit handlers must return normally.

Input: IR

Details

The C Standard provides three functions that cause an application to terminate normally: `_Exit()`, `exit()`, and `quick_exit()`. These are collectively called *exit functions*. When the `exit()` function is called, or control transfers out of the `main()` entry point function, functions registered with `atexit()` are called (but not `at_quick_exit()`). When the `quick_exit()` function is called, functions registered with `at_quick_exit()` (but not `atexit()`) are called. These functions are collectively called *exit handlers*. When the `_Exit()` function is called, no exit handlers or signal handlers are called.

Exit handlers must terminate by returning. It is important and potentially safety-critical for all exit handlers to be allowed to perform their cleanup actions. This is particularly true because the application programmer does not always know about handlers that may have been installed by support libraries. Two specific issues include nested calls to an exit function and terminating a call to an exit handler by invoking `longjmp()`.

A nested call to an exit function is [undefined behavior](#). (See [undefined behavior 182](#).) This behavior can occur only when an exit function is invoked from an exit handler or when an exit function is called from within a signal handler. (See [SIG30-C. Call only asynchronous-safe functions within signal handlers](#).)

If a call to the `longjmp()` function is made that would terminate the call to a function registered with `atexit()`, the behavior is [undefined](#).

Noncompliant Code Example

In this noncompliant code example, the `exit1()` and `exit2()` functions are registered by `atexit()` to perform required cleanup upon program termination. However, if `some_condition` evaluates to true, `exit()` is called a second time, resulting in [undefined behavior](#).

```
#include <stdlib.h>

void exit1(void) {
    /* ... Cleanup code ... */
    return;
}

void exit2(void) {
    extern int some_condition;
    if (some_condition) {
        /* ... More cleanup code ... */
        exit(0);
    }
    return;
}

int main(void) {
    if (atexit(exit1) != 0) {
        /* Handle error */
    }
    if (atexit(exit2) != 0) {
        /* Handle error */
    }
    /* ... Program code ... */
    return 0;
}
```

Functions registered by the `atexit()` function are called in the reverse order from which they were registered. Consequently, if `exit2()` exits in any way other than by returning, `exit1()` will not be executed. The same may also be true for `atexit()` handlers installed by support libraries.

Compliant Solution

A function that is registered as an exit handler by `atexit()` must exit by returning, as in this compliant solution:

```
#include <stdlib.h>

void exit1(void) {
    /* ... Cleanup code ... */
    return;
}

void exit2(void) {
    extern int some_condition;
    if (some_condition) {
        /* ... More cleanup code ... */
    }
    return;
}

int main(void) {
    if (atexit(exit1) != 0) {
        /* Handle error */
    }
    if (atexit(exit2) != 0) {
        /* Handle error */
    }
    /* ... Program code ... */
    return 0;
}
```

Noncompliant Code Example

In this noncompliant code example, `exit1()` is registered by `atexit()` so that upon program termination, `exit1()` is called. The `exit1()` function jumps back to `main()` to return, with undefined results.

```
#include <stdlib.h>
#include <setjmp.h>
```

```

jmp_buf env;
int val;

void exit1(void) {
    longjmp(env, 1);
}

int main(void) {
    if (atexit(exit1) != 0) {
        /* Handle error */
    }
    if (setjmp(env) == 0) {
        exit(0);
    } else {
        return 0;
    }
}

```

Compliant Solution

This compliant solution does not call `longjmp()` but instead returns from the exit handler normally:

```

#include <stdlib.h>

void exit1(void) {
    return;
}

int main(void) {
    if (atexit(exit1) != 0) {
        /* Handle error */
    }
    return 0;
}

```

Risk Assessment

Terminating a call to an exit handler in any way other than by returning is [undefined behavior](#) and may result in [abnormal program termination](#) or other unpredictable behavior. It may also prevent other registered handlers from being invoked.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ENV32-C	Medium	Likely	Medium	P12	L1

Related Guidelines

CERT C Secure Coding Standard	SIG30-C. Call only asynchronous-safe functions within signal handlers
ISO/IEC TR 24772:2013	Structured Programming [EWD] Termination Strategy [REU]
MITRE CWE	CWE-705 , Incorrect Control Flow Scoping

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/v0Ag>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
exit_functions	Functions which should not be used in exit handlers.	['exit', 'quick_exit', 'longjmp']
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
non_normal_return_exit_handler	All exit handlers must return normally.

Do not call system().

Input: IR

Source languages: C++

Details

The C Standard `system()` function executes a specified command by invoking an [implementation-defined](#) command processor, such as a UNIX shell or `CMD.EXE` in Microsoft Windows. The POSIX `popen()` and Windows `_popen()` functions also invoke a command processor but create a pipe between the calling program and the executed command, returning a pointer to a stream that can be used to either read from or write to the pipe [[IEEE Std 1003.1:2013](#)].

Use of the `system()` function can result in exploitable [vulnerabilities](#), in the worst case allowing execution of arbitrary system commands. Situations in which calls to `system()` have high risk include the following:

- When passing an unsanitized or improperly sanitized command string originating from a tainted source
- If a command is specified without a path name and the command processor path name resolution mechanism is accessible to an attacker
- If a relative path to an executable is specified and control over the current working directory is accessible to an attacker
- If the specified executable program can be spoofed by an attacker

Do not invoke a command processor via `system()` or equivalent functions to execute a command.

Noncompliant Code Example

In this noncompliant code example, the `system()` function is used to execute `any_cmd` in the host environment. Invocation of a command processor is not required.

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

enum { BUFSIZE = 512 };

void func(const char *input) {
    char cmdbuf[BUFSIZE];
    int len_wanted = snprintf(cmdbuf, BUFSIZE,
        "any_cmd '%s'", input);
    if (len_wanted >= BUFSIZE) {
        /* Handle error */
    } else if (len_wanted < 0) {
        /* Handle error */
    } else if (system(cmdbuf) == -1) {
        /* Handle error */
    }
}
```

If this code is compiled and run with elevated privileges on a Linux system, for example, an attacker can create an account by entering the following string:

```
'happy'; useradd 'attacker'
```

The shell would interpret this string as two separate commands:

```
any_cmd 'happy';
useradd 'attacker'
```

and create a new user account that the attacker can use to access the compromised system.

This noncompliant code example also violates [STR02-C. Sanitize data passed to complex subsystems](#).

Compliant Solution [POSIX]

In this compliant solution, the call to `system()` is replaced with a call to `execve()`. The `exec` family of functions does not use a full shell interpreter, so it is not vulnerable to command-injection attacks, such as the one illustrated in the noncompliant code example.

The `exec1()`, `execle()`, `execv()`, and `execve()` functions duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a forward slash character (/). As a result, they should be used without a forward slash character (/) only if the `PATH` environment variable is set to a safe value, as described in [ENV03-C. Sanitize the environment when invoking external programs](#).

The `exec1()`, `execle()`, `execv()`, and `execve()` functions do not perform path name substitution.

Additionally, precautions should be taken to ensure the external executable cannot be modified by an untrusted user, for example, by ensuring the executable is not writable by the user.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>

void func(char *input) {
    pid_t pid;
    int status;
    pid_t ret;
    char *const args[3] = {"any_exe", input, NULL};
    char **env;
    extern char **environ;
```

```

/* ... Sanitize arguments ... */

pid = fork();
if (pid == -1) {
    /* Handle error */
} else if (pid != 0) {
    while ((ret = waitpid(pid, &status, 0)) == -1) {
        if (errno != EINTR) {
            /* Handle error */
            break;
        }
    }
    if ((ret != -1) &&
        (!WIFEXITED(status) || !WEXITSTATUS(status)) ) {
        /* Report unexpected child status */
    }
} else {
    /* ... Initialize env as a sanitized copy of environ ... */
    if (execve("/usr/bin/any_cmd", args, env) == -1) {
        /* Handle error */
        _Exit(127);
    }
}
}

```

This compliant solution is significantly different from the preceding noncompliant code example. First, `input` is incorporated into the `args` array and passed as an argument to `execve()`, eliminating concerns about buffer overflow or string truncation while forming the command string. Second, this compliant solution forks a new process before executing `"/usr/bin/any_cmd"` in the child process. Although this method is more complicated than calling `system()`, the added security is worth the additional effort.

The exit status of 127 is the value set by the shell when a command is not found, and POSIX recommends that applications should do the same. XCU, Section 2.8.2, of *Standard for Information Technology-Portable Operating System Interface [POSIX®], Base Specifications, Issue 7* [[IEEE Std 1003.1:2013](#)], says

If a command is not found, the exit status shall be 127. If the command name is found, but it is not an executable utility, the exit status shall be 126. Applications that invoke utilities without using the shell should use these exit status values to report similar errors.

Compliant Solution (Windows)

This compliant solution uses the Microsoft Windows [CreateProcess\(\)](#) API:

```
#include <Windows.h>

void func(TCHAR *input) {
    STARTUPINFO si = { 0 };
    PROCESS_INFORMATION pi;
    si.cb = sizeof(si);
    if (!CreateProcess(TEXT("any_cmd.exe"), input, NULL, NULL, FALSE,
                      0, 0, 0, &si, &pi)) {
        /* Handle error */
    }
    CloseHandle(pi.hThread);
    CloseHandle(pi.hProcess);
}
```

This compliant solution relies on the `input` parameter being non-`const`. If it were `const`, the solution would need to create a copy of the parameter because the `CreateProcess()` function can modify the command-line arguments to be passed into the newly created process.

This solution creates the process such that the child process does not inherit any handles from the parent process, in compliance with [WIN03-C. Understand HANDLE inheritance](#).

Noncompliant Code Example (POSIX)

This noncompliant code invokes the C `system()` function to remove the `.config` file in the user's home directory.

```
#include <stdlib.h>

void func(void) {
    system("rm ~/.config");
}
```

If the vulnerable program has elevated privileges, an attacker can manipulate the value of the `HOME` environment variable such that this program can remove any file named `.config` anywhere on the system.

Compliant Solution (POSIX)

An alternative to invoking the `system()` call to execute an external program to perform a required operation is to implement the functionality directly in the program using existing library calls. This compliant solution calls the POSIX [unlink\(\)](#) function to remove a file without invoking the `system()` function [[IEEE Std 1003.1:2013](#)]

```
#include <pwd.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
void func(void) {
    const char *file_format = "%s/.config";
    size_t len;
    char *pathname;
    struct passwd *pwd;
```

```

/* Get /etc/passwd entry for current user */
pwd = getpwuid(getuid());
if (pwd == NULL) {
    /* Handle error */
}

/* Build full path name home dir from pw entry */

len = strlen(pwd->pw_dir) + strlen(file_format) + 1;
pathname = (char *)malloc(len);
if (NULL == pathname) {
    /* Handle error */
}
int r = snprintf(pathname, len, file_format, pwd->pw_dir);
if (r < 0 || r >= len) {
    /* Handle error */
}
if (unlink(pathname) != 0) {
    /* Handle error */
}

free(pathname);
}

```

The `unlink()` function is not susceptible to a symlink attack where the final component of `pathname` (the file name) is a symbolic link because `unlink()` will remove the symbolic link and not affect any file or directory named by the contents of the symbolic link. (See [FI001-C. Be careful using functions that use file names for identification](#).) While this reduces the susceptibility of the `unlink()` function to symlink attacks, it does not eliminate it. The `unlink()` function is still susceptible if one of the directory names included in the `pathname` is a symbolic link. This could cause the `unlink()` function to delete a similarly named file in a different directory.

Compliant Solution (Windows)

This compliant solution uses the Microsoft Windows [SHGetKnownFolderPath\(\)](#) API to get the current user's My Documents folder, which is then combined with the file name to create the path to the file to be deleted. The file is then removed using the [DeleteFile\(\)](#) API.

```

#include <Windows.h>
#include <ShlObj.h>
#include <Shlwapi.h>

#if defined(_MSC_VER)
    #pragma comment(lib, "Shlwapi")
#endif

void func(void) {
    HRESULT hr;
    LPWSTR path = 0;
    WCHAR full_path[MAX_PATH];

    hr = SHGetKnownFolderPath(&FOLDERID_Documents, 0, NULL, &path);
    if (FAILED(hr)) {
        /* Handle error */
    }
    if (!PathCombineW(full_path, path, L".config")) {
        /* Handle error */
    }
    CoTaskMemFree(path);
    if (!DeleteFileW(full_path)) {
        /* Handle error */
    }
}

```

Exceptions

ENV33-C-EX1: It is permissible to call `system()` with a null pointer argument to determine the presence of a command processor for the system.

Risk Assessments

If the command string passed to `system()`, `popen()`, or other function that invokes a command processor is not fully [sanitized](#), the risk of [exploitation](#) is high. In the worst case scenario, an attacker can execute arbitrary system commands on the compromised machine with the privileges of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ENV33-C	High	Probable	Medium	P12	L1

Related Guidelines

CERT C Secure Coding Standard	ENV03-C. Sanitize the environment when invoking external programs.
SEI CERT C++ Coding Standard	ENV02-CPP. Do not call system() if you do not need a command processor
CERT Oracle Secure Coding Standard for Java	IDS07-J. Sanitize untrusted data passed to the Runtime.exec() method
ISO/IEC TR 24772:2013	Unquoted Search Path or Element [XZQ]
ISO/IEC TS 17961:2013	Calling system [syscall]
MITRE CWE	CWE-78 , Improper Neutralization of Special Elements Used in an OS Command (aka "OS Command Injection") CWE-88 , Argument Injection or Modification

Bibliography

[IEEE Std 1003.1:2013]	XSH, System Interfaces, exec XSH, System Interfaces, popen XSH, System Interfaces, unlink
[Wheeler 2004]	

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/1IAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
blacklist	Dictionary of header globbing to (list of) function name globbing(s) of forbidden functions.	dict(...)
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	12
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
forbidden_libfunc_call	Do not call system().

CertC++-FLP30

Do not use floating-point variables as loop counters.

Input: IR

Source languages: C++

Details

Because floating-point numbers represent real numbers, it is often mistakenly assumed that they can represent any simple fraction exactly. Floating-point numbers are subject to representational limitations just as integers are, and binary floating-point numbers cannot represent all real numbers exactly, even if they can be represented in a small number of decimal digits.

In addition, because floating-point numbers can represent large values, it is often mistakenly assumed that they can represent all significant digits of those values. To gain a large dynamic range, floating-point numbers maintain a fixed number of precision bits (also called the significand) and an exponent, which limit the number of significant digits they can represent.

Different implementations have different precision limitations, and to keep code portable, floating-point variables must not be used as the loop induction variable. See Goldberg's work for an introduction to this topic [[Goldberg 1991](#)].

For the purpose of this rule, a *loop counter* is an induction variable that is used as an operand of a comparison expression that is used as the controlling expression of a `do`, `while`, or `for` loop. An *induction variable* is a variable that gets increased or decreased by a fixed amount on every iteration of a loop [[Aho 1986](#)]. Furthermore, the change to the variable must occur directly in the loop body (rather than inside a function executed within the loop).

Noncompliant Code Example

In this noncompliant code example, a floating-point variable is used as a loop counter. The decimal number 0.1 is a repeating fraction in binary and cannot be

exactly represented as a binary floating-point number. Depending on the implementation, the loop may iterate 9 or 10 times.

```
void func(void) {
    for (float x = 0.1f; x <= 1.0f; x += 0.1f) {
        /* Loop may iterate 9 or 10 times */
    }
}
```

For example, when compiled with GCC or Microsoft Visual Studio 2013 and executed on an x86 processor, the loop is evaluated only nine times.

Compliant Solution

In this compliant solution, the loop counter is an integer from which the floating-point value is derived:

```
#include <stddef.h>

void func(void) {
    for (size_t count = 1; count <= 10; ++count) {
        float x = count / 10.0f;
        /* Loop iterates exactly 10 times */
    }
}
```

Noncompliant Code Example

In this noncompliant code example, a floating-point loop counter is incremented by an amount that is too small to change its value given its precision:

```
void func(void) {
    for (float x = 100000001.0f; x <= 100000010.0f; x += 1.0f) {
        /* Loop may not terminate */
    }
}
```

On many implementations, this produces an infinite loop.

Compliant Solution

In this compliant solution, the loop counter is an integer from which the floating-point value is derived. The variable `x` is assigned a computed value to reduce compounded rounding errors that are present in the noncompliant code example.

```
void func(void) {
    for (size_t count = 1; count <= 10; ++count) {
        float x = 100000000.0f + (count * 1.0f);
        /* Loop iterates exactly 10 times */
    }
}
```

Risk Assessment

The use of floating-point variables as loop counters can result in [unexpected behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FLP30-C	Low	Probable	Low	P6	L2

Related Guidelines

SEI CERT C++ Coding Standard	FLP30-CPP. Do not use floating-point variables as loop counters
CERT Oracle Secure Coding Standard for Java	NUM09-J. Do not use floating-point variables as loop counters
ISO/IEC TR 24772:2013	Floating-Point Arithmetic [PLF]
MISRA C:2012	Directive 1.1 (required) Rule 14.1 (required)

Bibliography

[Aho 1986]	
[Goldberg 1991]	
[Lockheed Martin 05]	AV Rule 197

Excerpt from SEI CERT C++ Coding Standard Wiki [https://www.securecoding.cert.org/confluence/x/AoG_/], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
loop_counter_model		CertLoopCounters
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
float_loop_counter	Use of floating-point loop counter.

CertC++-FLP32

Prevent or detect domain and range errors in math functions.

Input: IR

Source languages: C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Details

The C Standard, 7.12.1 [[ISO/IEC 9899:2011](#)], defines three types of errors that relate specifically to math functions in <math.h>. Paragraph 2 states

A *domain error* occurs if an input argument is outside the domain over which the mathematical function is defined.

Paragraph 3 states

A *pole error* (also known as a singularity or infinitary) occurs if the mathematical function has an exact infinite result as the finite input argument(s) are approached in the limit.

Paragraph 4 states

A *range error* occurs if the mathematical result of the function cannot be represented in an object of the specified type, due to extreme magnitude.

An example of a domain error is the square root of a negative number, such as `sqrt(-1.0)`, which has no meaning in real arithmetic. Contrastingly, 10 raised to the 1-millionth power, `pow(10., 1e6)`, cannot be represented in many floating-point [implementations](#) because of the limited range of the type `double` and consequently constitutes a range error. In both cases, the function will return some value, but the value returned is not the correct result of the computation. An example of a pole error is `log(0.0)`, which results in negative infinity.

Programmers can prevent domain and pole errors by carefully bounds-checking the arguments before calling mathematical functions and taking alternative action if the bounds are violated.

Range errors usually cannot be prevented because they are dependent on the implementation of floating-point numbers as well as on the function being applied. Instead of preventing range errors, programmers should attempt to detect them and take alternative action if a range error occurs.

The following table lists the `double` forms of standard mathematical functions, along with checks that should be performed to ensure a proper input domain, and indicates whether they can also result in range or pole errors, as reported by the C Standard. Both `float` and `long double` forms of these functions also exist but are omitted from the table for brevity. If a function has a specific domain over which it is defined, the programmer must check its input values. The programmer must also check for range errors where they might occur. The standard math functions not listed in this table, such as `fabs()`, have no domain restrictions and cannot result in range or pole errors.

Function	Domain	Range	Pole
acos(x)	-1 <= x && x <= 1	No	No
asin(x)	-1 <= x && x <= 1	Yes	No
atan(x)	None	Yes	No
atan2(y, x)	x != 0 && y != 0	No	No
acosh(x)	x >= 1	Yes	No
asinh(x)	None	Yes	No
atanh(x)	-1 < x && x < 1	Yes	Yes
cosh(x), sinh(x)	None	Yes	No
exp(x), exp2(x), expm1(x)	None	Yes	No
ldexp(x, exp)	None	Yes	No
log(x), log10(x), log2(x)	x >= 0	No	Yes
log1p(x)	x >= -1	No	Yes
ilogb(x)	x != 0 && !isinf(x) && !isnan(x)	Yes	No
logb(x)	x != 0	Yes	Yes
scalbn(x, n), scalbln(x, n)	None	Yes	No
hypot(x, y)	None	Yes	No
pow(x,y)	x > 0 (x == 0 && y > 0) (x < 0 && y is an integer)	Yes	Yes
sqrt(x)	x >= 0	No	No
erf(x)	None	Yes	No
erfc(x)	None	Yes	No
lgamma(x), tgamma(x)	x != 0 && [x < 0 && x is an integer]	Yes	Yes
lrint(x), lround(x)	None	Yes	No
fmod(x, y), remainder(x, y), remquo(x, y, quo)	y != 0	Yes	No
nextafter(x, y), nexttoward(x, y)	None	Yes	No
fdim(x,y)	None	Yes	No
fma(x,y,z)	None	Yes	No

Domain and Pole Checking

The most reliable way to handle domain and pole errors is to prevent them by checking arguments beforehand, as in the following exemplar:

```
double safe_sqrt(double x) {
    if (x < 0) {
        fprintf(stderr, "sqrt requires a nonnegative argument");
        /* Handle domain / pole error */
    }
    return sqrt (x);
}
```

Range Checking

Programmers usually cannot prevent range errors, so the most reliable way to handle them is to detect when they have occurred and act accordingly.

The exact treatment of error conditions from math functions is tedious. The C Standard, 7.12.1 [\[ISO/IEC 9899:2011\]](#), defines the following behavior for floating-point overflow:

A floating result overflows if the magnitude of the mathematical result is finite but so large that the mathematical result cannot be represented without extraordinary roundoff error in an object of the specified type. If a floating result overflows and default rounding is in effect, then the

function returns the value of the macro `HUGE_VAL`, `HUGE_VALF`, or `HUGE_VALL` according to the return type, with the same sign as the correct value of the function; if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, the integer expression `errno` acquires the value `ERANGE`; if the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero, the "overflow" floating-point exception is raised.

It is preferable not to check for errors by comparing the returned value against `HUGE_VAL` or `0` for several reasons:

- These are, in general, valid (albeit unlikely) data values.
- Making such tests requires detailed knowledge of the various error returns for each math function.
- Multiple results aside from `HUGE_VAL` and `0` are possible, and programmers must know which are possible in each case.
- Different versions of the library have varied in their error-return behavior.

It can be unreliable to check for math errors using `errno` because an [implementation](#) might not set `errno`. For real functions, the programmer determines if the implementation sets `errno` by checking whether `math_errhandling & MATH_ERRNO` is nonzero. For complex functions, the C Standard, 7.3.2, paragraph 1, simply states that "an implementation may set `errno` but is not required to" [\[ISO/IEC 9899:2011\]](#).

The obsolete *System V Interface Definition* (SVID3) [\[UNIX 1992\]](#) provides more control over the treatment of errors in the math library. The programmer can define a function named `matherr()` that is invoked if errors occur in a math function. This function can print diagnostics, terminate the execution, or specify the desired return value. The `matherr()` function has not been adopted by C or POSIX, so it is not generally portable.

The following error-handling template uses C Standard functions for floating-point errors when the C macro `math_errhandling` is defined and indicates that they should be used; otherwise, it examines `errno`:

```
#include <math.h>
#include <fenv.h>
#include <errno.h>

/* ... */
/* Use to call a math function and check errors */
{
    #pragma STDC FENV_ACCESS ON

    if ((math_errhandling & MATH_ERREXCEPT) {
        feclearexcept(FE_ALL_EXCEPT);
    }
    errno = 0;

    /* Call the math function */

    if (((math_errhandling & MATH_ERRNO) && errno != 0) {
        /* Handle range error */
    } else if (((math_errhandling & MATH_ERREXCEPT) &&
               fetestexcept(FE_INVALID | FE_DIVBYZERO |
                           FE_OVERFLOW | FE_UNDERFLOW) != 0) {
        /* Handle range error */
    }
}
```

See [FLP03-C. Detect and handle floating-point errors](#) for more details on how to detect floating-point errors.

Subnormal Numbers

A subnormal number is a nonzero number that does not use all of its precision bits [\[IEEE 754 2006\]](#). These numbers can be used to represent values that are closer to 0 than the smallest normal number (one that uses all of its precision bits). However, the `asin()`, `asinh()`, `atan()`, `atanh()`, and `erf()` functions may produce range errors, specifically when passed a subnormal number. When evaluated with a subnormal number, these functions can produce an inexact, subnormal value, which is an underflow error. The C Standard, 7.12.1, paragraph 6 [\[ISO/IEC 9899:2011\]](#), defines the following behavior for floating-point underflow:

The result underflows if the magnitude of the mathematical result is so small that the mathematical result cannot be represented, without extraordinary roundoff error, in an object of the specified type. If the result underflows, the function returns an implementation-defined value whose magnitude is no greater than the smallest normalized positive number in the specified type; if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, whether `errno` acquires the value `ERANGE` is implementation-defined; if the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero, whether the "underflow" floating-point exception is raised is implementation-defined.

Implementations that support floating-point arithmetic but do not support subnormal numbers, such as IBM S/360 hex floating-point or nonconforming IEEE-754 implementations that skip subnormals (or support them by flushing them to zero), can return a range error when calling one of the following families of functions with the following arguments:

- `fmod((min+subnorm), min)`
- `remainder((min+subnorm), min)`
- `remquo((min+subnorm), min, quo)`

where `min` is the minimum value for the corresponding floating point type and `subnorm` is a subnormal value.

If Annex F is supported and subnormal results are supported, the returned value is exact and a range error cannot occur. The C Standard, F.10.7.1 [\[ISO/IEC 9899:2011\]](#), specifies the following for the `fmod()`, `remainder()`, and `remquo()` functions:

When subnormal results are supported, the returned value is exact and is independent of the current rounding direction mode.

Annex F, subclause F.10.7.2, paragraph 2, and subclause F.10.7.3, paragraph 2, of the C Standard identify when subnormal results are supported.

Noncompliant Code Example (`sqrt()`)

This noncompliant code example determines the square root of `x`:

```
#include <math.h>
void func(double x) {
```

```
    double result;
    result = sqrt(x);
}
```

However, this code may produce a domain error if x is negative.

Compliant Solution (`sqrt()`)

Because this function has domain errors but no range errors, bounds checking can be used to prevent domain errors:

```
#include <math.h>

void func(double x) {
    double result;

    if (isless(x, 0.0)) {
        /* Handle domain error */
    }

    result = sqrt(x);
}
```

Noncompliant Code Example (`sinh()`, Range Errors)

This noncompliant code example determines the hyperbolic sine of x :

```
#include <math.h>

void func(double x) {
    double result;
    result = sinh(x);
}
```

This code may produce a range error if x has a very large magnitude.

Compliant Solution (`sinh()`, Range Errors)

Because this function has no domain errors but may have range errors, the programmer must detect a range error and act accordingly:

```
#include <math.h>
#include <fenv.h>
#include <errno.h>

void func(double x) {
    double result;
    {

        #pragma STDC FENV_ACCESS ON
        if (math_errhandling & MATH_ERREXCEPT) {
            feclearexcept(FE_ALL_EXCEPT);
        }
        errno = 0;

        result = sinh(x);

        if ((math_errhandling & MATH_ERRNO) && errno != 0) {
            /* Handle range error */
        } else if ((math_errhandling & MATH_ERREXCEPT) &&
                   fetestexcept(FE_INVALID | FE_DIVBYZERO |
                               FE_OVERFLOW | FE_UNDERFLOW) != 0) {
            /* Handle range error */
        }
    }

    /* Use result... */
}
```

Noncompliant Code Example (`pow()`)

This noncompliant code example raises x to the power of y :

```
#include <math.h>

void func(double x, double y) {
    double result;
    result = pow(x, y);
}
```

This code may produce a domain error if x is negative and y is not an integer value or if x is 0 and y is 0. A domain error or pole error may occur if x is 0 and y is negative, and a range error may occur if the result cannot be represented as a `double`.

Compliant Solution (`pow()`)

Because the `pow()` function can produce domain errors, pole errors, and range errors, the programmer must first check that x and y lie within the proper domain and do not generate a pole error and then detect whether a range error occurs and act accordingly:

```
#include <math.h>
#include <fenv.h>
```

```

#include <errno.h>

void func(double x, double y) {
    double result;

    if (((x == 0.0f) && islesseq(y, 0.0)) || isless(x, 0.0)) {
        /* Handle domain or pole error */
    }

    {
        #pragma STDC FENV_ACCESS ON
        if (math_errhandling & MATH_ERREXCEPT) {
            feclearexcept(FE_ALL_EXCEPT);
        }
        errno = 0;

        result = pow(x, y);

        if ((math_errhandling & MATH_ERRNO) && errno != 0) {
            /* Handle range error */
        } else if ((math_errhandling & MATH_ERREXCEPT) &&
                   fetestexcept(FE_INVALID | FE_DIVBYZERO |
                               FE_OVERFLOW | FE_UNDERFLOW) != 0) {
            /* Handle range error */
        }
    }

    /* Use result... */
}

```

Noncompliant Code Example {asin(), Subnormal Number}

This noncompliant code example determines the inverse sine of x:

```

#include <math.h>

void func(float x) {
    float result = asin(x);
    /* ... */
}

```

Compliant Solution {asin(), Subnormal Number}

Because this function has no domain errors but may have range errors, the programmer must detect a range error and act accordingly:

```

#include <math.h>
#include <fenv.h>
#include <errno.h>
void func(float x) {
    float result;

    {
        #pragma STDC FENV_ACCESS ON
        if (math_errhandling & MATH_ERREXCEPT) {
            feclearexcept(FE_ALL_EXCEPT);
        }
        errno = 0;

        result = asin(x);

        if ((math_errhandling & MATH_ERRNO) && errno != 0) {
            /* Handle range error */
        } else if ((math_errhandling & MATH_ERREXCEPT) &&
                   fetestexcept(FE_INVALID | FE_DIVBYZERO |
                               FE_OVERFLOW | FE_UNDERFLOW) != 0) {
            /* Handle range error */
        }
    }

    /* Use result... */
}

```

Risk Assessment

Failure to prevent or detect domain and range errors in math functions may cause unexpected results.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FLP32-C	Medium	Probable	Medium	P8	L2

Related Guidelines

CERT C Secure Coding Standard	FLP03-C. Detect and handle floating-point errors
MITRE CWE	CWE-682, Incorrect Calculation

Bibliography

[ISO/IEC 9899:2011]	7.3.2, "Conventions" 7.12.1, "Treatment of Error Conditions" F.10.7, "Remainder Functions"
[IEEE 754 2006]	
[Plum 1985]	Rule 2-2
[Plum 1989]	Topic 2.10, "conv-Conversions and Overflow"
[UNIX 1992]	<i>System V Interface Definition</i> (SVID3)

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/rgQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
math_functions	Functions which require a proper input domain check.	dict{...}
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
domain_check_expected	Prevent or detect domain and range errors in math functions.
literal_outside_domain	Literal used is outside of math function's domain.

CertC++-FLP37

Do not use object representations to compare floating-point values.

Input: IR

Source languages: C++

Details

The object representation for floating-point values is implementation defined. However, an implementation that defines the `__STDC_IEC_559__` macro shall conform to the IEC 60559 floating-point standard and uses what is frequently referred to as IEEE 754 floating-point arithmetic [[ISO/IEC 9899:2011](#)]. The floating-point object representation used by IEC 60559 is one of the most common floating-point object representations in use today.

All floating-point object representations use specific bit patterns to encode the value of the floating-point number being represented. However, equivalence of floating-point values is not encoded solely by the bit pattern used to represent the value. For instance, if the floating-point format supports negative zero values (as IEC 60559 does), the values `-0.0` and `0.0` are equivalent and will compare as equal, but the bit patterns used in the object representation are not identical. Similarly, if two floating-point values are both (the same) NaN, they will not compare as equal, despite the bit patterns being identical, because they are not equivalent.

Do not compare floating-point object representations directly, such as by calling `memcmp()` or its moral equivalents. Instead, the equality operators (`==` and `!=`) should be used to determine if two floating-point values are equivalent.

Noncompliant Code Example

In this noncompliant code example, `memcmp()` is used to compare two structures for equality. However, since the structure contains a floating-point object, this code may not behave as the programmer intended.

```
#include <stdbool.h>
#include <string.h>

struct S {
    int i;
    float f;
};

bool are_equal(const struct S *s1, const struct S *s2) {
    if (!s1 && !s2)
        return true;
    else if (!s1 || !s2)
        return false;
    return 0 == memcmp(s1, s2, sizeof(struct S));
}
```

}

Compliant Solution

In this compliant solution, the structure members are compared individually:

```
#include <stdbool.h>
#include <string.h>

struct S {
    int i;
    float f;
};

bool are_equal(const struct S *s1, const struct S *s2) {
    if (!s1 && !s2)
        return true;
    else if (!s1 || !s2)
        return false;
    return s1->i == s2->i &&
           s1->f == s2->f;
}
```

Risk Assessment

Using the object representation of a floating-point value for comparisons can lead to incorrect equality results, which can lead to unexpected behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FLP37-C	Low	Unlikely	Medium	P2	L3

Bibliography

[ISO/IEC 9899:2011] Annex F, "IEC 60559 floating-point arithmetic"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/J4DkC>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
allow_char	Whether to allow memcmp on 'char' type.	True
allow_composites_without_padding	Whether to allow using memcmp on structs and unions that have no padding bytes.	True
allow_float	Whether to allow memcmp on floating point types.	False
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
disallowed_memcmp_pointer_arg	Disallowed type of pointer argument. (disabled)
memcmp_char_pointer_arg	memcmp shall not be used with char pointer argument, use strncmp instead.
memcmp_float	memcmp shall not be used to compare floats as the same value may be stored using different representations.
memcmp_padding	memcmp shall not be used to compare structs with padding. (disabled)
memcmp_struct_pointer_arg	memcmp shall not be used with struct pointer argument as it would compare padding as well. (disabled)
memcmp_union_pointer_arg	memcmp shall not be used with union pointer argument as it would compare padding and different kinds of representation. (disabled)

CertC++-MSC30

Do not use the `rand()` function for generating pseudorandom numbers.

Input: IR

Source languages: C++

Details

Pseudorandom number generators use mathematical algorithms to produce a sequence of numbers with good statistical properties, but the numbers produced are not genuinely random.

The C Standard `rand()` function makes no guarantees as to the quality of the random sequence produced. The numbers generated by some implementations of `rand()` have a comparatively short cycle and the numbers can be predictable. Applications that have strong pseudorandom number requirements must use a generator that is known to be sufficient for their needs.

Noncompliant Code Example

The following noncompliant code generates an ID with a numeric part produced by calling the `rand()` function. The IDs produced are predictable and have limited randomness.

```
#include <stdio.h>
#include <stdlib.h>

enum { len = 12 };

void func(void) {
    /*
     * id will hold the ID, starting with the characters
     * "ID" followed by a random integer.
     */
    char id[len];
    int r;
    int num;
    /* ... */
    r = rand(); /* Generate a random integer */
    num = sprintf(id, len, "ID%-d", r); /* Generate the ID */
    /* ... */
}
```

Compliant Solution (POSIX)

This compliant solution replaces the `rand()` function with the POSIX `random()` function:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

enum { len = 12 };

void func(void) {
    /*
     * id will hold the ID, starting with the characters
     * "ID" followed by a random integer.
     */
    char id[len];
    int r;
    int num;
    /* ... */
    struct timespec ts;
    if (timespec_get(&ts, TIME_UTC) == 0) {
        /* Handle error */
    }
    srand(ts.tv_nsec ^ ts.tv_sec); /* Seed the PRNG */
    /* ... */
    r = random(); /* Generate a random integer */
    num = sprintf(id, len, "ID%-d", r); /* Generate the ID */
    /* ... */
}
```

The POSIX `random()` function is a better pseudorandom number generator. Although on some platforms the low dozen bits generated by `rand()` go through a cyclic pattern, all the bits generated by `random()` are usable. The `rand48` family of functions provides another alternative for pseudorandom numbers.

Although not specified by POSIX, [arc4random\(\)](#) is another possibility for systems that support it. The `arc4random(3)` manual page [OpenBSD] states

... provides higher quality of data than those described in `rand(3)`, `random(3)`, and `drand48(3)`.

To achieve the best random numbers possible, an [implementation](#)-specific function must be used. When unpredictability is crucial and speed is not an issue, as in the creation of strong cryptographic keys, use a true entropy source, such as `/dev/random`, or a hardware device capable of generating random numbers. The `/dev/random` device can block for a long time if there are not enough events going on to generate sufficient entropy.

Compliant Solution (Windows)

On Windows platforms, the [CryptGenRandom\(\)](#) function can be used to generate cryptographically strong random numbers. The exact details of the implementation are unknown, including, for example, what source of entropy `CryptGenRandom()` uses. The Microsoft Developer Network `CryptGenRandom()` reference [MSDN] states

If an application has access to a good random source, it can fill the `pbBuffer` buffer with some random data before calling `CryptGenRandom()`. The

CSP [cryptographic service provider] then uses this data to further randomize its internal seed. It is acceptable to omit the step of initializing the pbBuffer buffer before calling `CryptGenRandom()`.

```
#include <Windows.h>
#include <wincrypt.h>
#include <stdio.h>

void func(void) {
    HCRYPTPROV prov;
    if (CryptAcquireContext(&prov, NULL, NULL,
                           PROV_RSA_FULL, 0)) {
        long int li = 0;
        if (CryptGenRandom(prov, sizeof(li), (BYTE *)&li)) {
            printf("Random number: %ld\n", li);
        } else {
            /* Handle error */
        }
        if (!CryptReleaseContext(prov, 0)) {
            /* Handle error */
        }
    } else {
        /* Handle error */
    }
}
```

Risk Assessment

The use of the `rand()` function can result in predictable random numbers.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC30-C	Medium	Unlikely	Low	P6	L2

Related Guidelines

SEI CERT C++ Coding Standard	MSC50-CPP. Do not use std::rand() for generating pseudorandom numbers
CERT Oracle Secure Coding Standard for Java	MSC02-J. Generate strong random numbers
MITRE CWE	CWE-327 , Use of a Broken or Risky Cryptographic Algorithm CWE-330 , Use of Insufficiently Random Values CWE-331 , Insufficient Entropy CWE-338 , Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)

Bibliography

[MSDN]	"CryptGenRandom Function"
[OpenBSD]	arc4random()

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/qw4>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
blacklist	Dictionary of header globbing to [list of] function name globbing(s) of forbidden functions.	dict(...)
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
forbidden_libfunc_call	Do not use the <code>rand()</code> function for generating pseudorandom numbers.

CertC++-MSC32

Properly seed pseudorandom number generators.

Input: IR

Source languages: C++

Details

A pseudorandom number generator (PRNG) is a deterministic algorithm capable of generating sequences of numbers that approximate the properties of random numbers. Each sequence is completely determined by the initial state of the PRNG and the algorithm for changing the state. Most PRNGs make it possible to set the initial state, also called the *seed state*. Setting the initial state is called *seeding* the PRNG.

Calling a PRNG in the same initial state, either without seeding it explicitly or by seeding it with the same value, results in generating the same sequence of random numbers in different runs of the program. Consider a PRNG function that is seeded with some initial seed value and is consecutively called to produce a sequence of random numbers, *s*. If the PRNG is subsequently seeded with the same initial seed value, then it will generate the same sequence *s*.

As a result, after the first run of an improperly seeded PRNG, an attacker can predict the sequence of random numbers that will be generated in the future runs. Improperly seeding or failing to seed the PRNG can lead to [vulnerabilities](#), especially in security protocols.

The solution is to ensure that the PRNG is always properly seeded. A properly seeded PRNG will generate a different sequence of random numbers each time it is run.

Not all random number generators can be seeded. True random number generators that rely on hardware to produce completely unpredictable results do not need to be and cannot be seeded. Some high-quality PRNGs, such as the `/dev/random` device on some UNIX systems, also cannot be seeded. This rule applies only to algorithmic pseudorandom number generators that can be seeded.

Noncompliant Code Example (POSIX)

This noncompliant code example generates a sequence of 10 pseudorandom numbers using the `random()` function. When `random()` is not seeded, it behaves like `rand()`, producing the same sequence of random numbers each time any program that uses it is run.

```
#include <stdio.h>
#include <stdlib.h>

void func(void) {
    for (unsigned int i = 0; i < 10; ++i) {
        /* Always generates the same sequence */
        printf("%ld, ", random());
    }
}
```

The output is as follows:

```
1st run: 1804289383, 846930886, 1681692777, 1714636915, 1957747793, 424238335, 719885386, 1649760492, 596516649,
         1189641421,
2nd run: 1804289383, 846930886, 1681692777, 1714636915, 1957747793, 424238335, 719885386, 1649760492, 596516649,
         1189641421,
...
nth run: 1804289383, 846930886, 1681692777, 1714636915, 1957747793, 424238335, 719885386, 1649760492, 596516649,
         1189641421,
```

Compliant Solution (POSIX)

Call `random()` before invoking `random()` to seed the random sequence generated by `random()`. This compliant solution produces different random number sequences each time the function is called, depending on the resolution of the system clock:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void func(void) {
    struct timespec ts;
    if (timespec_get(&ts, TIME_UTC) == 0) {
        /* Handle error */
    } else {
        srand(ts.tv_nsec ^ ts.tv_sec);
        for (unsigned int i = 0; i < 10; ++i) {
            /* Generates different sequences at different runs */
            printf("%ld, ", random());
        }
    }
}
```

The output is as follows:

```
1st run: 198682410, 2076262355, 910374899, 428635843, 2084827500, 1558698420, 4459146, 733695321, 2044378618, 1649046624,
2nd run: 1127071427, 252907983, 1358798372, 2101446505, 1514711759, 229790273, 954268511, 1116446419, 368192457,
         12975948050,
3rd run: 2052868434, 1645663878, 731874735, 1624006793, 938447420, 1046134947, 1901136083, 418123888, 836428296,
         2017467418,
```

This may not be sufficiently random for concurrent execution, which may lead to correlated generated series in different threads. Depending on the application and the desired level of security, a programmer may choose alternative ways to seed PRNGs. In general, hardware is more capable than software of generating real random numbers (for example, by sampling the thermal noise of a diode).

Compliant Solution (Windows)

The [`CryptGenRandom\(\)`](#) function does not run the risk of not being properly seeded because its arguments serve as seeders:

```

#include <Windows.h>
#include <wincrypt.h>
#include <stdio.h>

void func(void) {
    HCRYPTPROV hCryptProv;
    long rand_buf;
    /* Example of instantiating the CSP */
    if (CryptAcquireContext(&hCryptProv, NULL, NULL,
                           PROV_RSA_FULL, 0)) {
        printf("CryptAcquireContext succeeded.\n");
    } else {
        printf("Error during CryptAcquireContext!\n");
    }

    for (unsigned int i = 0; i < 10; ++i) {
        if (!CryptGenRandom(hCryptProv, sizeof(rand_buf),
                            (BYTE *)&rand_buf)) {
            printf("Error\n");
        } else {
            printf("%ld, ", rand_buf);
        }
    }
}

```

The output is as follows:

```

1st run: -1597837311, 906130682, -1308031886, 1048837407, -931041900, -658114613, -1709220953, -1019697289, 1802206541,
406505841,
2nd run: 885904119, -687379556, -1782296854, 1443701916, -624291047, 2049692692, -990451563, -142307804, 1257079211,
897185104,
3rd run: 190598304, -1537409464, 1594174739, -424401916, -1975153474, 826912927, 1705549595, -1515331215, 474951399,
1982500583,

```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC32-C	Medium	Likely	Low	P18	L1

Related Guidelines

CERT C Secure Coding Standard	MSC30-C. Do not use the rand() function for generating pseudorandom numbers
SEI CERT C++ Coding Standard	MSC51-CPP. Ensure your random number generator is properly seeded
MITRE CWE	CWE-327 , Use of a Broken or Risky Cryptographic Algorithm CWE-330 , Use of Insufficiently Random Values CWE-331 , Insufficient Entropy CWE-338 , Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)

Bibliography

[MSDN]	"CryptGenRandom Function"
------------------------	---

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/hABhAQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
seed_properly	Properly seed pseudorandom number generators.

Do not pass invalid data to the `asctime()` function.

Input: IR

Source languages: C++

Details

The C Standard, 7.27.3.1 [ISO/IEC 9899:2011], provides the following sample implementation of the `asctime()` function:

```
char *asctime(const struct tm *timeptr) {
    static const char wday_name[7][3] = {
        "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
    };
    static const char mon_name[12][3] = {
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
    };
    static char result[26];
    sprintf(
        result,
        "%3s %3d %2d:%2d:%2d %d\n",
        wday_name[timeptr->tm_wday],
        mon_name[timeptr->tm_mon],
        timeptr->tm_mday, timeptr->tm_hour,
        timeptr->tm_min, timeptr->tm_sec,
        1900 + timeptr->tm_year
    );
    return result;
}
```

This function is supposed to output a character string of 26 characters at most, including the terminating null character. If we count the length indicated by the format directives, we arrive at 25. Taking into account the terminating null character, the array size of the string appears sufficient.

However, this [implementation](#) assumes that the values of the `struct tm` data are within normal ranges and does nothing to enforce the range limit. If any of the values print more characters than expected, the `sprintf()` function may overflow the `result` array. For example, if `tm_year` has the value 12345, then 27 characters (including the terminating null character) are printed, resulting in a buffer overflow.

The *POSIX® Base Specifications* [IEEE Std 1003.1:2013] says the following about the `asctime()` and `asctime_r()` functions:

These functions are included only for compatibility with older implementations. They have undefined behavior if the resulting string would be too long, so the use of these functions should be discouraged. On implementations that do not detect output string length overflow, it is possible to overflow the output buffers in such a way as to cause applications to fail, or possible system security violations. Also, these functions do not support localized date and time formats. To avoid these problems, applications should use `strftime()` to generate strings from broken-down times.

The C Standard, Annex K, also defines `asctime_s()`, which can be used as a secure substitute for `asctime()`.

The `asctime()` function appears in the list of obsolescent functions in [MSC24-C. Do not use deprecated or obsolescent functions](#).

Noncompliant Code Example

This noncompliant code example invokes the `asctime()` function with potentially unsanitized data:

```
#include <time.h>

void func(struct tm *time_tm) {
    char *time = asctime(time_tm);
    /* ... */
}
```

Compliant Solution (`strftime()`)

The `strftime()` function allows the programmer to specify a more rigorous format and also to specify the maximum size of the resulting time string:

```
#include <time.h>

enum { maxsize = 26 };

void func(struct tm *time) {
    char s[maxsize];
    /* Current time representation for locale */
    const char *format = "%c";
    size_t size = strftime(s, maxsize, format, time);
}
```

This call has the same effects as `asctime()` but also ensures that no more than `maxsize` characters are printed, preventing buffer overflow.

Compliant Solution (`asctime_s()`)

The C Standard, Annex K, defines the `asctime_s()` function, which serves as a close replacement for the `asctime()` function but requires an additional argument that specifies the maximum size of the resulting time string:

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <time.h>

enum { maxsize = 26 };
```

```

void func(struct tm *time_tm) {
    char buffer[maxsize];
    if (asctime_s(buffer, maxsize, &time_tm)) {
        /* Handle error */
    }
}

```

Risk Assessment

On [implementations](#) that do not detect output-string-length overflow, it is possible to overflow the output buffers.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC33-C	High	Likely	Low	P27	L1

Related Guidelines

CERT C Secure Coding Standard	MSC24-C. Do not use deprecated or obsolescent functions
---	---

Bibliography

[IEEE Std 1003.1:2013]	XSH, System Interfaces, asctime
[ISO/IEC 9899:2011]	7.27.3.1, "The asctime Function"

Excerpt from SEI CERT C++ Coding Standard Wiki <https://www.securecoding.cert.org/confluence/x/CgCuAQ>, Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
blacklist	Dictionary of header globbing to (list of) function name globbing(s) of forbidden functions.	dict(...)
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	27
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
forbidden_libfunc_call	Potentially unsanitized data passed to asctime. Use asctime_s or strftime instead.

CertC++-MSC37

Ensure that control never reaches the end of a non-void function.

Input: IR

Source languages: C++

Details

If control reaches the closing curly brace {} of a non-void function without evaluating a `return` statement, using the return value of the function call is [undefined behavior](#). (See [undefined behavior 88](#).)

Noncompliant Code Example

In this noncompliant code example, control reaches the end of the `checkpass()` function when the two strings passed to `strcmp()` are not equal, resulting in undefined behavior. Many compilers will generate code for the `checkpass()` function, returning various values along the execution path where no `return` statement is defined.

```

#include <string.h>
#include <stdio.h>

int checkpass(const char *password) {
    if (strcmp(password, "pass") == 0) {
        return 1;
    }
}

```

```

}

void func(const char *userinput) {
    if (checkpass(userinput)) {
        printf("Success\n");
    }
}

```

This error is frequently diagnosed by compilers. (See [MSC00-C. Compile cleanly at high warning levels](#).)

Compliant Solution

This compliant solution ensures that the `checkpass()` function always returns a value:

```

#include <string.h>
#include <stdio.h>

int checkpass(const char *password) {
    if (strcmp(password, "pass") == 0) {
        return 1;
    }
    return 0;
}

void func(const char *userinput) {
    if (checkpass(userinput)) {
        printf("Success!\n");
    }
}

```

Noncompliant Code Example

In this noncompliant code example, control reaches the end of the `getlen()` function when `input` does not contain the integer `delim`. Because the potentially undefined return value of `getlen()` is later used as an index into an array, a buffer overflow may occur.

```

#include <stddef.h>

size_t getlen(const int *input, size_t maxlen, int delim) {
    for (size_t i = 0; i < maxlen; ++i) {
        if (input[i] == delim) {
            return i;
        }
    }
}

void func(int userdata) {
    size_t i;
    int data[] = { 1, 1, 1 };
    i = getlen(data, sizeof(data), 0);
    data[i] = userdata;
}

```

Implementation Details (GCC)

Violating this rule can have unexpected consequences, as in the following example:

```

#include <stdio.h>

size_t getlen(const int *input, size_t maxlen, int delim) {
    for (size_t i = 0; i < maxlen; ++i) {
        if (input[i] == delim) {
            return i;
        }
    }
}

int main(int argc, char **argv) {
    size_t i;
    int data[] = { 1, 1, 1 };

    i = getlen(data, sizeof(data), 0);
    printf("Returned: %zu\n", i);
    data[i] = 0;

    return 0;
}

```

When this program is compiled with `-Wall` on most versions of the GCC compiler, the following warning is generated:

```

example.c: In function 'getlen':
example.c:12: warning: control reaches end of non-void function

```

None of the inputs to the function equal the delimiter, so when run with GCC 5.3 on Linux, control reaches the end of the `getlen()` function, which is undefined behavior and in this test returns 3, causing an out-of-bounds write to the `data` array.

Compliant Solution

This compliant solution changes the interface of `getlen()` to store the result in a user-provided pointer and returns a status indicator to report success or failure. The best method for handling this type of error is specific to the application and the type of error. (See [ERR00-C. Adopt and implement a consistent](#)

[and comprehensive error-handling policy](#) for more on error handling.)

```
#include <stddef.h>

int getlen(const int *input, size_t maxlen, int delim,
           size_t *result) {
    if (result == NULL) {
        return -1;
    }
    for (size_t i = 0; i < maxlen; ++i) {
        if (input[i] == delim) {
            *result = i;
            return 0;
        }
    }
    return -1;
}

void func(int userdata) {
    size_t i;
    int data[] = {1, 1, 1};
    if (getlen(data, sizeof(data), 0, &i) != 0) {
        /* Handle error */
    } else {
        data[i] = userdata;
    }
}
```

Exceptions

MSC37-C-EX1: According to the C Standard, 5.1.2.2.3, paragraph 1 [[ISO/IEC 9899:2011](#)], "Reaching the } that terminates the main function returns a value of 0." As a result, it is permissible for control to reach the end of the `main()` function without executing a `return` statement.

MSC37-C-EX2: It is permissible for a control path to not return a value if that code path is never taken and a function marked `_Noreturn` is called as part of that code path. For example:

```
#include <stdio.h>
#include <stdlib.h>

_Noreturn void unreachable(const char *msg) {
    printf("Unreachable code reached: %s\n", msg);
    exit(1);
}

enum E {
    One,
    Two,
    Three
};

int f(enum E e) {
    switch (e) {
        case One: return 1;
        case Two: return 2;
        case Three: return 3;
    }
    unreachable("Can never get here");
}
```

Risk Assessment

Using the return value from a non-void function where control reaches the end of the function without evaluating a `return` statement can lead to buffer overflow [vulnerabilities](#) as well as other [unexpected program behaviors](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC37-C	High	Unlikely	Low	P9	L2

Related Guidelines

CERT C Secure Coding Standard	MSC01-C. Strive for logical completeness
---	--

Bibliography

[ISO/IEC 9899:2011]	5.1.2.2.3, "Program Termination"
-------------------------------------	----------------------------------

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/goCGAg>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	9
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
lambda_return_missing_value	Return without value in non-void lambda expression
missing_return	Ensure that control never reaches the end of a non-void function.
missing_return_in_lambda	Non-void lambda expression needs return with value at end.
return_missing_value	Return without value in non-void function

CertC++-MSC50

Do not use `std::rand()` for generating pseudorandom numbers.

Input: IR

Source languages: C++

Details

Pseudorandom number generators use mathematical algorithms to produce a sequence of numbers with good statistical properties, but the numbers produced are not genuinely random.

The C Standard `rand()` function, exposed through the C++ standard library through `<cstdlib>` as `std::rand()`, makes no guarantees as to the quality of the random sequence produced. The numbers generated by some implementations of `std::rand()` have a comparatively short cycle, and the numbers can be predictable. Applications that have strong pseudorandom number requirements must use a generator that is known to be sufficient for their needs.

Noncompliant Code Example

The following noncompliant code generates an ID with a numeric part produced by calling the `rand()` function. The IDs produced are predictable and have limited randomness. Further, depending on the value of `RAND_MAX`, the resulting value can have modulo bias.

```
#include <cstdlib>
#include <string>

void f() {
    std::string id("ID"); // Holds the ID, starting with the characters "ID" followed
                          // by a random integer in the range [0-10000].
    id += std::to_string(std::rand() % 10000);
    // ...
}
```

Compliant Solution

The C++ standard library provides mechanisms for fine-grained control over pseudorandom number generation. It breaks random number generation into two parts: one is the algorithm responsible for providing random values (the engine), and the other is responsible for distribution of the random values via a density function (the distribution). The distribution object is not strictly required, but it works to ensure that values are properly distributed within a given range instead of improperly distributed due to bias issues. This compliant solution uses the [Mersenne Twister](#) algorithm as the engine for generating random values and a uniform distribution to negate the modulo bias from the noncompliant code example.

```
#include <random>
#include <string>

void f() {
    std::string id("ID"); // Holds the ID, starting with the characters "ID" followed
                          // by a random integer in the range [0-10000].
    std::uniform_int_distribution<int> distribution(0, 10000);
    std::random_device rd;
    std::mt19937 engine(rd());
    id += std::to_string(distribution(engine));
    // ...
}
```

This compliant solution also seeds the random number engine, in conformance with [MSC51-CPP. Ensure your random number generator is properly seeded](#).

Risk Assessment

Using the `std::rand()` function could lead to predictable random numbers.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC50-CPP	Medium	Unlikely	Low	P6	L2

Related Guidelines

SEI CERT C++ Coding Standard	MSC51-CPP. Ensure your random number generator is properly seeded
SEI CERT C Coding Standard	MSC30-C. Do not use the rand() function for generating pseudorandom numbers
CERT Oracle Secure Coding Standard for Java	MSC02-J. Generate strong random numbers
MITRE CWE	CWE-327, Use of a Broken or Risky Cryptographic Algorithm CWE-330, Use of Insufficiently Random Values

Bibliography

[ISO/IEC 9899:2011]	Subclause 7.22.2, "Pseudo-random Sequence Generation Functions"
[ISO/IEC 14882-2014]	Subclause 26.5, "Random Number Generation"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/MSC50-CPP+Do+not+use+std%3A%3Arand%28%29+for+generating+pseudorandom+numbers>], Copyright (C) 1995-2016 Carnegie Mellon University. See [axivion_copyright_guide.pdf](#) for full details.

Configuration

Name	Explanation	Value
blacklist	Dictionary of header globbing to [list of] function name globbing(s) of forbidden functions.	dict(...)
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	6
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
forbidden_libfunc_call	Call to forbidden function.

CertC++-MSC51

Ensure your random number generator is properly seeded.

Input: IR

Source languages: C++

Details

A pseudorandom number generator (PRNG) is a deterministic algorithm capable of generating sequences of numbers that approximate the properties of random numbers. Each sequence is completely determined by the initial state of the PRNG and the algorithm for changing the state. Most PRNGs make it possible to set the initial state, also called the *seed state*. Setting the initial state is called *seeding* the PRNG.

Calling a PRNG in the same initial state, either without seeding it explicitly or by seeding it with a constant value, results in generating the same sequence of random numbers in different runs of the program. Consider a PRNG function that is seeded with some initial seed value and is consecutively called to produce a sequence of random numbers. If the PRNG is subsequently seeded with the same initial seed value, then it will generate the same sequence.

Consequently, after the first run of an improperly seeded PRNG, an attacker can predict the sequence of random numbers that will be generated in the future runs. Improperly seeding or failing to seed the PRNG can lead to [vulnerabilities](#), especially in security protocols.

The solution is to ensure that a PRNG is always properly seeded with an initial seed value that will not be predictable or controllable by an attacker. A properly seeded PRNG will generate a different sequence of random numbers each time it is run.

Not all random number generators can be seeded. True random number generators that rely on hardware to produce completely unpredictable results do not need to be and cannot be seeded. Some high-quality PRNGs, such as the `/dev/random` device on some UNIX systems, also cannot be seeded. This rule applies only to algorithmic PRNGs that can be seeded.

Noncompliant Code Example

This noncompliant code example generates a sequence of 10 pseudorandom numbers using the [Mersenne Twister](#) engine. No matter how many times this code is executed, it always produces the same sequence because the default seed is used for the engine.

```
#include <random>
#include <iostream>

void f() {
    std::mt19937 engine;

    for (int i = 0; i < 10; ++i) {
        std::cout << engine() << ", ";
    }
}
```

The output of this example follows.

```
1st run: 3499211612, 581869302, 3890346734, 3586334585, 545404204, 4161255391, 3922919429, 949333985, 2715962298, 1323567403,
2nd run: 3499211612, 581869302, 3890346734, 3586334585, 545404204, 4161255391, 3922919429, 949333985, 2715962298, 1323567403,
...
nth run: 3499211612, 581869302, 3890346734, 3586334585, 545404204, 4161255391, 3922919429, 949333985, 2715962298, 1323567403,
```

Noncompliant Code Example

This noncompliant code example improves the previous noncompliant code example by seeding the random number generation engine with the current time. However, this approach is still unsuitable when an attacker can control the time at which the seeding is executed. Predictable seed values can result in [exploits](#) when the subverted PRNG is used.

```
#include <ctime>
#include <random>
#include <iostream>

void f() {
    std::time_t t;
    std::mt19937 engine(std::time(&t));

    for (int i = 0; i < 10; ++i) {
        std::cout << engine() << ", ";
    }
}
```

Compliant Solution

This compliant solution uses `std::random_device` to generate a random value for seeding the Mersenne Twister engine object. The values generated by `std::random_device` are nondeterministic random numbers when possible, relying on random number generation devices, such as `/dev/random`. When such a device is not available, `std::random_device` may employ a random number engine; however, the initial value generated should have sufficient randomness to serve as a seed value.

```
#include <random>
#include <iostream>

void f() {
    std::random_device dev;
    std::mt19937 engine(dev());

    for (int i = 0; i < 10; ++i) {
        std::cout << engine() << ", ";
    }
}
```

The output of this example follows.

```
1st run: 3921124303, 1253168518, 1183339582, 197772533, 83186419, 2599073270, 3238222340, 101548389, 296330365, 3335314032,
2nd run: 2392369099, 2509898672, 2135685437, 3733236524, 883966369, 2529945396, 764222328, 138530885, 4209173263, 1693483251,
3rd run: 914243768, 2191798381, 2961426773, 3791073717, 2222867426, 1092675429, 2202201605, 850375565, 3622398137, 422940882,
...
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC51-CPP	Medium	Likely	Low	P18	L1

Related Guidelines

SEI CERT C Coding Standard	MSC32-C. Properly seed pseudorandom number generators
MITRE CWE	CWE-327 , Use of a Broken or Risky Cryptographic Algorithm CWE-330 , Use of Insufficiently Random Values CWE-337 , Predictable Seed in PRNG

Bibliography

[ISO/IEC 9899:2011]	Subclause 7.22.2, "Pseudo-random Sequence Generation Functions"
[ISO/IEC 14882-2014]	Subclause 26.5, "Random Number Generation"

Excerpt from SEI CERT C++ Coding Standard Wiki <https://wiki.sei.cmu.edu/confluence/display/cplusplus/MSC51-CP+Ensure+your+random+number+generator+is+properly+seeded>, Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
additional_blacklist	Sequence of additional forbidden class names where calling the default constructor is forbidden.	{}
blacklist	Set of class names where calling the default constructor is forbidden.	set['std::mt19937', 'std::subtract_with_carry_engine', 'std::mt19937_64', 'std::ranlux48_base', 'std::knuth_b', 'std::mersenne_twister_engine', 'std::ranlux24', 'std::minstd_rand0', 'std::default_random_engine', 'std::ranlux48', 'std::linear_congruential_engine', 'std::ranlux24_base']
level	Grouping of priorities into different levels	1
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
priority	Priority based on the combination of severity, likelihood and remediation cost	18
proper_seed_classes	List of acceptable Seed class names	['std::random_device']
random_generators	List of random generator classes that need to be initialized with a proper seed.	['std::minstd_rand0', 'std::mt19937', 'std::mt19937_64', 'std::ranlux24_base', 'std::ranlux48_base', 'std::ranlux24', 'std::ranlux48', 'std::knuth_b', 'std::default_random_engine', 'std::linear_congruential_engine', 'std::mersenne_twister_engine', 'std::subtract_with_carry_engine']
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
forbidden_default_constructor_call	Call to forbidden default constructor.
forbidden_seed	Random number generation engine may be initialized with non-compliant seed.

CertC++-MSC52

Value-returning functions must return a value from all exit paths.

Input: IR

Source languages: C++

Details

The C++ Standard, [stmt.return], paragraph 2 [\[ISO/IEC 14882-2014\]](#), states the following:

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.

A value-returning function must return a value from all code paths; otherwise, it will result in [undefined behavior](#). This includes returning through less-common code paths, such as from a *function-try-block*, as explained in the C++ Standard, [except.handle], paragraph 15:

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 (6.6.3).

Noncompliant Code Example

In this noncompliant code example, the programmer forgot to return the input value for positive input, so not all code paths return a value.

```
int absolute_value(int a) {
    if (a < 0) {
        return -a;
    }
}
```

Compliant Solution

In this compliant solution, all code paths now return a value.

```
int absolute_value(int a) {
    if (a < 0) {
        return -a;
    }
    return a;
}
```

Noncompliant Code Example

In this noncompliant code example, the *function-try-block* handler does not return a value, resulting in [undefined behavior](#) when an exception is thrown.

```
#include <vector>

std::size_t f(std::vector<int> &v, std::size_t s) try {
    v.resize(s);
    return s;
} catch (...) {
}
```

Compliant Solution

In this compliant solution, the exception handler of the *function-try-block* also returns a value.

```
#include <vector>

std::size_t f(std::vector<int> &v, std::size_t s) try {
    v.resize(s);
    return s;
} catch (...) {
    return 0;
}
```

Exceptions

MSC54-CPP-EX1: Flowing off the end of the `main()` function is equivalent to a `return 0;` statement, according to the C++ Standard, [basic.start.main], paragraph 5 [[ISO/IEC 14882-2014](#)]. Thus, flowing off the end of the `main()` function does not result in [undefined behavior](#).

MSC54-CPP-EX2: It is permissible for a control path to not return a value if that code path is never expected to be taken and a function marked `[[noreturn]]` is called as part of that code path or if an exception is thrown, as is illustrated in the following code example.

```
#include <cstdlib>
#include <iostream>
[[noreturn]] void unreachable(const char *msg) {
    std::cout << "Unreachable code reached: " << msg << std::endl;
    std::exit(1);
}

enum E {
    One,
    Two,
    Three
};

int f(E e) {
    switch (e) {
        case One: return 1;
        case Two: return 2;
        case Three: return 3;
    }
    unreachable("Can never get here");
}
```

Risk Assessment

Failing to return a value from a code path in a value-returning function results in [undefined behavior](#) that might be [exploited](#) to cause data integrity violations.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC52-CPP	Medium	Probable	Medium	P8	L2

Bibliography

[ISO/IEC 14882-2014]	Subclause 3.6.1, "Main Function" Subclause 6.6.3, "The <code>return</code> Statement" Subclause 15.3, "Handling an Exception"
--------------------------------------	---

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://wiki.sei.cmu.edu/confluence/display/cplusplus/MSC52-CPP.+Value-returning+functions+must+return+a+value+from+all+exit+paths>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	probable
priority	Priority based on the combination of severity, likelihood and remediation cost	8
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
lambda_return_missing_value	Return without value in non-void lambda expression
missing_return	Value-returning functions must return a value from all exit paths.
missing_return_in_lambda	Value-returning lambda expressions must return a value from all exit paths.
return_missing_value	Return without value in non-void function

CertC++-MSC53

Do not return from a function declared `[[noreturn]]`.

Input: IR

Source languages: C++

Note: Rule requires `CONFIG.Run_IRAnalysis_Checks = True`.

Details

The `[[noreturn]]` attribute specifies that a function does not return. The C++ Standard, [dcl.attr.noreturn] paragraph 2 [[ISO/IEC 14882-2014](#)], states the following:

If a function `f` is called where `f` was previously declared with the `noreturn` attribute and `f` eventually returns, the behavior is undefined.

A function that specifies `[[noreturn]]` can prohibit returning by throwing an exception, entering an infinite loop, or calling another function designated with the `[[noreturn]]` attribute.

Noncompliant Code Example

In this noncompliant code example, if the value `0` is passed, control will flow off the end of the function, resulting in an implicit return and [undefined behavior](#).

```
#include <cstdlib>

[[noreturn]] void f(int i) {
    if (i > 0)
        throw "Received positive input";
    else if (i < 0)
        std::exit(0);
}
```

Compliant Solution

In this compliant solution, the function does not return on any code path.

```
#include <cstdlib>

[[noreturn]] void f(int i) {
    if (i > 0)
        throw "Received positive input";
    std::exit(0);
}
```

Risk Assessment

Returning from a function marked `[[noreturn]]` to cause data-integrity violations.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC53-CPP	Medium	Unlikely	Low	P2	L3

Bibliography

[ISO/IEC 14882-2014]	Subclause 7.6.3, "noreturn Attribute"
Excerpt from SEI CERT C++ Coding Standard Wiki (https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046346), Copyright (C) 1995-2016 Carnegie Mellon University. See <code>axivion_copyright_guide.pdf</code> for full details.	

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
noreturnViolation	Do not return from a noreturn function.

CertC++-PRE30

Do not create a universal character name through concatenation.

Input: IR

Source languages: C++

Details

The C Standard supports universal character names that may be used in identifiers, character constants, and string literals to designate characters that are not in the basic character set. The universal character name `\unnnnnnnn` designates the character whose 8-digit short identifier (as specified by ISO/IEC 10646) is `nnnnnnnn`. Similarly, the universal character name `\unnnn` designates the character whose 4-digit short identifier is `nnnn` (and whose 8-digit short identifier is `0000nnnn`).

The C Standard, 5.1.1.2, paragraph 4 [[ISO/IEC 9899:2011](#)], says,

If a character sequence that matches the syntax of a universal character name is produced by token concatenation (6.10.3.3), the behavior is undefined.

See also [undefined behavior 3](#).

In general, avoid universal character names in identifiers unless absolutely necessary.

Noncompliant Code Example

This code example is noncompliant because it produces a universal character name by token concatenation:

```
#define assign(uc1, uc2, val) uc1##uc2 = val

void func(void) {
    int \u0401;
    /* ... */
    assign(\u04, 01, 4);
    /* ... */
}
```

Implementation Details

This code compiles and runs with Microsoft Visual Studio 2013, assigning 4 to the variable as expected.

GCC 4.8.1 on Linux refuses to compile this code; it emits a diagnostic reading, "stray '\'' in program," referring to the universal character fragment in the invocation of the `assign` macro.

Compliant Solution

This compliant solution uses a universal character name but does not create it by using token concatenation:

```
#define assign(ucn, val) ucn = val

void func(void) {
    int \u0401;
    /* ... */
    assign(\u0401, 4);
    /* ... */
}
```

Risk Assessment

Creating a universal character name through token concatenation results in undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
PRE30-C	Low	Unlikely	Medium	P2	L3

Bibliography

[ISO/IEC 10646-2003]	
[ISO/IEC 9899:2011]	Subclause 5.1.1.2, "Translation Phases"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/Zg4>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
universal_name_by_concat	Do not create a universal character name through concatenation

CertC++-PRE31

Avoid side effects in arguments to unsafe macros.

Input: IR

Source languages: C++

Details

An [unsafe function-like macro](#) is one whose expansion results in evaluating one of its parameters more than once or not at all. Never invoke an unsafe macro with arguments containing an assignment, increment, decrement, volatile access, input/output, or other expressions with side effects (including function calls, which may cause side effects).

The documentation for unsafe macros should warn against invoking them with arguments with side effects, but the responsibility is on the programmer using the macro. Because of the risks associated with their use, it is recommended that the creation of unsafe function-like macros be avoided. (See [PRE00-C.Prefer inline or static functions to function-like macros](#).)

This rule is similar to [EXP44-C. Do not rely on side effects in operands to sizeof, __Alignof, or __Generic](#).

Noncompliant Code Example

One problem with unsafe macros is [side effects](#) on macro arguments, as shown by this noncompliant code example:

```
#define ABS(x) ((x) < 0) ? -(x) : (x)

void func(int n) {
    /* Validate that n is within the desired range */
    int m = ABS(++n);

    /* ... */
}
```

The invocation of the `ABS()` macro in this example expands to

```
m = (((++n) < 0) ? -(++n) : (++n));
```

The resulting code is well defined but causes `n` to be incremented twice rather than once.

Compliant Solution

In this compliant solution, the increment operation `++n` is performed before the call to the unsafe macro.

```
#define ABS(x) (((x) < 0) ? -(x) : (x)) /* UNSAFE */  
  
void func(int n) {  
    /* Validate that n is within the desired range */  
    ++n;  
    int m = ABS(n);  
    /* ... */  
}
```

Note the comment warning programmers that the macro is unsafe. The macro can also be renamed `ABS_UNSAFE()` to make it clear that the macro is unsafe. This compliant solution, like all the compliant solutions for this rule, has undefined behavior if the argument to `ABS()` is equal to the minimum (most negative) value for the signed integer type. (See [INT32-C. Ensure that operations on signed integers do not result in overflow](#) for more information.)

Compliant Solution

This compliant solution follows the guidance of [PRE00-C. Prefer inline or static functions to function-like macros](#) by defining an inline function `iabs()` to replace the `ABS()` macro. Unlike the `ABS()` macro, which operates on operands of any type, the `iabs()` function will truncate arguments of types wider than `int` whose value is not in range of the latter type.

```
#include <complex.h>  
#include <math.h>  
  
static inline int iabs(int x) {  
    return ((x) < 0) ? -(x) : (x);  
}  
  
void func(int n) {  
    /* Validate that n is within the desired range */  
    int m = iabs(++n);  
    /* ... */  
}
```

Compliant Solution

A more flexible compliant solution is to declare the `ABS()` macro using a `_Generic` selection. To support all arithmetic data types, this solution also makes use of inline functions to compute integer absolute values. (See [PRE00-C. Prefer inline or static functions to function-like macros](#) and [PRE12-C. Do not define unsafe macros](#).)

According to the C Standard, 6.5.1.1, paragraph 3 [[ISO/IEC 9899:2011](#)]:

The controlling expression of a generic selection is not evaluated. If a generic selection has a generic association with a type name that is compatible with the type of the controlling expression, then the result expression of the generic selection is the expression in that generic association. Otherwise, the result expression of the generic selection is the expression in the `default` generic association. None of the expressions from any other generic association of the generic selection is evaluated.

Because the expression is not evaluated as part of the generic selection, the use of a macro in this solution is guaranteed to evaluate the macro parameter `v` only once.

```
#include <complex.h>  
#include <math.h>  
  
static inline long long llabs(long long v) {  
    return v < 0 ? -v : v;  
}  
static inline long labs(long v) {  
    return v < 0 ? -v : v;  
}  
static inline int iabs(int v) {  
    return v < 0 ? -v : v;  
}  
static inline int sabs(short v) {  
    return v < 0 ? -v : v;  
}  
static inline int scabs(signed char v) {  
    return v < 0 ? -v : v;  
}  
  
#define ABS(v) _Generic(v, signed char : scabs, \  
                      short : sabs, \  
                      int : iabs, \  
                      long : labs, \  
                      long long : llabs, \  
                      float : fabsf, \  
                      double : fabs, \  
                      long double : fabsl, \  
                      double complex : cabs, \  
                      float complex : cabsf, \  
                      long double complex : cabsl)(v)
```

```

void func(int n) {
    /* Validate that n is within the desired range */
    int m = ABS(++n);
    /* ... */
}

```

Generic selections were introduced in C11 and are not available in C99 and earlier editions of the C Standard.

Compliant Solution (GCC)

GCC's `__typeof__` extension makes it possible to declare and assign the value of the macro operand to a temporary of the same type and perform the computation on the temporary, consequently guaranteeing that the operand will be evaluated exactly once. Another GCC extension, known as `statement expression`, makes it possible for the block statement to appear where an expression is expected:

```
#define ABS(x) __extension__ ({ __typeof__ (x) tmp = x; \
                           tmp < 0 ? -tmp : tmp; })
```

Note that relying on such extensions makes code nonportable and violates [MSC14-C. Do not introduce unnecessary platform dependencies](#).

Noncompliant Code Example (`assert()`)

The `assert()` macro is a convenient mechanism for incorporating diagnostic tests in code. (See [MSC11-C. Incorporate diagnostic tests using assertions](#).) Expressions used as arguments to the standard `assert()` macro should not have side effects. The behavior of the `assert()` macro depends on the definition of the object-like macro `NDEBUG`. If the macro `NDEBUG` is undefined, the `assert()` macro is defined to evaluate its expression argument and, if the result of the expression compares equal to 0, call the `abort()` function. If `NDEBUG` is defined, `assert` is defined to expand to `((void)0)`. Consequently, the expression in the assertion is not evaluated, and no side effects it may have had otherwise take place in non-debugging executions of the code.

This noncompliant code example includes an `assert()` macro containing an expression (`index++`) that has a side effect:

```
#include <assert.h>
#include <stddef.h>

void process(size_t index) {
    assert(index++ > 0); /* Side effect */
    /* ... */
}
```

Compliant Solution (`assert()`)

This compliant solution avoids the possibility of side effects in assertions by moving the expression containing the side effect outside of the `assert()` macro.

```
#include <assert.h>
#include <stddef.h>

void process(size_t index) {
    assert(index > 0); /* No side effect */
    ++index;
    /* ... */
}
```

Exceptions

PRE31-C-EX1: An exception can be made for invoking an [unsafe macro](#) with a function call argument provided that the function has no [side effects](#). However, it is easy to forget about obscure side effects that a function might have, especially library functions for which source code is not available; even changing `errno` is a side effect. Unless the function is user-written and does nothing but perform a computation and return its result without calling any other functions, it is likely that many developers will forget about some side effect. Consequently, this exception must be used with great care.

Risk Assessment

Invoking an unsafe macro with an argument that has side effects may cause those side effects to occur more than once. This practice can lead to [unexpected program behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
PRE31-C	Low	Unlikely	Low	P3	L3

Related Guidelines

SEI CERT C Coding Standard	PRE00-C. Prefer inline or static functions to function-like macros PRE12-C. Do not define unsafe macros MSC14-C. Do not introduce unnecessary platform dependencies DCL37-C. Do not declare or define a reserved identifier
SEI CERT C++ Coding Standard	PRE31-CPP. Avoid side-effects in arguments to unsafe macros
CERT Oracle Secure Coding Standard for Java	EXP06-J. Expressions used in assertions must not produce side effects
ISO/IEC TR 24772:2013	Pre-processor Directives [NMP]
MISRA C:2012	Rule 20.5 (advisory)

Bibliography

[Dewhurst 2002]	Gotcha #28, "Side Effects in Assertions"
[ISO/IEC 9899:2011]	Subclause 6.5.1.1, "Generic Selection"
[Plum 1985]	Rule 1-11

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/agBj>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
<code>side_effect_in_unsafe_macro_call</code>	Side-effect in call to unsafe macro

CertC++-PRE32

Do not use preprocessor directives in invocations of function-like macros.

Input: IR

Source languages: C++

Details

The arguments to a macro must not include preprocessor directives, such as `#define`, `#ifdef`, and `#include`. Doing so results in [undefined behavior](#), according to the C Standard, 6.10.3, paragraph 11 [[ISO/IEC 9899:2011](#)]:

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, the behavior is undefined.

See also [undefined behavior 93](#).

This rule also applies to the use of preprocessor directives in arguments to a function where it is unknown whether or not the function is implemented using a macro. For example, standard library functions, such as `memcpy()`, `printf()`, and `assert()`, may be implemented as macros.

Noncompliant Code Example

In this noncompliant code example [[GCC Bugs](#)], the programmer uses preprocessor directives to specify platform-specific arguments to `memcpy()`. However, if `memcpy()` is implemented using a macro, the code results in undefined behavior.

```
#include <string.h>

void func(const char *src) {
    /* Validate the source string; calculate size */
    char *dest;
    /* malloc() destination string */
    memcpy(dest, src,
        #ifdef PLATFORM1
        12
        #else
        24
        #endif
    );
    /* ... */
}
```

Compliant Solution

In this compliant solution [[GCC Bugs](#)], the appropriate call to `memcpy()` is determined outside the function call:

```
#include <string.h>

void func(const char *src) {
    /* Validate the source string; calculate size */
    char *dest;
```

```

/* malloc() destination string */
#ifndef PLATFORM1
    memcpy(dest, src, 12);
#else
    memcpy(dest, src, 24);
#endif
/* ... */
}

```

Risk Assessment

Including preprocessor directives in macro arguments is undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
PRE32-C	Low	Unlikely	Medium	P2	L3

Bibliography

[GCC Bugs]	"Non-bugs"
[ISO/IEC 9899:2011]	6.10.3, "Macro Replacement"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/JYC2AQ>], Copyright (C) 1995-2016 Carnegie Mellon University. See axivion_copyright_guide.pdf for full details.

Configuration

Name	Explanation	Value
directives	Preprocessing directives to check for in macro arguments.	{'#if', '#ifdef', '#ifndef', '#elif', '#else', '#endif', '#pragma', '#warning', '#error', '#line', '#include', '#include_next', '#ident', '#region', '#endregion', '#asm', '#endasm', '#define', '#undef'}
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	2
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	medium

Possible Messages

Name	Message
pp_directive_as_macro_arg	Macro invocation argument looks like preprocessing directive

CertC++-SIG31

Do not access shared objects in signal handlers.

Input: IR

Source languages: C++

Details

Accessing or modifying shared objects in signal handlers can result in race conditions that can leave data in an inconsistent state. The two exceptions (C Standard, 5.1.2.3, paragraph 5) to this rule are the ability to read from and write to lock-free atomic objects and variables of type `volatile sig_atomic_t`. Accessing any other type of object from a signal handler is [undefined behavior](#). (See [undefined behavior 131](#).)

The need for the `volatile` keyword is described in [DCL22-C. Use volatile for data that cannot be cached](#).

The type `sig_atomic_t` is the integer type of an object that can be accessed as an atomic entity even in the presence of asynchronous interrupts. The type of `sig_atomic_t` is [implementation-defined](#), though it provides some guarantees. Integer values ranging from `SIG_ATOMIC_MIN` through `SIG_ATOMIC_MAX`, inclusive, may be safely stored to a variable of the type. In addition, when `sig_atomic_t` is a signed integer type, `SIG_ATOMIC_MIN` must be no greater than -127 and `SIG_ATOMIC_MAX` no less than 127. Otherwise, `SIG_ATOMIC_MIN` must be 0 and `SIG_ATOMIC_MAX` must be no less than 255. The macros `SIG_ATOMIC_MIN` and `SIG_ATOMIC_MAX` are defined in the header `<stdint.h>`.

According to the C99 Rationale [[C99 Rationale 2003](#)], other than calling a limited, prescribed set of library functions,

the C89 Committee concluded that about the only thing a [strictly conforming](#) program can do in a signal handler is to assign a value to a `volatile`

static variable which can be written uninterruptedly and promptly return.

However, this issue was discussed at the April 2008 meeting of ISO/IEC WG14, and it was agreed that there are no known [implementations](#) in which it would be an error to read a value from a `volatile sig_atomic_t` variable, and the original intent of the committee was that both reading and writing variables of `volatile sig_atomic_t` would be strictly conforming.

The signal handler may also call a handful of functions, including `abort()`. (See [SIG30-C. Call only asynchronous-safe functions within signal handlers](#) for more information.)

Noncompliant Code Example

In this noncompliant code example, `err_msg` is updated to indicate that the `SIGINT` signal was delivered. The `err_msg` variable is a character pointer and not a variable of type `volatile sig_atomic_t`.

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

enum { MAX_MSG_SIZE = 24 };
char *err_msg;

void handler(int signum) {
    strcpy(err_msg, "SIGINT encountered.");
}

int main(void) {
    signal(SIGINT, handler);

    err_msg = (char *)malloc(MAX_MSG_SIZE);
    if (err_msg == NULL) {
        /* Handle error */
    }
    strcpy(err_msg, "No errors yet.");
    /* Main code loop */
    return 0;
}
```

Compliant Solution {Writing `volatile sig_atomic_t`}

For maximum portability, signal handlers should only unconditionally set a variable of type `volatile sig_atomic_t` and return, as in this compliant solution:

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

enum { MAX_MSG_SIZE = 24 };
volatile sig_atomic_t e_flag = 0;

void handler(int signum) {
    e_flag = 1;
}

int main(void) {
    char *err_msg = (char *)malloc(MAX_MSG_SIZE);
    if (err_msg == NULL) {
        /* Handle error */
    }

    signal(SIGINT, handler);
    strcpy(err_msg, "No errors yet.");
    /* Main code loop */
    if (e_flag) {
        strcpy(err_msg, "SIGINT received.");
    }
    return 0;
}
```

Compliant Solution {Lock-Free Atomic Access}

Signal handlers can refer to objects with static or thread storage durations that are lock-free atomic objects, as in this compliant solution:

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <stdatomic.h>

#ifndef __STDC_NO_ATOMICS__
#error "Atomics are not supported"
#elif ATOMIC_INT_LOCK_FREE == 0
#error "int is never lock-free"
#endif

atomic_int e_flag = ATOMIC_VAR_INIT(0);

void handler(int signum) {
    e_flag = 1;
}

int main(void) {
    enum { MAX_MSG_SIZE = 24 };
    char err_msg[MAX_MSG_SIZE];
    #if ATOMIC_INT_LOCK_FREE == 1
        if (!atomic_is_lock_free(&e_flag)) {

```

```

        return EXIT_FAILURE;
    }
#endif
    if (signal(SIGINT, handler) == SIG_ERR) {
        return EXIT_FAILURE;
    }
    strcpy(err_msg, "No errors yet.");
    /* Main code loop */
    if (e_flag) {
        strcpy(err_msg, "SIGINT received.");
    }
    return EXIT_SUCCESS;
}

```

Exceptions

SIG31-C-EX1: The C Standard, 7.14.1.1 paragraph 5 [[ISO/IEC 9899:2011](#)], makes a special exception for `errno` when a valid call to the `signal()` function results in a `SIG_ERR` return, allowing `errno` to take an indeterminate value. (See [ERR32-C. Do not rely on indeterminate values of errno.](#))

Risk Assessment

Accessing or modifying shared objects in signal handlers can result in accessing data in an inconsistent state. Michal Zalewski's paper "Delivering Signals for Fun and Profit" [[Zalewski 2001](#)] provides some examples of [vulnerabilities](#) that can result from violating this and other signal-handling rules.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SIG31-C	High	Likely	High	P9	L2

Related Guidelines

ISO/IEC TS 17961:2013	Accessing shared objects in signal handlers [accsig]
MITRE CWE	CWE-662 , Improper Synchronization

Bibliography

[C99 Rationale 2003]	5.2.3, "Signals and Interrupts"
[ISO/IEC 9899:2011]	Subclause 7.14.1.1, "The <code>signal</code> Function"
[Zalewski 2001]	

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/GIEt>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
access_kinds	Access kinds (e.g. Reading_Operand_Interface, Writing_Operand_Interface, Address_Operand_Interface).	['Reading_Operand_Interface', 'Writing_Operand_Interface']
allow_c11_atomics	If set, don't report races on C11 atomic variables.	True
allow_volatile_sig_atomic_t	If set, don't report races on variables of type "volatile sig_atomic_t".	True
debug_output	Option to provide diagnostic output.	False
enter_critical_functions	List of function names to enter a critical region.	[]
enter_critical_macros	List of macro names to enter a critical region (macros must expand to asm() statement).	[]
excluded_routines	List of functions that should be excluded from check.	[]
excluded_subgraphs	List of entry functions to subgraphs that should be excluded as subgraph from check.	[]
exit_critical_functions	List of function names to exit a critical region.	[]
exit_critical_macros	List of macro names to exit a critical region (macros must expand to asm() statement).	[]
inspect_pointers	Whether pointer targets should be inspected to detect more global variable uses.	False
level	Grouping of priorities into different levels	2
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	likely
nested_critical_regions	If set to True, critical regions nest; if set to False, a single exit-critical-region terminates all open critical regions.	True
output_safe_accesses	When enabled, outputs not only unsafe variable accesses, but also the safe ones.	False
partitions	Dict with partition name as key and dict as value which may contain keys 'entries' and/or 'vectors' with lists of entry points or vector table variables respectively. If special partition '*IRQ*' to configure interrupt handlers is missing, all functions not reached by any of the other options are treated as interrupt handlers.	dict(...)
priority	Priority based on the combination of severity, likelihood and remediation cost	9
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high
report_cfg_based_critical_region_issues	Report unbalanced lock/unlock pairs within a routine. This has the same intention, but is slightly less strict than the purely syntactic check performed by the rule Parallelism-IncorrectCriticalSection.	False
show_identical_access	When enabled, outputs variable accesses of same kind (i.e., R/R and W/W).	False
show_object_number	Option for debugging (shows internal node numbers).	False
treat_types_as_atomic	List of type-patterns. A type-pattern is either a regular expression of a type name, or a triple of {min. alignment, max. size, type name-regex}. Each of the triple's components may be None. None is interpreted as general wildcard.	[]

Possible Messages

Name	Message
multiple_lock_add	Lock is acquired while it is already locked.
removed_nonexisting_lock	Lock is released, although it is not currently locked.
unbalanced_locks_path	Different control flow paths have different sets of locks.
unbalanced_locks_routine	Routine may return with different lock set than it is entered with ({in_set} vs {out_set}).

CertC++-SIG34

Do not call `signal()` from within interruptible signal handlers.

Input: IR

Source languages: C++

Details

A signal handler should not reassert its desire to handle its own signal. This is often done on *nonpersistent* platforms—that is, platforms that, upon receiving a signal, reset the handler for the signal to SIG_DFL before calling the bound signal handler. Calling `signal()` under these conditions presents a race condition. (See [SIG01-C. Understand implementation-specific details regarding signal handler persistence](#).)

A signal handler may call `signal()` only if it does not need to be [asynchronous-safe](#) (that is, if all relevant signals are masked so that the handler cannot be interrupted).

Noncompliant Code Example (POSIX)

On nonpersistent platforms, this noncompliant code example contains a race window, starting when the host environment resets the signal and ending when the handler calls `signal()`. During that time, a second signal sent to the program will trigger the default signal behavior, consequently defeating the persistent behavior implied by the call to `signal()` from within the handler to reassert the binding.

If the environment is persistent (that is, it does not reset the handler when the signal is received), the `signal()` call from within the `handler()` function is redundant.

```
#include <signal.h>

void handler(int signum) {
    if (signal(signum, handler) == SIG_ERR) {
        /* Handle error */
    }
    /* Handle signal */
}

void func(void) {
    if (signal(SIGUSR1, handler) == SIG_ERR) {
        /* Handle error */
    }
}
```

Compliant Solution (POSIX)

Calling the `signal()` function from within the signal handler to reassert the binding is unnecessary for persistent platforms, as in this compliant solution:

```
#include <signal.h>

void handler(int signum) {
    /* Handle signal */
}

void func(void) {
    if (signal(SIGUSR1, handler) == SIG_ERR) {
        /* Handle error */
    }
}
```

Compliant Solution (POSIX)

POSIX defines the `sigaction()` function, which assigns handlers to signals in a similar manner to `signal()` but allows the caller to explicitly set persistence. Consequently, the `sigaction()` function can be used to eliminate the race window on nonpersistent platforms, as in this compliant solution:

```
#include <signal.h>
#include <stddef.h>

void handler(int signum) {
    /* Handle signal */
}

void func(void) {
    struct sigaction act;
    act.sa_handler = handler;
    act.sa_flags = 0;
    if (sigemptyset(&act.sa_mask) != 0) {
        /* Handle error */
    }
}
```

```

if (sigaction(SIGUSR1, &act, NULL) != 0) {
    /* Handle error */
}

```

Although the handler in this example does not call `signal()`, it could do so safely because the signal is masked and the handler cannot be interrupted. If the same handler is installed for more than one signal, the signals must be masked explicitly in `act.sa_mask` to ensure that the handler cannot be interrupted because the system masks only the signal being delivered.

POSIX recommends that new applications should use `sigaction()` rather than `signal()`. The `sigaction()` function is not defined by the C Standard and is not supported on some platforms, including Windows.

Compliant Solution [Windows]

There is no safe way to implement persistent signal-handler behavior on Windows platforms, and it should not be attempted. If a design depends on this behavior, and the design cannot be altered, it may be necessary to claim a deviation from this rule after completing an appropriate risk analysis.

The reason for this is that Windows is a nonpersistent platform as discussed above. Just before calling the current handler function, Windows resets the handler for the next occurrence of the same signal to `SIG_DFL`. If the handler calls `signal()` to reinstall itself, there is still a race window. A signal might occur between the start of the handler and the call to `signal()`, which would invoke the default behavior instead of the desired handler.

Exceptions

SIG34-C-EX1: For implementations with persistent signal handlers, it is safe for a handler to modify the behavior of its own signal. Behavior modifications include ignoring the signal, resetting to the default behavior, and having the signal handled by a different handler. A handler reasserting its binding is also safe but unnecessary.

The following code example resets a signal handler to the system's default behavior:

```

#include <signal.h>

void handler(int signum) {
    #if !defined(_WIN32)
    if (signal(signum, SIG_DFL) == SIG_ERR) {
        /* Handle error */
    }
    #endif
    /* Handle signal */
}

void func(void) {
    if (signal(SIGUSR1, handler) == SIG_ERR) {
        /* Handle error */
    }
}

```

Risk Assessment

Two signals in quick succession can trigger a race condition on nonpersistent platforms, causing the signal's default behavior despite a handler's attempt to override it.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SIG34-C	Low	Unlikely	Low	P3	L3

Related Guidelines

CERT C Secure Coding Standard	SIG01-C. Understand implementation-specific details regarding signal handler persistence
ISO/IEC TS 17961:2013	Calling signal from interruptible signal handlers [sigcall]
MITRE CWE	CWE-479, Signal Handler Use of a Non-reentrant Function

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/rIDp>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	3
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	low

Possible Messages

Name	Message
signal_call_in_interruptible_signal_handler	Do not call signal() from within interruptible signal handlers.

CertC++-SIG35

Do not return from a computational exception signal handler.

Input: IR

Source languages: C++

Details

According to the C Standard, 7.14.1.1 [[ISO/IEC 9899:2011](#)], if a signal handler returns when it has been entered as a result of a computational exception (that is, with the value of its argument of SIGFPE, SIGILL, SIGSEGV, or any other implementation-defined value corresponding to such an exception) returns, then the behavior is undefined. (See [undefined behavior 130](#).)

The Portable Operating System Interface (POSIX®), Base Specifications, Issue 7 [[IEEE Std 1003.1:2013](#)], adds SIGBUS to the list of computational exception signal handlers:

The behavior of a process is undefined after it returns normally from a signal-catching function for a SIGBUS, SIGFPE, SIGILL, or SIGSEGV signal that was not generated by kill(), sigqueue(), or raise().

Do not return from SIGFPE, SIGILL, SIGSEGV, or any other implementation-defined value corresponding to a computational exception, such as SIGBUS on POSIX systems, regardless of how the signal was generated.

Noncompliant Code Example

In this noncompliant code example, the division operation has undefined behavior if denom equals 0. (See [INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors](#)) and may result in a SIGFPE signal to the program.)

```
#include <errno.h>
#include <limits.h>
#include <signal.h>
#include <stdlib.h>

volatile sig_atomic_t denom;

void sighandle(int s) {
    /* Fix the offending volatile */
    if (denom == 0) {
        denom = 1;
    }
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        return 0;
    }

    char *end = NULL;
    long temp = strtol(argv[1], &end, 10);

    if (end == argv[1] || 0 != *end ||
        ((LONG_MIN == temp || LONG_MAX == temp) && errno == ERANGE)) {
        /* Handle error */
    }

    denom = (sig_atomic_t)temp;
    signal(SIGFPE, sighandle);

    long result = 100 / (long)denom;
    return 0;
}
```

When compiled with some implementations, this noncompliant code example will loop infinitely if given the input 0. It illustrates that even when a SIGFPE handler attempts to fix the error condition while obeying all other rules of signal handling, the program still does not behave as expected.

Compliant Solution

The only portably safe way to leave a SIGFPE, SIGILL, or SIGSEGV handler is to invoke abort(), quick_exit(), or _Exit(). In the case of SIGFPE, the default action is abnormal termination, so no user-defined handler is required:

```
#include <errno.h>
#include <limits.h>
#include <signal.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        return 0;
    }

    char *end = NULL;
    long denom = strtol(argv[1], &end, 10);
```

```

if (end == argv[1] || 0 != *end ||
    ((LONG_MIN == denom || LONG_MAX == denom) && errno == ERANGE)) {
    /* Handle error */
}

long result = 100 / denom;
return 0;
}

```

Implementation Details

Some implementations define useful behavior for programs that return from one or more of these signal handlers. For example, Solaris provides the `sigfpe()` function specifically to set a `SIGFPE` handler that a program may safely return from. Oracle also provides platform-specific computational exceptions for the `SIGTRAP`, `SIGBUS`, and `SIGEMT` signals. Finally, GNU libsigsegv takes advantage of the ability to return from a `SIGSEGV` handler to implement page-level memory management in user mode.

Risk Assessment

Returning from a computational exception signal handler is [undefined behavior](#).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SIG35-C	Low	Unlikely	High	P1	L3

Bibliography

[IEEE Std 1003.1:2013]	2.4.1, Signal Generation and Delivery
[ISO/IEC 9899:2011]	Subclause 7.14.1.1, "The <code>signal</code> Function"

Excerpt from SEI CERT C++ Coding Standard Wiki [<https://www.securecoding.cert.org/confluence/x/QgGRAG>], Copyright (C) 1995-2016 Carnegie Mellon University. See `axivion_copyright_guide.pdf` for full details.

Configuration

Name	Explanation	Value
level	Grouping of priorities into different levels	3
likelihood	How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	unlikely
priority	Priority based on the combination of severity, likelihood and remediation cost	1
recommendation	Whether this check is classified as a recommendation or rule	False
remediation_cost	How expensive is it to comply with the rule?	high
signal_names	List of signal names that represent computational exceptions.	['SIGBUS', 'SIGFPE', 'SIGILL', 'SIGSEGV']

Possible Messages

Name	Message
noreturn_computational_exception	Do not return from a computational exception signal handler.

Rules in Group Expressions.Assignments

Expressions.Assignments-StrongTypeViolations

At assignment sites, the strong types of left and right side should match

Input: IR

Source languages: C, C++, C++/CLI

Details

The compiler usually ignores `typedefs`, replacing them with the underlying type. In contrast, this rule will warn on implicit conversions that would be needed when `typedefs` were respected as individual types. `typedefs` checked in this way are called *strong types*.

The rule allows to select the `typedefs` that should be seen as strong, and it allows to limit the operations (like assignment sites, comparison sites etc.) that should be checked. Both is done with the parameter `checks` which is an ordered dictionary of entries that provide `typedef` names (as globbing patterns) and the checks to perform for these `typedefs`.

The checks to be done are chosen from the below list as provided in module `strong_types`. They can be freely combined with Python operations to create iterables (e.g. lists or sets). The default setup only configures `['*', strong_types.all_checks]`.

which means that all typedefs [globbing pattern '*'] should be seen as strong, and being checked with all checks. An example custom configuration could be

```
checks = collections.OrderedDict([
('MY_INT', set(strong_types.all_checks) - set(strong_types.all_extracts)),
('Other*', strong_types.all_assignments),
])
```

In this case the typedef MY_INT would be assumed to be strong, and all checks except those at extracting operations would be applied. Also, all typedefs having a name that matches Other* would be strong and checked at assignments.

The following list summarizes the available predefined elementary checks (from module `strong_types`) as well as predefined groups of such checks which simplify typical selections of similar operations:

- Assignments to strong types (lint: A flag with modifiers)
 - `all_assignments` = [`assignment_lhs`, `field_init_lhs`, `init_lhs`, `return_lhs`, `param_lhs`]
 The elementary checks like `assignment_lhs` each cover a certain kind of assignment (assignment operator, field initialization, variable initialization, return statement, parameter passing). This includes the below special categories with non-zero/const right-hand side
 - `all_nonzero_assignments` = [`nonzero_assignment_lhs`, `nonzero_field_init_lhs`, `nonzero_init_lhs`, `nonzero_return_lhs`, `nonzero_param_lhs`]
 This checks assignments to a strong type except where the assigned value is literally given as 0
 - `all_nonconst_assignments` = [`nonconst_assignment_lhs`, `nonconst_field_init_lhs`, `nonconst_init_lhs`, `nonconst_return_lhs`, `nonconst_param_lhs`]
 This checks assignments to a strong type except where the assigned value is an integer constant expression, string literal, or address of a stack or global object
- Assignments from strong types (extracts, lint: X flag with modifiers)
 - `all_extracts` = [`assignment_rhs`, `field_init_rhs`, `init_rhs`, `return_rhs`, `param_rhs`]
- Use of strong types as operand in binary/ternary operators (joins, lint: J flag with modifiers)
 - `all_joins` = [`conditional`, `equality`, `relational`, `multiplicative`, `additive`, `bitwise`] The elementary checks cover the conditional operator (?:), (in)equality comparisons, relational comparisons, multiplicative operators, additive operators, and bitwise operators. This includes the below special categories with non-zero/const right-hand side
 - `all_nonzero_joins` = [`nonzero_conditional`, `nonzero_equality`, `nonzero_relational`, `nonzero_multiplicative`, `nonzero_additive`, `nonzero_bitwise`] This checks uses of strong types as operand in binary/ternary operators except where the assigned value is literally given as 0
 - `all_nonconst_joins` = [`nonconst_conditional`, `nonconst_equality`, `nonconst_relational`, `nonconst_multiplicative`, `nonconst_additive`, `nonconst_bitwise`] This checks uses of strong types as operand in binary/ternary operators except where the assigned value is an integer constant expression, string literal, or address of a stack or global object
- Operands that should be boolean (similar to lint: B flag)
 - `all_bools` = [`binary_logical`, `unary_logical`] The elementary checks inspect logical operators and check that the operands are of boolean type (native bool, if present in the language, and those configured via `CONFIG.User_Bool_Type`)
 Notice that Misra checks exist as well to check for proper use of bool.
- Combination of all checks
 - `all_checks` = `all_assignments` + `all_extracts` + `all_joins` + `all_bools`

An even more symbolical checking can be achieved with dimensional analysis. Strong typedefs being marked as dimensions behave differently at multiplication sites; for example, a multiplication where both operands are of type Meter will have result type Meter with strong types only, but it will have result type Meter*Meter when Meter is marked as a physical dimension. Dimensions can also provide conversion formulas, e.g. you could have a typedef Area, declare it as a dimension, and provide the formula `Area = Meter*Meter` so that the rule allows assigning the result of the multiplication to a variable of type Area.

Configuration

Name	Explanation	Value
<code>checks</code>	Configuration of checks to apply. This maps globbing patterns for the (qualified) typedef names to a list of checks, using an ordered dict to retain the order (last matching entry for a typedef wins). The checks are functions from module <code>strong_types</code> , or your own predicates that should take {site, is_lhs, rhs} as arguments.	dict(...)
<code>dimensions</code>	Dictionary using names of strong types that should be treated as physical dimensions as keys, mapping them to a string for a formula how this type depends on other dimensions. For example, the entries 'Meter' : "", 'Area' : 'Meter*Meter' would declare the strong types Meter and Area to be dimensions, and the formula allows assigning the product of two variables of type Meter to a variable of type Area.	dict(...)

Possible Messages

Name	Message
arithmetic_strong_typeViolation	Mismatched operand type in arithmetic operator causes implicit strong type conversion from {} to {}
assignment_strong_typeViolation	Assignment causes implicit strong type conversion from {} to {}
bitwise_strong_typeViolation	Mismatched operand type in bitwise operator causes implicit strong type conversion from {} to {}
comparison_strong_typeViolation	Comparison causes implicit strong type conversion from {} to {}
conditional_strong_typeViolation	Mismatched operand type in conditional operator causes implicit strong type conversion from {} to {}
field_init_strong_typeViolation	Field initialization causes implicit strong type conversion from {} to {}
init_strong_typeViolation	Initialization of {} causes implicit strong type conversion from {} to {}
logical_strong_typeViolation	Mismatched operand type in logical operator causes implicit strong type conversion from {} to {}
parameter_strong_typeViolation	Passing {} as argument for parameter {} causes implicit strong type conversion from {} to {}
return_strong_typeViolation	Return statement causes implicit strong type conversion from {} to {}
switch_strong_typeViolation	Switch case causes implicit strong type conversion from {} to {}

Rules in Group Generic

Generic-AnsiStringUse

Use wide strings when calling the Microsoft C/C++ standard library.

Input: IR

Source languages: C, C++

Details

Rationale

For portable applications, the [recommendation](#) is to use UTF-8 as string encoding.

However, on Windows, this is problematic because the standard library expects `char*` strings to use the system ANSI codepage. Moreover, it is impossible to access all files when passing file names as `char*` to the standard library, because some file names may contain Unicode characters that have no representation in the ANSI codepage.

In order to use UTF-8 internally in the application, all strings must be converted to/from wide-strings when calling Microsoft APIs. Even if UTF-8 is not used internally, the ANSI APIs must be avoided.

You should only enable this rule when analyzing a Windows build of your application. On other platforms, the normal functions taking `char*` can be used with UTF-8.

Windows API

This rule only verifies calls to functions from the Microsoft C/C++ standard library. It does not verify direct calls to the Windows API. You should `#define UNICODE` and `_UNICODE` to use the Windows API with wide strings.

Example

```
void open_file(const std::string& filename)
{
    std::ifstream f1(filename); // Not compliant:
    // 'filename' is interpreted using the ANSI codepage

    std::ifstream f2(widen(filename)); // Compliant.
}
```

`widen()` is a helper function that converts from UTF-8 `std::string` to `std::wstring` on Windows. On other platforms, it returns the `std::string` without any conversion.

See Also

<http://utf8everywhere.org/#windows>

Configuration

Name	Explanation	Value
stdlib_console_functions		dict{...}
stdlib_env_functions		dict{...}
stdlib_exec_functions		dict{...}
stdlib_file_functions		dict{...}
stdlib_time_functions		dict{...}

Possible Messages

Name	Message
ansi_console_call	Use of ANSI string for Windows console API.
ansi_env_call	Access to environment using ANSI strings.
ansi_exec_call	Process start using ANSI strings.
ansi_file_call	Use of ANSI string for file path.

Generic-BusyHeaders

Identify files with large product of includes and includers

Input: IR

Source languages: C, C++

Details

This rule identifies header files where the product of the number of transitively included headers and the number of translation units including the header exceeds a configurable threshold.

Rationale

Busy header files can cause long compilation times. Eliminate dependencies between headers to reduce the amount of code included in each translation unit.

You should enable this rule if you are having problems with long compilation times, and are not using precompiled headers.

Configuration

Name	Explanation	Value
detailed_entity	Output number of includes, includers and their product as well.	False
stop_on_files	Stop recursion into includes for this set of file names.	set{[]}
threshold	Maximum allowed product of includes/includers.	1000

Possible Messages

Name	Message
busy_header	File is too busy (large product of includes and includers)

Generic-CCComments

Checks for correctly used comment style in C units.

Input: IR

Source languages: C

Details

This rule checks that C++-style comments (//) are not used in C units.

Rationale

C++-style comments are not valid in C90. This rule allows configuring the type of comment that is considered invalid.

See Also

[Rule CPPComments](#)

Configuration

Name	Explanation	Value
exception	Start of a valid comment, even if it has an invalid prefix.	None
invalid	Start of an invalid comment.	//

Possible Messages

Name	Message
c_comment_style	Use of invalid comment style in C unit.

Generic-CPPComments

Checks for correctly used comment style in C++ units.

Input: IR

Source languages: C++

Details

This rule checks that comments starting with the invalid prefix are not used in C++ units. This rule allows configuring the type of comment that is considered invalid.

See Also

[Rule CComments](#)

Configuration

Name	Explanation	Value
exception	Start of a valid comment, even if it has an invalid prefix.	/**
invalid	Start of an invalid comment.	/*

Possible Messages

Name	Message
cpp_comment_style	Use of invalid comment style in C++ unit.

Generic-CapitalizeFunctions

Start functions and methods with an upper-case letter.

Input: IR

Source languages: C, C++, C#

Details

Functions and methods must start with an upper-case letter.

See Also

[Rule NamingConvention](#) supercedes this rule, as it provides more flexible support for naming conventions.

Configuration

Name	Explanation	Value
allow_underscores	Allow underscores in identifier.	True

Possible Messages

Name	Message
funcname_starts_lowercase	Function name does not start with upper-case letter.
funcname_starts_lowercase_or_has_underscore	Function name does not start with upper-case letter or contains underscores.

Generic-DoNotMixLogicalOperators

The logical operators `&&` and `||` should not appear together in a full expression.

Input: IR

Source languages: C, C++, C#

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
mixed_logical_operators	Full expression contains both <code>&&</code> and <code> </code>

Generic-DoxxygenCommentAtDefinition

Place Doxygen comments above (class and function) definitions.

Input: IR

Source languages: C, C++

Details

This rule warns when a class or function definition does not have a Doxygen comment.

See Also

[Rule DoxygenCommentInHeader](#) is a similar rule that expects the Doxygen comments in the header file instead of the implementation file.

Configuration

Name	Explanation	Value
allow_inherited	If True, a definition does not need documentation, if a corresponding declaration is documented.	False
allow_missing_documentation_on_private	If True, a class-member definition does not need documentation, if it is `private`.	False
allow_missing_documentation_on_protected	If True, a class-member definition does not need documentation, if it is `protected`.	False
doxygen_start	Start of a valid Doxygen comment.	[`/*`, `/**/`]
ignore_defaulted	If True, defaulted function declarations are not checked for comments.	False
ignore_deleted	If True, deleted function declarations are not checked for comments.	False
ignore_redefinitions	If True, method redefinitions are not checked as they can 'inherit' the comment from the redefined method.	False
ignore_tool_comments	An optional compiled regular expression. Comments where this regex finds a matching substring are ignored in the search for a doxygen comment (e.g. control-comments of other tools).	None
node_types	IR node types to check for preceding Doxygen comment.	`['Routine_Definition', 'Composite_Type_Definition']`

Possible Messages

Name	Message
missing_doxxygen_comment_before_def	No Doxygen comment before definition.

Generic-DoxxygenCommentInHeader

Place Doxygen comments above classes and functions in header files.

Input: IR

Source languages: C, C++

Details

This rule warns when a class or function declaration does not have a Doxygen comment.

See Also

[Rule DoxygenCommentAtDefinition](#) is a similar rule that expects the Doxygen comments above the definitions instead of the declarations.

Configuration

Name	Explanation	Value
allow_inherited	If True, a definition does not need documentation, if a corresponding declaration is documented.	False
allow_missing_documentation_on_private	If True, a class-member definition does not need documentation, if it is `private`.	False
allow_missing_documentation_on_protected	If True, a class-member definition does not need documentation, if it is `protected`.	False
doxygen_start	Start of a valid Doxygen comment.	{'/**', '///'}
ignore_defaulted	If True, defaulted function declarations are not checked for comments.	False
ignore_deleted	If True, deleted function declarations are not checked for comments.	False
ignore_out_of_template_method_definitions	Whether definitions of template class member functions should be accepted without a comment if the definition is outside the template class	False
ignore_redefinitions	If True, method redefinitions are not checked as they can 'inherit' the comment from the redefined method.	False
ignore_tool_comments	An optional compiled regular expression. Comments where this regex finds a matching substring are ignored in the search for a doxygen comment (e.g. control-comments of other tools).	None
node_types	IR node types to check for preceding Doxygen comment.	{'Routine_Interface', 'Composite_Type_Interface'}

Possible Messages

Name	Message
missing_doxxygen_comment_before_def	No Doxygen comment before declaration.

Generic-DuplicateIncludeGuard

Different include files should not use the same include guard.

Input: IR

Source languages: C, C++

Details

Every header file should have a unique include guard. Sometimes a header file is created as a copy of another header file and one forgets to update the include guard macro. This can result in problems when the same unit includes both header files.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
duplicate_include_guard	Duplicate include guard.

Generic-Filemarker

Reports a (suppressed) issue per file.

Input: IR

Source languages: C, C++, C#

Details

This rule generates one issue per file under analysis and immediately suppresses the issue. The rule is useful for checking completeness of the analysis. Note that you have to import the suppressed issues (see Reference Guide) into the database.

Configuration

Name	Explanation	Value
justification_checker	Can be set to a callable classmethod taking (and possibly modifying) a style violation instance. Will be called per issue and can be used to check a disabled issue's justification, for example, against permitted ones.	the_justification_checker(['klass', 'sv'])

Possible Messages

Name	Message
included_file	File is checked.

Generic-FORBIDDENFUNCTIONS

Do not call certain functions from certain header files.

Input: IR

Source languages: C, C++

Details

This rule prevents the use of a configurable set of functions.

Configuration

Name	Explanation	Value
blacklist	Dictionary of header globbing to (list of) function name globbing(s) of forbidden functions.	dict(...)

Possible Messages

Name	Message
forbidden_libfunc_call	Call to forbidden function.

Generic-FORBIDDENMACROS

Do not use certain macros.

Input: IR

Source languages: C, C++

Details

This rule prevents the use of a configurable set of macros.

Configuration

Name	Explanation	Value
blacklist	Dictionary of header globbing to (list of) macro name globbing(s) of forbidden macros.	dict(...)

Possible Messages

Name	Message
forbidden_macro	Use of forbidden macro.

Generic-ForbiddenTokens

Do not use certain token sequences.

Input: IR

Source languages: C, C++, C#

Details

This rule prevents the use of a configurable set of keywords.

Configuration

Name	Explanation	Value
blacklist	List of tokens that must not appear in source code.	['const', 'int', 'unsigned', 'char']

Possible Messages

Name	Message
forbidden_token	Use of forbidden token.

Generic-FormatSpecifier

Validates the use of format specifiers.

Input: IR

Source languages: C, C++

Details

This rule checks for incorrect use of format specifiers with the `printf`/`scanf` family of functions.

Due to these functions using C varargs, compilers cannot check that the parameter types expected by the function. In cases where the `format` argument to these function is a string literal, this rule will verify that the remaining arguments are compatible with those expected by the format string. This rule also checks for invalid combination of flags/modifiers and conversion specifiers.

Configuration

Name	Explanation	Value
allow_extra_args	Whether to allow additional arguments that are not used by the format string.	False
allow_gnu_extensions	Whether to allow the GNU extensions to format specifications.	False
allow_unknown_specs	Whether to allow unknown format specifications. It may be necessary to set this option when using implementation-specific extensions. Arguments are not checked when the format string contains unknown format specifications.	False
functions	A dictionary mapping the names of the functions to check, to a triple {function_kind, fmt_param_index, arg_start_index} where function_kind is either 'printf' or 'scanf', fmt_param_index is the index of the format-string parameter, and arg_start_index is the index of the first variadic argument.	dict(...)

Possible Messages

Name	Message
arg_type_mismatch	{} expects argument of type '{}', but argument {} has type '{}'
buffer_too_small	{} may write up to {} characters to buffer of size {}.
invalid_conversion	Invalid or non-standard conversion specification
matching_arg_expected	{} expects a matching '{}' argument
precision_for_conversion	Precision must not be used with %{} conversion specifier
too_many_args	Too many arguments for format.
unknown_buffer_size	Potential buffer overflow: {} used with buffer of unknown size. (disabled)
unlimited_read	Potential buffer overflow: {} has no limit on amount of characters read.
unsupported_assignment_suppression	%n does not support assignment suppression
unsupported_field_width	%n does not support field width
unsupported_flags	%n does not support flags
unsupported_flags_modifiers	Cannot use any flags or modifiers with '%%'
unsupported_hash	%{} does not support the '#' flag
unsupported_i_flag	%{} does not support the 'l' flag
unsupported_length_modifier	%{} does not support the '{}' length modifier
unsupported_tick	%{} does not support the "" flag
unsupported_zero	%{} does not support the '0' flag

Generic-IncludeKind

Use the correct include kind "" vs. <>.

Input: IR

Source languages: C, C++

Details

Use "" to include user headers, and <> to include system headers.

Example

```
#include <string>
#include <Python.h>
#include "MyProgram.h"
```

Configuration

Name	Explanation	Value
enforced_system_headers	Iterable of {basepath-relative} file names that require #include <> despite being a user-include file. Globbing patterns are supported.	None
enforced_user_headers	Iterable of {basepath-relative} file names that require #include "" despite being a system-include file. Globbing patterns are supported.	None

Possible Messages

Name	Message
sys_include_of_header	Include with "" instead of <>.
user_include_of_sysheader	Include with <> instead of "".

Generic-InitializeAllFieldsInConstructor

A constructor must initialize all data members of the class.

Input: IR

Source languages: C, C++

Details

If a constructor leaves data members uninitialized, accessing those data members may cause undefined behavior. This rule enforces the initialization of all data members.

If the configuration option `report_missing_field_constructors` is enabled, this rule will also warn when fields are implicitly initialized using their default constructor.

If the configuration option `report_missing_base_constructors` is enabled, this rule will warn when a base class is implicitly initialized using its default constructor.

See Also

[Rule MissingConstructor](#) checks that a constructor exists in every class.

Configuration

Name	Explanation	Value
<code>init_functions</code>	Names of functions to be inspected as well when called directly from constructor.	{'Init', 'init'}
<code>inspect_directly_called_methods</code>	Inspect all methods directly called from constructor.	False
<code>only_member_initializer_list</code>	Only inspect member initializer list and not the constructor body/methods.	False
<code>report_missing_base_constructors</code>	Enables detection of constructors which rely on implicit base constructor calls.	False
<code>report_missing_field_constructors</code>	Enables detection of constructors which rely on implicit field constructor calls.	False

Possible Messages

Name	Message
<code>implicit_field_init</code>	Field is only implicitly initialized in constructor.
<code>missing_base_class_init</code>	Base class is not explicitly initialized in constructor.
<code>missing_field_init</code>	Field is not initialized in constructor.

Generic-InitializeAllVariables

Initialize all variables when defining them.

Input: IR

Source languages: C, C++

Details

Reading uninitialized variables causes undefined behavior. This rule enforces the immediate initialization of variables at their definition point.

See Also

[Rule InitializeAllFieldsInConstructor](#) (equivalent rule for member variables)

Configuration

Name	Explanation	Value
<code>ignore_arrays</code>	If True, array variables are not checked.	False
<code>ignore_composites</code>	If True, struct/class/union variables are not checked.	False
<code>ignore_types</code>	Set of type names to ignore. Can be used to relax the initialization requirement for specific typedefs or struct/classes/unions.	set()

Possible Messages

Name	Message
<code>uninitialized_variable_definition</code>	Variable not initialized at definition point.

Generic-LineBreaks

You have to use the specified sort of line breaks.

Input: IR

Source languages: C, C++, C#

Configuration

Name	Explanation	Value
allowed_line_break_form	Selects the form of line break that is allowed ('\r', '\r\n' or '\n').	
excluded_files	Globbing patterns for files to exclude from this check in addition to -exclude.	[]
file_types	Selects file types to be checked: Primary_File, User_Include_File, System_Include_File.	('Primary_File', 'User_Include_File')

Possible Messages

Name	Message
bad_line_break	Line break is not of the allowed form {!r}
bad_line_break_block	Line breaks from here on up to line {} (inclusive) are not of the allowed form {!r}
file_using_bad_line_breaks	All line breaks in this file are not of the allowed form {!r}

Generic-LocallInclude

Replace #include with forward declaration or more precise #include where possible

Input: IR

Source languages: C, C++

Details

Including as few headers as possible makes the dependencies in the code more explicit, and can help reduce compilation time. This check is the stylecheck version of the tool 'incprof'.

Configuration

Name	Explanation	Value
accept_incomplete_arguments	Names of templates for which incomplete types should be accepted as allowed template type arguments (so forward declarations of these arguments are enough to instantiate the template).	['shared_ptr', 'unique_ptr', 'weak_ptr', 'auto_ptr', 'default_delete', 'enable_shared_from_this', 'list', 'forward_list', 'vector', 'allocator', 'polymorphic_allocator', 'is_void', 'is_null_pointer', 'is_integral', 'is_floating_point', 'is_array', 'is_pointer', 'is_lvalue_reference', 'is_rvalue_reference', 'is_member_object_pointer', 'is_member_function_pointer', 'is_enum', 'is_union', 'is_class', 'is_function', 'is_reference', 'is_arithmetic', 'is_fundamental', 'is_object', 'is_scalar', 'is_compound', 'is_member_pointer', 'is_const', 'is_volatile', 'is_signed', 'is_unsigned', 'rank', 'extent', 'is_same', 'remove_const', 'remove_volatile', 'remove_cv', 'add_const', 'add_VOLATILE', 'add_cv', 'remove_reference', 'add_lvalue_reference', 'add_rvalue_reference', 'make_signed', 'make_unsigned', 'remove_extent', 'remove_all_extents', 'remove_pointer', 'add_pointer', 'decay', 'enable_if', 'conditional']
avoid_uncertain_messages	Whether messages should be omitted if they are tagged as uncertain due to possibly unknown symbol references in templates.	True
context_macros	List of macro names which are intentionally not defined inside a file or in files included from it. The names can contain regexp patterns.	[]
enums_can_be_redeclared	Whether local redeclarations of enums should be allowed to avoid #includes	False
interface_headers	Globbing patterns for files which should be seen as ok although they might just #include files without using them. #includes to these interface files are also treated as ok.	[]
limit_for_moving	If an include is unused locally, but used in client files or inside subsequently included headers, this limit determines the maximum number of such target files up to which a Move_Include suggestion will be generated; if there are too many target files, a Move_Include_With_Many_Clients will be reported instead.	3
limit_for_replacing	If an include can be replaced by forward declarations and other, more precise #includes covering only parts of the previous #include, this limit determines the maximum number of such replacements up to which such a suggestion is generated; if there are too many replacements, the include is accepted instead.	3
routines_can_be_redeclared	Whether local redeclarations of functions should be allowed to avoid #includes	False
typedefs_can_be_redeclared	Whether local redeclarations of typedefs should be allowed to avoid #includes	False

Possible Messages

Name	Message
add_include	#include {srcfile}
add_symbol_declaration	declaration of {sym}
bad_pch_use	PCH-#include not first in file and thus not used as PCH
circular_include	Circular #include
contents_covered_include	#include can be removed as it only provides symbols covered by other #include
covered_include	#include can be removed as it is covered by other #include
local_decl_instead_of_include	#include can be replaced with local forward declaration of {sym}
local_def_instead_of_include	#include can be replaced with local redefinition of {sym}
more_precise_include	#include can be replaced with #include {srcfile}
move_include	#include can be moved into {srcfile}
move_include_with_many_clients	#include is unused locally, but there are many other files possibly relying on it
unused_include	#include can be removed as file does not use anything from it

Generic-LocalScope

Declare variables as local as possible.

Input: IR

Source languages: C, C++

Details

Defining variables in the minimum block scope possible reduces the visibility of those variables. This makes the code more readable as less code needs to be understood to see all usage of the variable.

Configuration

Name	Explanation	Value
allow_moving_to_other_primary_file	When a variable is only used in a single file X, but currently implemented in a different file Y, this controls whether the check suggests to move it into X and make it static there.	False
check_loop_counter	Whether to report for-loop counters being declared before the loop that could be declared in the for-init part instead from C99 on	False
exclude_dllexport	If True, no suggestions will be produced to make functions marked as dllexport or dllimport static in a primary file.	True
exclude_function_locals	If True, variables that could even be function-local are not reported.	True
exclude_undefined	Whether only-declared symbols should be reported as well	True
ignore_types	Variables of types named here are ignored in this check. Globbing patterns are supported.	[]
move_global_const_into_function	Controls suggestions for global constants that could be declared locally in a function.	True
only_check_unit_locals	Whether only global static variables should be checked.	False
report_global_constants	Whether unused global constants should be reported.	False
report_nonstatic_functions	Whether functions that can be made unit-local (static) should be reported.	False
report_undefined_variables	Whether only-declared variables should be reported.	True
report_unused	Report unused variables.	True
template_args_can_be_static	Whether functions whose address is used as template argument can be made static (some compilers, like Microsoft's, don't allow it then).	False
treat_initialization_as_use	Whether an explicit initialization should be considered a use of the variable.	False
treat_side_effect_constructors_as_use	Whether variables should be seen as used if they are of a class type and initialized through a call to a constructor having a side-effect, e.g. std::lock_guard	False

Possible Messages

Name	Message
function_file_static	{ } can be declared static in primary file.
locality_block	{ } can be declared in a more local scope.
locality_function	Global { } can be declared inside function.
locality_loop_init	{ } can be declared in the for-loop's initialization.
unused_variable	Variable is unused.
var_file_static	{ } can be declared static in primary file.

Generic-MaxComplexity

Functions must not exceed cyclomatic complexity limit.

Input: IR

Source languages: C, C++, C#

Details

Complex control flow can make code hard to understand. Try splitting up the code into multiple functions that can be understood individually.

Note

Cyclomatic complexity is also available as a metric, which can be configured to produce a metric violation when limits are exceeded. This stylecheck rule is available an alternative to the metric.

Configuration

Name	Explanation	Value
maxcomplexity	Maximum acceptable cyclomatic complexity.	15
show_value	Whether metric value should be displayed.	False

Possible Messages

Name	Message
cyclomatic_complexity_with_value	Cyclomatic complexity of { } exceeds limit of { }.
excessive_cyclomatic_complexity	Cyclomatic complexity exceeds limit of { }.

Generic-MaxConditions

An expression must not have more than N logical operators.

Input: IR

Source languages: C, C++, C#

Details

This rule warns when more than the configured number of logical operators (&& or ||) are used within a single expression.

As an exception, this rule does not warn about complex conditions within custom operator == or operator != definitions. Those operators are often implemented by compare all the class members in a long chain of logical operators.

Rationale

Complex conditions can be hard to understand, and should be split into several statements or extracted into a function.

Configuration

Name	Explanation	Value
ignore_in	Names of routines to ignore for this check.	('operator==', 'operator!=')
maxconditions	Maximum acceptable number of conditions in an expression.	7

Possible Messages

Name	Message
excessive_condition_complexity	An expression must not have more than {} logical operators.

Generic-MaxNesting

A function must have a nesting level less than or equal N.

Input: IR

Source languages: C, C++, C#

Details

Avoid exceeding the configured maximum nesting level. Use guard clauses to reduce nesting, or extract code into separate functions.

Configuration

Name	Explanation	Value
count_elseif	Whether nesting should be increased for else if	False
maxnesting	Maximum acceptable nesting level. Statements on top-level have a nesting level of 1.	10

Possible Messages

Name	Message
excessive_nesting	Nesting level {} greater than {}.

Generic-MaxOneStmtPerLine

Do not put more than one statement in a line.

Input: IR

Source languages: C, C++, C#

Details

To keep the code readable, put each statement on its own line.

Example

```
// BAD:  
a = 1; b = 2;  
  
// GOOD:  
a = 1;  
b = 2;
```

Configuration

Name	Explanation	Value
ignore_stmts	Statements to be ignored when counting statements.	['Statement_Sequence']

Possible Messages

Name	Message
multiple_statements_per_line	Multiple statements per line.

Generic-MaxParams

A function must not have more than N parameters.

Input: IR

Source languages: C, C++, C#

Configuration

Name	Explanation	Value
ignore_inherited	Do not report functions inheriting an unacceptable number of parameters.	False
maxparams	Maximum acceptable number of parameters.	7

Possible Messages

Name	Message
excessive_parameter_number	Function with {} parameters more than {}.

Generic-MissingConstructor

Each class needs an explicitly declared constructor.

Input: IR

Source languages: C++

Details

If a class does not have an explicitly declared constructor, the C++ compiler will generate a default constructor. The compiler-generated default constructor will leave POD fields uninitialized, which may cause undefined behavior if the class is used without proper initialization.

See Also

[Rule InitializeAllFieldsInConstructor](#) ensures that constructors do not leave fields uninitialized.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
missing_constructor	Class does not have an explicitly declared constructor.

Generic-MissingDestructor

Each class needs an explicitly declared destructor.

Input: IR

Source languages: C++

Details

This rule ensures that destruction semantics are made clear by providing an explicit destructor.

See Also

[Rule of Three](#) demands a destructor for classes that have explicitly declared copy/move constructors or assignment operators, but does not warn in cases where the compiler-provided destructor suffices.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
missing_explicit_destructor	Class does not have an explicitly declared destructor.

Generic-MissingIncludeGuard

Include files need an include guard.

Input: IR

Source languages: C, C++

Details

Every header file should have an include guard. This ensures that including the file multiple times into the same compilation unit (e.g. indirect inclusions through other headers) do not cause compiler errors due to duplicate type definitions.

Include guards also help reduce compilation time, as compilers that recognize the include guard can avoid repeatedly scanning the header file.

Example

```
#ifndef DIR_FILE_H
#define DIR_FILE_H

...
#endif
```

The identifiers used for include guards must be unique. If header file names are not necessarily unique in your project, include the directory name in the identifier.

Configuration

Name	Explanation	Value
macro_name_restrictions	Python iterable of functions with parameters (file, define, macro) to perform additional checks on the macro used as include guard in the given file. Should return None if macro is accepted, else a message to print for the violation. The parameters are: PIR File node for the header, token for the macro #define or #pragma once, and the name of the macro or the string 'pragma once'.	None

Possible Messages

Name	Message
include_guard_missing	Missing include guard.
include_guard_partial	Include guard does not cover complete file.
include_guard_with_wrong_define	Bad include guard: #define uses wrong macro, should be {}.
include_guard_without_define	Incomplete include guard: missing #define {}.

Generic-MissingInlineDefinition

Inline functions must be defined in every compilation unit where they are declared.

Input: IR

Source languages: C

Details

If a compilation unit declares an inline function, but does not contain a definition, undefined behavior occurs.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
missing_inline_def_in_multiple_units	Inline function is missing definition in {} compilation units (e.g. '{}')
missing_inline_def_in_single_unit	Inline function is missing definition in compilation unit '{}'

Generic-MissingOverride

Override of functions is only permitted with keyword override.

Input: IR

Source languages: C++

Details

This rule requires the use of the C++11 `override` keyword when overriding a method.

Example

```
class A
{
    virtual void foo();
```

```

}
class B : public A
{
    void foo() override;
}

```

See Also

Rule MisraC++-10.3.2

Configuration

Name	Explanation	Value
allow_final	If set to True, don't report overriding virtual functions declared with final.	False
ignore_destructors	If set to False, also report destructors. Note that not all compilers support this.	True

Possible Messages

Name	Message
missing_override	Override of functions is only permitted with keyword override.

Generic-MissingParameterAssert

Assert parameters before actual function body.

Input: IR

Source languages: C, C++

Details

This rule requires that every parameter is used in at least one `assert` statement

Configuration

Name	Explanation	Value
macro_names	Specifies the names of the assertion macros.	set(['assert'])

Possible Messages

Name	Message
missing_parameter_assert	Missing assertion for parameter(s).

Generic-MissingSelfHeaderInclude

Each compilation unit shall include its own header file.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
missing_selfheader_include	Compilation unit does not include its own header file.

Generic-NamingConvention

Check named entities against naming conventions.

Input: IR

Source languages: C, C++, C#

Details

This is a highly flexible rule that checks names against a configurable list of conditions.

In the current configuration, this rule will check:

- Const data member must be all upper-case
- Const global variable must be all upper-case
- Data member must be all lower-case
- Data member of a class must start with "m_"
- Enumeration type must use camel-case, starting with upper-case letter
- Enumerator must be all upper-case
- Enumerators must start with common prefix
- Function must be all lower-case
- Global variable must be all lowercase and start with g_
- Label must be all lower-case
- Local variable must be all lower-case
- Macro must be all upper-case
- Namespace must be all lower-case
- Parameter must be all lower-case
- Private or protected function member must be prefixed by underscore, use camel-case, starting with lower-case character
- Public function member must use camel-case, starting with lower-case character
- Static data member must be all lower-case
- Static function member must be all lower-case
- Typedef type must use camel-case, starting with upper-case letter
- User-defined type must use camel-case, starting with upper-case letter

Configuration

Name	Explanation	Value
excluded_global_functions	Names of global functions to which the check should not be applied.	('main', 'WinMain', 'wmain', 'wWinMain', 'DllMain')
naming	Each partial naming rule for the identifier of a certain node type has the following format: node_type_name : [(checker, message), (checker, message), ...] "checker" can be either a regular expression (in which case it must match the identifier) or a complex checker consisting of a guarding predicate (guard) and an regular expression (in which case the guard governs if the regexp must match). If a checker fails, its message is issued. All checkers in the list must match in order to get no message, i.e., if you want to state different regexps for different combinations of aspects (visibility etc), make sure the guards create the needed disjointness of your regexp checks. If multiple checkers fail for the same identifier, you will end up with multiple messages for that identifier. Some aspects are expressed by attributes (constness, visibility), some by the hierarchy (staticness).	dict(...)

Generic-NoAbsoluteInclude

Do not use absolute path names in #include.

Input: IR

Source languages: C, C++

Details

Absolute path names are non-portable and may cause problems when compiling the code on other machines.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
absolute_include	#include with absolute path

Generic-NoAssembler

Do not use the assembler.

Input: IR

Source languages: Assembler, C, C++

Details

Inline assembler is non-portable, and may use different syntax for different compilers even on the same target platform.

When choosing specific CPU instructions, the compiler's intrinsic functions should be preferred over inline assembler.

If the use of assembly language is necessary, consider putting the assembly code into separate `.asm` files.

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
use_of_assembler	Use of assembler.

Generic-NoAutoType

Do not use the C++11 auto type specifier.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_for_loop_counter	Disables message if auto is used to declare a for-loop counter.	False
allow_generic_lambda_parameters	Allows auto as parameter type in a generic lambda	False
allow_nonfundamental_initializer	Allows auto to declare variables having a function call or initializer of non-fundamental type	False
allow_template_instance	Disables message if auto stands for a template instance.	False
allowed_contexts	Set of context predicates in which auto is allowed.	set[[])
allowed_types	Set of types for which auto is allowed, given as name pattern, function, or LIR class name.	set[[])

Possible Messages

Name	Message
cpp11_auto	Use of C++11 auto type specifier.

Generic-NoCCasts

Do not use C-style casts in C++ code.

Input: IR

Source languages: C++

Details

C++-style casts (`static_cast`, `dynamic_cast`, `const_cast` and `reinterpret_cast`) make the intended semantics of the cast clear to readers of the code.

Configuration

Name	Explanation	Value
allow_void_cast	If True, <code>(void)</code> is always allowed, else only for return value of calls.	True
allow_void_cast_on_call	If True, <code>(void)</code> is allowed, for return value of calls.	True

Possible Messages

Name	Message
c_cast	Use of C-style cast in C++ unit.

Generic-NoCFunctionCall

Do not call C functions from C++ code.

Input: IR

Source languages: C++

Details

This rule prevents the call of `extern "C"` functions from C++ functions.

Configuration

Name	Explanation	Value
allowlist	List of C function names that are allowed to be called from C++ code.	<code>{'__assert_fail'}</code>

Possible Messages

Name	Message
c_function_call	Call to C function from C++ code.

Generic-NoCHeaderInclude

Do not use C standard headers when a C++ one is available.

Input: IR

Source languages: C++

Details

In C++ code, prefer the C++ names of the C standard headers. For example, use `#include <cassert>` instead of `#include <assert.h>`.

Also, when including C++ standard headers, prefer the standard name without the `.h` suffix.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
c_lib_header	Include <{}> instead of <{}>.
cpp_lib_header_with_suffix	Include <{}> instead of <{}>.

Generic-NoCPPStructs

Do not use structs in C++, rather use classes.

Input: IR

Source languages: C++

Details

In C++, classes and structs are equivalent except for the default visibility of members (`public` in struct, `private` in class).

Configuration

Name	Explanation	Value
require_class_for_pod	Whether POD types should be classes as well	True

Possible Messages

Name	Message
cpp_struct	Use of struct in C++ unit.

Generic-NoCharPointer

Do not use `char*` in C++ code.

Input: IR
Source languages: C++

Details

This rule prevents declarations of type `char*` in C++ code.

Rationale

C++ code should prefer the use of `std::string` over C strings.

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
cpp_char_pointer	Use of <code>char*</code> in C++ code.

Generic-NoCommaSequence

Do not use the C comma sequence operator.

Input: IR
Source languages: C, C++

Details

Code tends to be easier to read if side effects are written as separate statements.

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
use_of_comma	Use of comma sequence operator.

Generic-NoCompilerWarnings

Fix all compiler warnings and errors.

Input: IR
Source languages: C, C++

Details

Compiler warnings indicate problems with the source code and should thus be taken seriously.

Configuration

Name	Explanation	Value
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	If provided, only messages of these types are reported.	None
reported_severities	List of severities to display.	{'error', 'warning'}
use_error_number	Whether the error number from the frontend should be used.	True
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	False

Generic-NoConditionalOperator

Do not use the conditional operator.

Input: IR

Source languages: C, C++

Details

The conditional operator `{a ? b : c}` can make code hard to read, especially for unexperienced programmers. Use more readable `if / else` statements instead.

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
use_of_conditional_operator	Use of conditional operator.

Generic-NoConstCasts

Do not use `const_cast`.

Input: IR

Source languages: C++

Details

Const casts should be used only for interoperability with 3rd-party code that is not const-correct. In other cases, changing the code for const-correctness should be preferred over the use of `const_cast`.

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
const_cast	Use of <code>const_cast</code> .

Generic-NoConstOnRHS

Put constants on left side of equality comparisons in conditions.

Input: IR

Source languages: C, C++

Details

When a variable is compared with a constant, put the constant on the left-hand side of the comparison. This coding style is known as "Yoda conditions", and helps prevent accidental assignments when the `==` operator in equality comparison is mistyped as the assignment operator `=`.

Example

```
if (10 == value) ...
```

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
rhs_rvalue_should_be_lhs	Put rvalue operand on left side of equality comparisons.

Generic-NoDebugMacro

Do not use certain debugging macros.

Input: IR

Source languages: C, C++

Details

This rule prevents the use of a configurable set of macros.

See Also

[Rule Generic-ForbiddenMacros](#)

Configuration

Name	Explanation	Value
macros	Use of these macros will be reported.	('DEBUG', 'LOG')

Possible Messages

Name	Message
debug_macro_use	Macro must not be used.

Generic-NoDiamondInheritance

Do not introduce a non-virtual diamond inheritance.

Input: IR

Source languages: C++

Details

Non-virtual diamond inheritance causes each object of the derived class to contain multiple copies of the base class. This can lead to confusing behavior, especially if the base class contains mutable state.

Avoid diamond inheritance when possible. If the derived class is intended to have multiple copies of the base class, prefer composition over inheritance. If the derived class should have only one copy of the base class, use virtual inheritance.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
nonvirtual_diamond_inheritance	Use of non-virtual diamond inheritance.

Generic-NoDiscardedReturnCode

Do not ignore return values of functions.

Input: IR

Source languages: C, C++, C#

Details

Most C functions return error codes. Ignoring the return value of those functions will silently ignore errors. This can cause incorrect behavior as the program continues despite an error.

This rule checks that integer return values are used and not ignored. To discard a return value, cast the result of the function call to `void`.

Example

```
// Propagate error:  
if (fseek(f, 0, SEEK_SET) != 0)  
{  
    return ERR_CANNOT_SEEK;  
}  
  
// Ignore errors when closing the file:  
(void)fclose(f);
```

Configuration

Name	Explanation	Value
error_types	List of user defined error type names (if empty, all ignored int values are reported).	[]
inspect_template_instances	Whether calls in template instances should be reported.	False
whitelist	Dictionary of header globbing to [list of] function names whose return codes can be ignored.	dict(...)

Possible Messages

Name	Message
discarded_return	Return value of function discarded.

Generic-NoDoubleUnderscoreInMacro

Do not use more than one consecutive underscore in a macro.

Input: IR

Source languages: C, C++

Details

This rule prevents the use of multiple consecutive underscores within macro names.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
double_underscore_in_macro	Macro definition with consecutive underscores.
macro_starting_with_double_underscore	Macro name starting with consecutive underscores.

Generic-NoEllipsis

Do not use variable arguments in functions.

Input: IR

Source languages: C, C++

Details

This rule prevents the definition of C-style variadic functions. The use of strongly-typed means of parameter passing is preferable. In C++11, consider the use of parameter packs.

Calls to existing functions with variable arguments (such as `printf`) is allowed by this rule.

Rationale

Variable arguments are difficult to use correctly: if the number and types of the arguments passed by the caller does not match those expected by the callee, undefined behavior will occur.

Configuration

Name	Explanation	Value
ignore_declarations	Allow declarations of functions with variable number of arguments (e.g. for existing library functions); only report definitions.	True
ignore_inherited	Do not report functions inheriting a variable number of arguments.	False

Possible Messages

Name	Message
ellipsis_parameter	Function must not have variable number of arguments.

Generic-NoEmptyLoops

Loops must not be empty.

Input: IR

Source languages: C, C++, C#

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
empty_loop	Loops must not be empty.

Generic-NoEmptyStructs

Structs/unions must not be empty.

Input: IR

Source languages: C

Details

Standard C requires structs and unions to contain at least one named member. This restriction does not apply in C++, nor in C with GNU extensions. However, these languages treat empty structs differently: `sizeof(struct EmptyStruct { })` is 1 in C++, but 0 in GNU C. To avoid portability problems, empty structs should be avoided in C code.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
empty_struct	Empty struct has undefined behavior
empty_union	Empty union has undefined behavior
struct_without_named	struct without named members has undefined behavior
union_without_named	union without named members has undefined behavior

Generic-NoExternInImpl

Do not put extern declarations into implementation files.

Input: IR

Source languages: C, C++

Details

Any objects/functions with extern linkage should be declared in header files. Objects/functions in implementation files should use static linkage.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
extern_funcdef_in_implementation	Extern declaration in implementation file.
extern_variable_decl_in_implementation	Extern declaration in implementation file.

Generic-NoFriendClass

Do not declare any friend classes/unions/structs.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
friend_class	Do not use friend class/struct/union declarations.

Generic-NoFunctionCommentInImpl

Do not place comments above functions and methods in implementation files.

Input: IR

Source languages: C, C++

Details

Function documentation should be placed above the function declaration in the header file. The documentation should not be duplicated in the implementation file.

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
comment_in_funcimpl	Use of function/method comment in implementation file.

Generic-NoFunctionDefinitionInHeader

Do not define (non-inline) functions in header files.

Input: IR

Source languages: C, C++

Details

The declaration of a non-inline function in a header file violates the One Definition Rule if the header file is included in multiple compilation units.

Configuration

Name	Explanation	Value
find_non_external_function_declaration	Whether function declarations with non-external linkage should be reported.	False

Possible Messages

Name	Message
function_definition_in_header	Function definition in header file.

Generic-NoFunctionMacroInvocation

Function-like macros shall not be called.

Input: IR

Source languages: C, C++

Details

Avoid using function-like macros. Inline functions should be preferred over macros.

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
function_macro_invocation	Function-like macros shall not be called

Generic-NoFunctionPrototypeInImpl

Do not declare function prototypes in implementation files.

Input: IR

Source languages: C, C++

Details

Declarations should be placed in header files, not in implementation files.

Configuration

Name	Explanation	Value
accept_method_declarations	Whether prototypes of methods inside their class should be allowed, e.g. when the Pimpl-idiom is used.	False
accept_static_functions	Whether prototypes of static functions should be allowed.	False

Possible Messages

Name	Message
function_prototype_in_implementation	Function prototype in implementation file.

Generic-NoIfdefInHeader

Except for include guards there shall be no #if[n]def or #if in header files.

Input: IR

Source languages: C, C++

Details

Rationale

The use of `#ifdef` in header files is problematic, as it can make the preprocessing of the header file depend on the order in which headers are included. This in turn may lead to violations of the One Definition Rule.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
pp_if_in_header	Except for include guards there shall be no #if[n]def or #if in header files.

Generic-NoImplicitTypeConversion

Do not use implicit type conversions.

Input: IR

Source languages: C, C++

Details

This rule will warn on every use of implicit type conversions. C performs implicit type conversions in many places, e.g. numeric promotion whenever an operator is applied to a type smaller than `int`. This makes it nearly impossible to avoid all implicit conversions, so this rule is not very useful. Consider using the MisraC2012 rules 10.* and 11.* instead to prevent problematic implicit type conversions.

See Also

[Generic-NoPrecisionLoss](#)

Configuration

Name	Explanation	Value
excluded_classes	Types which are ignored both as source and target type after stripping pointers, qualifiers and typedefs.	{'Unknown_Type', 'Unknown_Class_Type'}

Possible Messages

Name	Message
suspicious_implicit_conversion	Use of implicit type conversion.

Generic-NoIncludePaths

Avoid using forbidden kinds of paths in #include

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_absolute_paths	Suppresses messages on absolute paths in #includes	False
allow_downward_paths	Suppresses messages on #include "a/b/c" cases	False
allow_same_directory	Suppresses messages on #include "./file" cases	False
allow_upward_paths	Suppresses messages on #include "../a" cases	False
use_scanner	Scan files of the project instead of relying on IR data.	True

Possible Messages

Name	Message
include_with_forbidden_path_kind	Avoid using forbidden kinds of paths in #include.

Generic-NoIrregularInclude

Avoid #includes which cannot be syntactically integrated.

Input: IR

Source languages: C, C++

Details

An #include cannot be syntactically integrated if it contains unbalanced #ifdef or braces, or otherwise incomplete syntax constructs.

Irregular includes should be avoided, as they might cause surprising behavior when the incomplete syntax construct is parsed in combination with the neighboring header file or the implementation file.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
irregular_include	irregular #include

Generic-NoLeakingReferenceToLocal

Do not assign to some nonlocal object a reference or pointer to a local variable.

Input: IR

Source languages: C, C++

Details

This rule will detect when a reference or pointer to a local variable is assigned to a non-local object. Because the non-local object will usually outlive the

local variable, the reference or pointer will be dangling, and any access will cause a use-after-free error.

See Also

[Rule NoReferenceToLocalVariable](#)

Configuration

Name	Explanation	Value
allow_longer_living_local	Whether assignment to a longer-living local variable should be accepted.	True

Possible Messages

Name	Message
possibly_leaking_reference_to_local_variable	Potentially leaking reference/pointer to local variable.

Generic-NoLinkerWarnings

Fix all linker warnings and errors.

Input: IR

Source languages: C, C++

Details

Linker warnings indicate problems with the source code and should thus be taken seriously.

Configuration

Name	Explanation	Value
reported_messages	If provided, only messages of these types are reported.	None
reported_severities	List of severities to display.	{'error', 'warning'}
use_error_number	Whether the error number from the frontend should be used.	True
use_rule_severity	Whether the rule's severity or the linker's severity should be used.	False

Generic-NoMagicNumbers

Do not use magic literals.

Input: IR

Source languages: C, C++, C#

Details

When source code contains a magic number literal, later readers of the code may have trouble determining the origin and meaning of that number.

Use a constant definition to assign a name to the number.

Example

```
// AVOID:  
for (int i = 0; i < 52; i++) ...  
  
// GOOD:  
const int cardGame_deckSize = 52;  
for (int i = 0; i < cardGame_deckSize; i++) ...
```

Configuration

Name	Explanation	Value
allow_nonconst_variable_initialization	If set to `True`, allow the initialization of a non-const variable with a string literal in addition to the initialization of a const variable. A reassignment of such a variable is still a violation.	False
allowed	Literal values that are ok.	{0.0, 1.0, 2.0}
allowed_contexts	Optional set of PIR classes for allowed contexts, e.g. Case_Label.	set[])
exceptions	Optional predicate to filter out cases that should be allowed as an exception. The predicate takes a literal node and returns True if the given node is such an exception.	None
exclude_pp_literals	If True, literals in conditions of #if are ignored.	True
exclude_single_uses	If True, report only literals used more than once.	False

Possible Messages

Name	Message
magic_number	Use of magic literal.
magic_number_without_token	Use of magic literal.
possible_magic_number	Potential use of magic literal.

Generic-NoMalloc

The library functions "malloc", "calloc", "realloc", and "free" from library <stdlib.h> shall not be used.

Input: IR

Source languages: C, C++

Details

In C++, memory should be managed automatically using classes such as std::unique_ptr, std::shared_ptr and std::vector.

Configuration

Name	Explanation	Value
allow_in_user_new_delete_operator	Whether to allow the library functions inside user defined new/delete operator overloads.	False

Possible Messages

Name	Message
stdlib_memory_function_call	Dynamic memory management function from <stdlib.h> is called.

Generic-NoMixOfPtrAndIntArithmetic

Do not mix pointer and integer arithmetic.

Input: IR

Source languages: C, C++

Details

This rule will warn when pointers are converted to integers.

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes, NullPointerTypes], [SignedTypes, UnsignedTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	True
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion of pointer type to integer type.

Generic-NoMultipleInheritance

Do not use multiple inheritance.

Input: IR

Source languages: C++

Details

Multiple inheritance can make code hard to understand and should be avoided. Compositions should be preferred over inheritance whenever possible.

Configuration

Name	Explanation	Value
ignore_pure_interfaces	Whether C++ interfaces are allowed, i.e. classes with only pure virtual members.	False

Possible Messages

Name	Message
multiple_inheritance	Use of multiple inheritance.

Generic-NoNewWithArrays

Do not use new with arrays.

Input: IR

Source languages: C++

Details

The use of `new` with arrays should be avoided, as there is a lot of potential for error:

- Buffer overflows are more likely as the size of the array needs to be handled separately from the array
- Operator `delete` will not properly free the array, and operator `delete[]` needs to be used instead.

Instead, use `std::vector` to manage heap-allocated arrays.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
new_on_array	Use of new operator on array type.

Generic-NoOverloadedOperators

Do not overload certain operators.

Input: IR

Source languages: C++

Details

Programmers expect `operator &&` and `operator ||` to have short-circuiting behavior. As it is impossible for user-defined overloads of these operators to have the expected behavior, these operators should never be overloaded.

The built-in `operator ,` introduces a sequence point between the left and right operands. However, user-defined operator overloads act like functions, and thus do not have a sequence point between their arguments. Overloaded `operator ,` thus cannot match programmer expectations, and should be avoided.

Configuration

Name	Explanation	Value
invalid	Selection of disallowed operator overloads by name.	<code>['operator&&', 'operator ', 'operator,']</code>
invalid_kinds	Selection of disallowed operator overloads by operator kind.	[7]

Possible Messages

Name	Message
forbidden_operator_overload	Do not overload operator.

Generic-NoPrecisionLoss

Do not assign longer data types to shorter ones.

Input: IR

Source languages: C, C++

Details

This rule will warn on implicit conversions from larger types to shorter types. Use an explicit conversion to make it clear that the conversion may cause loss of precision.

Example

```
short int_to_short(int i)
{
    return i; // BAD
    return (short)i; // OK
}
```

Configuration

Name	Explanation	Value
message_anchor	Allows to choose between 'lhs', 'rhs', and 'asgn' (default) node as anchor where the message should be reported, distinguished by PIR node type of the assignment kind.	dict[...]

Possible Messages

Name	Message
precision_loss	{ } with loss of precision.

Generic-NoPublicDataMembers

Do not declare non-const data members public.

Input: IR

Source languages: C, C++, C#

Details

Public data members could be modified by other types in a way that violates the invariants of the class. Data members should be encapsulated: they should be declared as private, and accessor functions should be declared to allow other classes access to the data members.

Example

```
class C
{
private:
    std::string m_member;

public:
    const std::string& member()
    {
        return m_member;
    }
    void member(const std::string& new_value)
    {
        assert(new_value.size() > 0);
        m_member = new_value;
    }
};
```

See Also

[Rule NoReferenceToDataMember](#)

Configuration

Name	Explanation	Value
allow_protected_members	Whether protected fields should be ignored.	True
allowed	Specifies allowed fields as pairs [class name pattern, field name pattern]. Example: [re.compile('.*'), re.compile('x')] to allow x in all classes.	[]
ignore_const_members	Whether public const fields should be ignored.	True
ignore_pod	Whether public fields in POD classes should be ignored.	True
ignore_structs	Whether public fields in structs should be ignored.	True
ignore_templates	Whether public fields in generic templates should be ignored.	False

Possible Messages

Name	Message
protected_field	Protected non-const data member.
public_field	Public non-const data member.

Generic-NoReferenceToLocalVariable

Do not return a reference or pointer to a local variable.

Input: IR

Source languages: C, C++

Details

This rule will detect when a reference or pointer to a local variable is returned from a function. Such a reference or pointer will be dangling, and any access will cause a use-after-free error.

See Also

[Rule NoLeakingReferenceToLocal](#)

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
global_reference_to_local_var	Address of local variable escapes via global variable.
parameter_reference_to_local_var	Address of local variable escapes via parameter.
returning_reference_to_local_var	Return of reference(pointer) to local variable.

Generic-NoReferenceToPrivateDataMember

Do not return non-const references to private data members.

Input: IR

Source languages: C++

Details

When a non-const reference to a private data member is handed out to code outside the class, that code could modify the data member in ways that violates the invariants of the class. Data members should be encapsulated: they should be declared as private, and accessor functions should be declared to allow other classes access to the data members.

Example

```
class C
{
private:
    std::string m_member;

public:
    const std::string& member()
    {
        return m_member;
    }
    void member(const std::string& new_value)
    {
        assert(new_value.size() > 0);
        m_member = new_value;
    }
};
```

See Also

[Rule NoPublicDataMembers](#)

Configuration

Name	Explanation	Value
include_copies_of_pointer_fields	Whether returning a pointer field should be reported as well.	False
inspect_only_const_methods	Whether all methods or only const methods should be checked.	False
only_report_references	Whether pointer and reference to field should be reported, or just references.	False
smart_pointer_names	Names that are considered smart pointers.	['unique_ptr', 'shared_ptr', 'weak_ptr']

Possible Messages

Name	Message
returning_nonconst_member_reference	Return of reference to private non-const data member.

Generic-NoSemicolonAtEndOfMacro

Macro replacement must not end with semicolon.

Input: IR

Source languages: C, C++

Details

Do not conclude macro definitions with a semicolon.

```
#define FOR_LOOP(n)  for(i=0; i<(n); i++);
```

Instead use

```
#define FOR_LOOP(n)  for(i=0; i<(n); i++)
```

Semicolons at the end of macro definitions may cause unexpected program behavior.

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
semicolon_at_end_of_macro	Macro replacement must not end with semicolon.

Generic-NoSignedDivision

Do not use signed variables in divisions.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
signed_division	Division uses signed variable.

Generic-NoSingleCharIdentifier

Do not use identifiers consisting of just a single character.

Input: IR

Source languages: C, C++, C#

Configuration

Name	Explanation	Value
exclude_catch	If True, identifiers to name the exception in catch handlers are tolerated.	False
exclude_local_nonstatic_variables	If True, local non-static variables are tolerated.	False
exclude_local_static_variables	If True, local static variables are tolerated.	False
exclude_loop_counter	If True, for-loop counters are tolerated.	False
whitelist	Allowed single-character names.	[]

Possible Messages

Name	Message
single_char_identifier	Identifier is single-character identifier.

Generic-NoStaticInHeader

Do not put static declarations into header files.

Input: IR

Source languages: C, C++

Details

Static declarations should only be used in implementation files for objects/functions that are used locally within that implementation file.

Functions declared in header files should have external linkage, and should be defined in the corresponding implementation file.

Configuration

Name	Explanation	Value
allow_static_const_in_cpp	If True, const globals in C++ (which are implicitly static) are tolerated.	True
allow_static_inline	Whether static inline functions are allowed in header files.	True

Possible Messages

Name	Message
static_function_in_header	Static declaration in header file.
static_variable_in_header	Static declaration in header file.

Generic-NoStdStringInternals

Do not rely on std::string internals when converting to char*.

Input: IR

Source languages: C++

Details

The operator[] of std::string should not be used to retrieve a reference or pointer into the string contents.

Instead, use the c_str() or data() member functions.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
cpp_string_internals	Use of std::string operator[], use c_str() instead.

Generic-NoTabs

Do not use tabs.

Input: IR

Source languages: C, C++, C#

Configuration

Name	Explanation	Value
per_file	Flag which combines all violations in a file to one violation where individual findings are handled as additional entities.	False

Possible Messages

Name	Message
tab_character	Do not use tabs.

Generic-NoTrailingWhitespace

Do not use trailing whitespace.

Input: IR

Source languages: C, C++, C#

Configuration

Name	Explanation	Value
allow_in_comment	Whether trailing whitespace in comments should be allowed.	False

Possible Messages

Name	Message
trailing_whitespace	Do not use trailing whitespace.

Generic-NoTypeConversionToBool

Do not use type conversions to bool, use value comparison.

Input: IR

Source languages: C, C++

Details

Instead of using implicit conversions to bool, make the comparison explicit:

```
void (int i, float* pf)
{
    if (i != 0)
    {
    }
    if (pf != NULL)
    {
    }
}
```

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, FloatingTypes, CharacterTypes, BoolTypes, EnumTypes, VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes, NullPointerTypes, OtherTypes, VoidTypes], [BoolTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	True
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion to bool - use value comparison instead.

Generic-NoUncheckedMalloc

The result of calls to the library functions "malloc", "calloc", and "realloc" must be checked.

Input: IR

Source languages: C, C++

Details

Memory allocations can fail due to out-of-memory conditions. You should check for a `NULL` result immediately after calling a memory allocation function to avoid undefined behavior when running out of memory.

Example

```
S* ps = (S*)malloc(sizeof(S));
if (ps == NULL)
{
    return ERR_OUT_OF_MEMORY;
}
```

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
unchecked_malloc	Result of call to malloc, calloc, or realloc is not checked.

Generic-NoUncheckedPointerParamDereference

Check pointer parameters for NULL before dereferencing them.

Input: IR

Source languages: C, C++, C#

Details

To avoid undefined behavior when a `NULL` pointers is passed into a function, all functions should check their parameters for null pointers before dereferencing them.

Configuration

Name	Explanation	Value
check_all_funcs	If False, check only non-static nonmember functions and public member functions.	False
null_check_macro	Name of macro used to represent check for NULL	
null_check_routines	Qualified names of functions used to realize a null check	[]

Possible Messages

Name	Message
unchecked_param_dereference	Check pointer parameter for NULL before dereferencing it.

Generic-NoUnnamedNamespaceInHeader

Do not use unnamed namespaces in header files.

Input: IR

Source languages: C++

Details

The contents of anonymous namespaces are only visible to the current compilation unit.

See Also

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
unnamed_namespace_in_header	Use of unnamed namespace in header file.

Generic-NoUnsafeMacro

Do not use macro definitions with parameters or expressions without parentheses around them.

Input: IR

Source languages: C, C++

Details

Macros that involve operators must be fully parenthesized:

```
#define MUL(A, B) ((A) * (B))
```

Macros that lack parentheses may parse in unexpected ways when used in certain contexts.

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
unsafe_macro_body	Macro replacement text potentially must be parenthesized.
unsafe_macro_param	Macro argument(s) must be parenthesized.

Generic-NoUsingNamespacelnHeader

Do not use 'using namespace' in a header file.

Input: IR

Source languages: C++

Details

`using namespace` in headers can unexpectedly change the meaning of identifiers in other code files that import that header.

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
using_namespace_in_header	Use of "using namespace" in header file.

Generic-NoVirtualDestructor

Class needs a virtual destructor if at least one member function is virtual.

Input: IR

Source languages: C++

Details

Classes that are used polymorphically should have a virtual destructor.

If an object is deleted through a pointer to a base class and that base class does not have a virtual destructor, undefined behavior occurs.

Example

```
class C {
    C() {} // explicit constructor, may take arguments
    C(const C&) = delete;
    C& operator=(const C&) = delete;
    virtual ~C() { }
};
```

Configuration

Name	Explanation	Value
accept_none_destructor	If set true, classes without any destructor are tolerated.	True

Possible Messages

Name	Message
missing_virtual_destructor	Class needs a virtual destructor.

Generic-NoVirtualInheritance

Do not use virtual inheritance.

Input: IR
Source languages: C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
virtual_inheritance	Use of virtual inheritance.

Generic-NoWhitespaceMemberSelection

No whitespace before or after a member selection.

Input: IR
Source languages: C, C++, C#

Details

The should be no whitespace around the following operators: ., ->, .* and ->*

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
whitespace_member_selection	Whitespace around member selection.

Generic-NoWhitespaceUnaryOperator

No whitespace after an unary operator.

Input: IR
Source languages: C, C++, C#

Details

There should be no whitespace after the unary operators ~, !, + and -.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
whitespace_after_unary	Whitespace after unary operator.

Generic-PCHIncludes

Add #includes for often-used files to precompiled header

Input: IR
Source languages: C, C++

Details

Headers that are included by more than the configured numbers of compilation units should be added to the precompiled header (PCH). This can improve compilation time.

Configuration

Name	Explanation	Value
min_pch_clients_for_include	Adding a file to the PCH is only recommended if there are at least this many units using the file.	4

Possible Messages

Name	Message
add_file_to_pch	add file to PCH {} as it is used by {} clients

Generic-RuleOfThree

Certain special functions must always be declared together (e.g. copy constructor and copy assignment operator).

Input: IR

Source languages: C++

Details

If only some of the *special member functions* are explicitly declared, the compiler will generate the remaining special member functions. The compiler-generated implementations of these functions likely are inconsistent with the explicitly declared functions, which leads to resource management bugs such as dangling references and double frees.

In most cases, you should follow the **Rule of Zero**: resource management should be encapsulated in classes that do nothing except managing a single resource. Often, the classes `std::unique_ptr` or `std::shared_ptr` can be used for this purpose (with custom deleters). This frees up all other classes from the task of resource management: those classes should not define any *special member functions*, and rely on the compiler-generated default implementations instead.

When implementing a class that manages resources, follow the **Rule of Three** or **Rule of Five**:

- Explicitly declare the *copy constructor*, *copy assignment operator* and *destructor* (copyable class without move semantics); or
- Explicitly declare the *move constructor*, *move assignment operator* and *destructor* (non-copyable class with move semantics); or
- Explicitly declare the *copy constructor*, *copy assignment operator*, *move constructor*, *move assignment operator* and *destructor* (copyable class with move semantics)

A class that is neither copyable nor movable still needs to follow these rules, but should mark the *copy constructor* and *copy assignment operator* as deleted.

Exception

A destructor with an empty body (or one defined as '= defaulted;') does not require declaring any other special member functions. Empty destructors are functionally identical to the default destructor (which calls all member destructors), but may need to be defined explicitly if the member destructor can only be called in certain compilation units (e.g. `std::unique_ptr<Incomplete_Type>`).

References

- [Rule of Three](#)
- [Rule of Zero](#)

Configuration

Name	Explanation	Value
allow_defaulted_destructor_only	Allow classes with a defaulted destructor and no other special member functions.	True
allow_destructor_only	Allow all destructors without copy or move constructors	False
allow_empty_destructor	Allow empty destructors without copy or move constructors.	True
allow_missing_destructor	Suppress messages about missing destructors.	False
ignore_pod_classes	Whether POD classes should be checked at all	True

Possible Messages

Name	Message
missing_constructor_and_asgn	Class with destructor should also declare a copy or move constructor and assignment operator.
missing_copy_asgn	Class with copy constructor is missing copy assignment operator.
missing_copy_constructor	Class with copy assignment operator is missing copy constructor.
missing_destructor	Class with copy or move constructors or assignment operators should also declare a destructor.
missing_move_asgn	Class with move constructor is missing move assignment operator.
missing_move_constructor	Class with move assignment operator is missing move constructor.

Generic-Templates

List templates with instances.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
include_dependent	If true, includes nonreal instances inside other templates.	False
only_toplevel	If true, inspect only templates in global or namespace scope.	True
show_instances	If true, prints all instances per template.	False
show_uninstantiated	If true, includes templates with no instances.	True

Possible Messages

Name	Message
template_ast_weight	{}, AST weight: {}
template_instances	{} {}, AST weight: {}
template_without_instance	0

Generic-ThrowByValueCatchByReference

Throw exceptions by value and catch them by reference.

Input: IR

Source languages: C++

Details

Do not use `new` when throwing exceptions, as it is unclear who would have the responsibility for freeing the exception object. Instead, throw exceptions by value.

Catching exceptions by value will lead to object slicing when the thrown exception inherits from the exception type specific in the catch clause. Catch exceptions by reference to allow for polymorphism.

Example

```
try
{
    throw MyException(args);
}
catch (const MyException& ex)
{
    ...
}
```

Configuration

Name	Explanation	Value
only_class_types	Whether all types should be caught by reference or only class types.	False

Possible Messages

Name	Message
catch_without_reference	Exceptions shall be caught by reference.
throwing_pointer	Exceptions shall be thrown by value.

Generic-TooManyIncludes

Report units with high number of included files

Input: IR

Source languages: C, C++

Details

This rule will report compilation units that include more than the configured number of files.

Configuration

Name	Explanation	Value
allowed	Maximum permissible number of included files	400
count_indirect_system_includes	Whether system-includes from inside system headers should be counted.	True
show_value	Select whether message should indicate the current value and limit.	False

Possible Messages

Name	Message
too_many_includes_in_unit	unit includes more than {} files.
too_many_includes_with_value	unit includes {} files which is more than the limit of {} files.

Generic-TypedefCheck

Do not redefine a typedef.

Input: IR

Source languages: C, C++

Details

This rule will warn when there are multiple definitions for a typedef. Usually, each typedef should be defined only a single time in a header file.

Note that if multiple definitions of a typedefs exist, all those definitions must be equivalent to avoid violating the One Definition Rule (which causes undefined behavior).

Configuration

Name	Explanation	Value
report_identical_redefinitions	Enables messages for identical redefinitions of a typedef.	True
report_template_instance_redefinitions	Enables messages for identical redefinitions of a typedef naming a template instance.	True

Possible Messages

Name	Message
differing_typedef_redefinition	Typedef redefined with different definition.
identical_typedef_redefinition	Typedef redefined.

Generic-WrongIncludeCasing

Includes should use same casing as target file name.

Input: IR

Source languages: C, C++

Details

When porting from Windows / case-insensitive file system to case-sensitive file systems, the casing has to be identical. Even if you do not plan to port your code, the same casing will look more familiar and cause less surprises.

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
wrong_include_casing	#include should use casing of target file (which is {}).

Rules in Group Metric

Metric-Calling.CPP

Input: RFG

Details

Counts for a given routine r the number of distinct routines that call r . Calls from routines that are instances of the same template are not distinguished (except for partial specializations).

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
consider_dynamic_calls	Also count incoming dynamic calls (generated by iranalysis).	True
display_name	Description of the metric shown in the dashboard.	Number of functions calling this function (counting calls from template instances only once)
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	5
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Calling.CPP

Metric-Clone_Ratio

Clone Ratio.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Clone Ratio
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	25
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Clone_Ratio

Metric-Comment.Density

Comment Density.

Input: RFG

Details

Computes the comment density (also called comment frequency) of a routine, which roughly corresponds to the ratio between comments before, within, and on the same line as the routine and the number of statements of the routine. For a given routine r , let $st(r)$ be the number of statements of the routine. Let $p(r)$ be the number of comments before the routine and on the same line as the routine definition, i.e., the value Metric.Comment.Preceding, and let $i(r)$ be the number of comments within the routine, i.e., the value Metric.Comment.Internal. Then the comment density $d(r)$ of r is computed as follows:

```
comments(r) = p(r) + i(r)
if (st(r) > 0) {
    d(r) = comments(r) / st(r)
} else if (comments(r) > 0) {
    d(r) = MAX_INT
} else {
    d(r) = 0
}
```

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Comment Density
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	0.2
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Comment.Density

Metric-Comment.Internal

Internal Comments.

Input: RFG

Details

Counts the number of comments within the body of a routine.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Internal Comments
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Comment.Internal

Metric-Comment.Preceding

Preceding Comments.

Input: RFG

Details

Counts the number of comments directly before the routine header that do not share a line with another non-comment code construct.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Preceding Comments
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	0.2
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Comment.Preceding

Metric-Coupling

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
coupling_edge_name		Belongs_To
display_name	Description of the metric shown in the dashboard.	None
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Coupling

Metric-Extended_M McCabe

Extended Cyclomatic Complexity.

Input: RFG

Details

Computes the number of nodes in the control flow graph of a routine that have more than one outgoing edge.

For nodes originating in switch-statements, each nonempty case section is counted. It is also known as cyclomatic complexity.

Extended Cyclomatic Complexity differs from Metric.M McCabe_Complexity in its treatment of short circuit logical operators: it also interprets these operators as resulting in control flow nodes having more than one outgoing edge and therefore adds also the number of shortcut operators (&& and ||) to the value.

Roughly speaking, the value Extended Cyclomatic Complexity is computed by summing up

- the number of if-statements,
- the number of loop statements (e.g. for and while),
- the number of conditional (?) operators,
- the number of catch clauses,
- the number of nonempty case-sections, and
- the number of shortcut operators
- plus the value one.

So Extended Cyclomatic Complexity is always greater zero.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Extended Cyclomatic Complexity
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Extended_M McCabe

Metric-Fan.In

Number of incoming edges of certain configurable edge types.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Used internal. DO NOT USE IT - use edge_types['edge-descriptor-name']['display_name'] instead.	None
edge_types	Dictionary string --> {'display_name' --> string, 'min_value' --> float, 'max_value' --> float, 'disable' --> boolean}. * The key is the name of the edge_descriptor_type. * display_name: description of the metric shown in dashboard. * min_value: minimal number of allowed incoming edges. * max_value: maximal number of allowed incoming edges. * disable: disables this edge type (optional).	dict{...}
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Used internal. DO NOT USE IT - use edge_types['edge-descriptor-name']['max_value'] instead.	None
min_value	Used internal. DO NOT USE IT - use edge_types['edge-descriptor-name']['min_value'] instead.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Used internal. DO NOT USE IT.	

Metric-Fan.Out

Number of outgoing edges of certain configurable edge types.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Used internal. DO NOT USE IT - use edge_types['edge-descriptor-name']['display_name'] instead.	None
edge_types	Dictionary string --> {'display_name' --> string, 'min_value' --> float, 'max_value' --> float, 'disable' --> boolean}. * The key is the name of the edge_descriptor_type. * display_name: description of the metric shown in dashboard. * min_value: minimal number of allowed incoming edges. * max_value: maximal number of allowed incoming edges. * disable: disables this edge type (optional).	dict(...)
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Used internal. DO NOT USE IT - use edge_types['edge-descriptor-name']['max_value'] instead.	None
min_value	Used internal. DO NOT USE IT - use edge_types['edge-descriptor-name']['min_value'] instead.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Used internal. DO NOT USE IT.	

Metric-HIS.AP(CG)_CYCLE

HIS ap_cg_cycle.

Input: RFG

Details

This metric counts the number of call-graph recursions.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
alias_for	The name of the RFG node attribute of the original metric. This name is searched in the RFG. Allowed values: Metric.Number_Of_SCs 	Metric.Number_Of_SCs
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	HIS.ap_cg_cycle
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	0
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.HIS.AP(CG)_CYCLE

Metric-HIS.CALLING

HIS CALLING.

Input: RFG

Details

Counts for a given routine r the number of distinct routines that call r .

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
alias_for	The name of the RFG node attribute of the original metric. This name is searched in the RFG. Allowed values: Metric.Number_Of_Calling_Routines 	Metric.Number_Of_Calling_Routines
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	HIS CALLING
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	5
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.HIS.CALLING

Metric-HIS.CALLS

HIS CALLS.

Input: RFG

Details

Counts for a given routine r the number of distinct routines that r calls.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
alias_for	The name of the RFG node attribute of the original metric. This name is searched in the RFG. Allowed values: Metric.Number_Of_Called_Routines 	Metric.Number_Of_Called_Routines
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	HIS CALLS
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	7
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.HIS.CALLS

Metric-HIS.COMF

HIS COMF.

Input: RFG

Details

Computes the comment density (also called comment frequency) of a routine, which roughly corresponds to the ratio between comments before, within, and on the same line as the routine and the number of statements of the routine. For a given routine r , let $st(r)$ be the number of statements of the routine. Let $p(r)$ be the number of comments before the routine and on the same line as the routine definition, i.e., the value Metric.Comment.Preceding, and let $i(r)$ be the number of comments within the routine, i.e., the value Metric.Comment.Internal. Then the comment density $d(r)$ of r is computed as follows:

```
comments(r) = p(r) + i(r)
if (st(r) > 0) {
    d(r) = comments(r) / st(r)
} else if (comments(r) > 0) {
    d(r) = MAX_INT
} else {
    d(r) = 0
}
```

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
alias_for	The name of the RFG node attribute of the original metric. This name is searched in the RFG. Allowed values: Metric.Comment.Density 	Metric.Comment.Density
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	HIS COMF
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_loc	Minimal LOC threshold that a function must at least cross in order to be considered. If None, all functions are considered.	None
min_value	Minimal allowed value. None if unlimited.	0.2
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.HIS.COMF

Metric-HIS.GOTO

HIS GOTO.

Input: RFG

Details

Count of goto-statements in routine.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
alias_for	The name of the RFG node attribute of the original metric. This name is searched in the RFG. Allowed values: Metric.Number_Of_Gotos 	Metric.Number_Of_Gotos
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	HIS GOTO
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	0
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.HIS.GOTO

Metric-HIS.LEVEL

HIS LEVEL.

Input: RFG

Details

Counts the maximum nesting level of syntactic levels inside a routine.

HIS LEVEL differs from Metric.Maximum_Extended_Nesting by its treatment of else if-statements: HIS LEVEL does not increase the nesting level by one for each else if-statement.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
alias_for	The name of the RFG node attribute of the original metric. This name is searched in the RFG. Allowed values: Metric.Maximum_Nesting Metric.Maximum_Extended_Nesting 	Metric.Maximum_Nesting
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	HIS LEVEL
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	4
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.HIS.LEVEL

Metric-HIS.PARAM

HIS PARAM.

Input: RFG

Details

This metric counts the number of parameters of a function/method as to give an indication for the size of the interface.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
alias_for	The name of the RFG node attribute of the original metric. This name is searched in the RFG. Allowed values: Metric.Number_Of_Parameters 	Metric.Number_Of_Parameters
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	HIS PARAM
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	5
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.HIS.PARAM

Metric-HIS.PATH

HIS PATH.

Input: RFG

Details

Computes the number of maximal acyclic execution paths within a function. The rules for each syntactical construct can be found in "NPATH: a measure of execution path complexity and its applications" (Brian A. Nejmeh, Communications of the ACM, Volume 31 Issue 2, Feb. 1988).

The values of this metric can be very high and will be cut off at $2^{31} - 1$ (the highest possible value for any integer metric in the RFG).

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
alias_for	The name of the RFG node attribute of the original metric. This name is searched in the RFG. Allowed values: Metric.NPath: Number of acyclic paths Metric.LogNPath_Ceiling: upper bound for log2(Number of acyclic paths) Metric.LogNPath_Floor: lower bound for log2(Number of acyclic paths) 	Metric.NPath
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	HIS PATH
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	80
min_value	Minimal allowed value. None if unlimited.	1
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.HIS.PATH

Metric-HIS.RETURN

HIS RETURN.

Input: RFG

Details

Count of return statements in routine.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
alias_for	The name of the RFG node attribute of the original metric. This name is searched in the RFG. Allowed values: Metric.Number_Of_Returns 	Metric.Number_Of_Returns
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	HIS RETURN
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	1
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.HIS.RETURN

Metric-HIS.STMT

HIS STMT.

Input: RFG

Details

Count of all statements in routine.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
alias_for	The name of the RFG node attribute of the original metric. This name is searched in the RFG. Allowed values: Metric.Number_Of_Statements 	Metric.Number_Of_Statements
allow_empty_constructor	Do not treat an empty constructor as violation.	False
allow_empty_destructor	Do not treat an empty destructor as violation.	False
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	HIS STMT
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	1
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.HIS.STMT

Metric-HIS.VG

HIS v(G).

Input: RFG

Details

Computes the number of nodes in the control flow graph of a routine that have more than one outgoing edge.

For nodes originating in switch-statements, each nonempty case section is counted. It is also known as cyclomatic complexity.

Roughly speaking, the value HIS v(G) is computed by summing up

- the number of if-statements,
- the number of loop statements (e.g. for and while),
- the number of conditional [?] operators,
- the number of catch clauses, and
- the number of nonempty case-sections
- plus the value one.

So HIS v(G) is always greater zero.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
alias_for	The name of the RFG node attribute of the original metric. This name is searched in the RFG. Allowed values: Metric.M McCabe_Complexity Metric.Extended_M McCabe 	Metric.M McCabe_Complexity
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	HIS v(G)
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	10
min_value	Minimal allowed value. None if unlimited.	1
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.HIS.VG

Metric-HIS.VOCF

HIS VOCF.

Input: RFG

Details

The language scope is an indicator of the cost of maintaining/changing functions.

VOCF = $\frac{N1 + N2}{n1 + n2}$, where

- n1 = Number of different operators
- N1 = Sum of all Operators
- n2 = Number of different Operands
- N2 = Sum of all Operands

For more details please refer to section L-3 of the Axivion Language Reference Guide.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
alias_for	The name of the RFG node attribute of the original metric. This name is searched in the RFG. Allowed values: Metric.Halstead.Vocabulary_Frequency 	Metric.Halstead.Vocabulary_Frequency
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	HIS VOCF
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	4
min_value	Minimal allowed value. None if unlimited.	1
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.HIS.VOFC

Metric-Halstead.Different_Operands

Different Halstead Operands.

Input: RFG

Details

This metric calculates the base value *different operands* of the Halstead metrics. The base values are:

- n1 = the number of different operators
- **n2 = the number of different operands**
- N1 = the total number of operators
- N2 = the total number of operands

For more details please refer to section L-3 of the Axivion Language Reference Guide.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Different Halstead Operands
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Halstead.Different_Operands

Metric-Halstead.Different_Operators

Different Halstead Operators.

Input: RFG

Details

This metric calculates the base value *different operators* of the Halstead metrics. The base values are:

- n1 = the number of different operators
- n2 = the number of different operands
- N1 = the total number of operators
- N2 = the total number of operands

For more details please refer to section L-3 of the Axivion Language Reference Guide.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Different Halstead Operators
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Halstead.Different_Operators

Metric-Halstead.Difficulty

Halstead Difficulty.

Input: RFG

Details

Let

- n1 = the number of different operators
- n2 = the number of different operands
- N1 = the total number of operators
- N2 = the total number of operands
- n = n1 + n2 (vocabulary)
-
- N = N1 + N2 (program length)
- V = N * log2(n) (volume)
- D = [n1/2]*[N2/n2] (difficulty)
- E = D * V (effort)

For more details please refer to section L-3 of the Axivion Language Reference Guide.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Halstead Difficulty
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Halstead.Difficulty

Metric-Halstead.Effort

Halstead Effort.

Input: RFG

Details

Let

- n1 = the number of different operators
- n2 = the number of different operands
- N1 = the total number of operators
- N2 = the total number of operands
- n = n1 + n2 (vocabulary)
-
- N = N1 + N2 (program length)
- V = N * log2(n) (volume)
- D = (n1/2)*(N2/n2) (difficulty)
- E = D * V (effort)

For more details please refer to section L-3 of the Axivion Language Reference Guide.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Halstead Effort
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Halstead.Effort

Metric-Halstead.Length

Halstead Length.

Input: RFG

Details

Let

- n1 = the number of different operators
- n2 = the number of different operands
- N1 = the total number of operators
- N2 = the total number of operands
- n = n1 + n2 (vocabulary)
-
- **N = N1 + N2 (program length)**
- V = N * log2(n) (volume)
- D = (n1/2)*(N2/n2) (difficulty)
- E = D * V (effort)

For more details please refer to section L-3 of the Axivion Language Reference Guide.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Halstead Length
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Halstead.Length

Metric-Halstead.Total_Operands

Total Halstead Operands.

Input: RFG

Details

This metric calculates the base value *total number of operands* of the Halstead metrics. The base values are:

- n1 = the number of different operators
- n2 = the number of different operands
- N1 = the total number of operators
- **N2 = the total number of operands**

For more details please refer to section L-3 of the Axivion Language Reference Guide.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Total Halstead Operands
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Halstead.Total_Operands

Metric-Halstead.Total_Operators

Total Halstead Operators.

Input: RFG

Details

This metric calculates the base value *total number of operators* of the Halstead metrics. The base values are:

- n1 = the number of different operators
- n2 = the number of different operands
- **N1 = the total number of operators**
- N2 = the total number of operands

For more details please refer to section L-3 of the Axivion Language Reference Guide.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Total Halstead Operators
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Halstead.Total_Operators

Metric-Halstead.Vocabulary

Halstead Vocabulary.

Input: RFG

Details

Let

- n1 = the number of different operators
- n2 = the number of different operands
- N1 = the total number of operators
- N2 = the total number of operands
- $n = n1 + n2$ (vocabulary)
-
- $N = N1 + N2$ (program length)
- $V = N * \log_2(n)$ (volume)
- $D = (n1/2) * (N2/n2)$ (difficulty)
- $E = D * V$ (effort)

For more details please refer to section L-3 of the Axivion Language Reference Guide.

Configuration

Name	Explanation	Value
aggregation_names	Display names [descriptions] of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Halstead Vocabulary
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values [leaf values] up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report [only store in RFG]. 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Halstead.Vocabulary

Metric-Halstead.Vocabulary_Frequency

Halstead Vocabulary Frequency.

Input: RFG

Details

The language scope is an indicator of the cost of maintaining/changing functions.

$$VOCF = \frac{N1 + N2}{n1 + n2}, \text{ where}$$

- n1 = Number of different operators
- N1 = Sum of all Operators
- n2 = Number of different Operands
- N2 = Sum of all Operands

For more details please refer to section L-3 of the Axivion Language Reference Guide.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Halstead Vocabulary Frequency
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Halstead.Vocabulary_Frequency

Metric-Halstead.Volume

Halstead Volume.

Input: RFG

Details

Let

- n1 = the number of different operators
- n2 = the number of different operands
- N1 = the total number of operators
- N2 = the total number of operands
- n = n1 + n2 (vocabulary)
-
- N = N1 + N2 (program length)
- V = N * log2(n) (volume)
- D = (n1/2)*(N2/n2) (difficulty)
- E = D * V (effort)

For more details please refer to section L-3 of the Axivion Language Reference Guide.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Halstead Volume
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Halstead.Volume

Metric-Includes.Direct_Includers

Number of directly including files.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Number of directly including files
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	25
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Includes.Direct_Includers

Metric-Includes.Direct_Includes

Number of directly included files.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Number of directly included files
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	10
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Includes.Direct_Includes

Metric-Includes.Include_Burden

Include burden caused by file.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Include burden caused by file
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	1000
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Includes.Include_Burden

Metric-Includes.Maximum_Include_Depth

Maximum Include Depth.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Maximum Include Depth
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	20
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Includes.Maximum_Include_Depth

Metric-Includes.Maximum_Includer_Depth

Maximum Includer Depth.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Maximum Includer Depth
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	20
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Includes.Maximum_Includer_Depth

Metric-Includes.Transitive_Includers

Number of transitively including files.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Number of transitively including files
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Includes.Transitive_Includers

Metric-Includes.Transitive_Includes

Number of transitively included files.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Number of transitively included files
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	20
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Includes.Transitive_Includes

Metric-Inverse_Coupling

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
coupling_edge_name		Belongs_To
display_name	Description of the metric shown in the dashboard.	None
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Inverse_Coupling

Metric-LOC

Lines of Code.

Input: RFG

Details

Counts lines of code of a function body.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines of Code
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	100
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.LOC

Metric-Lines.Class.Code

Lines in Class containing Code.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines in Class containing Code
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	600
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.Class.Code

Metric-Lines.Class.Comment

Lines in Class containing Comments.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines in Class containing Comments
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	600
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.Class.Comment

Metric-Lines.Class.Comment_Ratio

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	None
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.Class.Comment_Ratio

Metric-Lines.Class.LOC

Lines of Code in a Class (Empty, Comment, Code).

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines of Code in a Class (Empty, Comment, Code)
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	600
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.Class.LOC

Metric-Lines.Code

Lines containing Code.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing Code
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.Code

Metric-Lines.Comment

Lines containing Comments.

Input: RFG

Details

Counts the lines within the function that contain a comment.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing Comments
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.Comment

Metric-Lines.Empty

Empty Lines.

Input: RFG

Details

Count of lines within the function that are completely empty (whitespaces allowed).

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Empty Lines
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	100
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.Empty

Metric-Lines.File.Code

Lines in File containing Code.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	Include
display_name	Description of the metric shown in the dashboard.	Lines in File containing Code
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	800
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.File.Code

Metric-Lines.File.Comment

Lines in File containing Comments.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	Include
display_name	Description of the metric shown in the dashboard.	Lines in File containing Comments
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	800
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.File.Comment

Metric-Lines.File.Comment_Ratio

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	None
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.File.Comment_Ratio

Metric-Lines.File.LOC

Lines of Code in a File (Empty, Comment, Code).

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	Include
display_name	Description of the metric shown in the dashboard.	Lines of Code in a File (Empty, Comment, Code)
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	800
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.File.LOC

Metric-Lines.LOC

Lines of Code (Empty, Comment, Code).

Input: RFG

Details

Counts the lines of code of the routine, starting at the beginning of the function, and ending at the line where the closing curly brace is located.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines of Code (Empty, Comment, Code)
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.LOC

Metric-Lines.Only_Comment

Lines containing only Comments.

Input: RFG

Details

Counts the lines within the function that contain only comments (but no other language tokens).

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing only Comments
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	100
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.Only_Comment

Metric-Lines.PP_Define

Lines containing #define.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict[...]
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing #define
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.PP_Define

Metric-Lines.PP_Elif

Lines containing #elif.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing #elif
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.PP_Elf

Metric-Lines.PP_Else

Lines containing #else.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing #else
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.PP_Elf

Metric-Lines.PP_Endif

Lines containing #endif.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing #endif
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.PP_Endif

Metric-Lines.PP_Error

Lines containing #error.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing #error
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.PP_Error

Metric-Lines.PP_Ident

Lines containing #ident.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing #ident
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.PP_Ident

Metric-Lines.PP_If

Lines containing #if.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing #if
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.PP_Ifdef

Metric-Lines.PP_Ifdef

Lines containing #ifdef.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing #ifdef
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.PP_Ifdef

Metric-Lines.PP_Ifndef

Lines containing #ifndef.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict[...]
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing #ifndef
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.PP_Ifndef

Metric-Lines.PP_Include

Lines containing #include.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict[...]
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing #include
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.PP_Include

Metric-Lines.PP_Line

Lines containing #line.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing #line
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.PP_Line

Metric-Lines.PP_Pragma

Lines containing #pragma.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing #pragma
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.PP_Pragma

Metric-Lines.PP_Sharp

Lines containing #.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing #
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.PP_Sharp

Metric-Lines.PP_Sharp_Sharp

Lines containing ##.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing ##
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.PP_Sharp_Sharp

Metric-Lines.PP_Undef

Lines containing #undef.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing #undef
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.PP_Undef

Metric-Lines.PP_Warning

Lines containing #warning.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing #warning
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.PP_Warning

Metric-Lines.Preproc

Lines containing Preprocessor Directives.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines containing Preprocessor Directives
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.Preproc

Metric-Lines.Routine.Code

Lines in Routine containing Code.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines in Routine containing Code
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	100
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.Routine.Code

Metric-Lines.Routine.Comment

Lines in Routine containing Comments.

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines in Routine containing Comments
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	100
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.Routine.Comment

Metric-Lines.Routine.Comment_Ratio

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	None
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.Routine.Comment_Ratio

Metric-Lines.Routine.Comment_Ratio.LOC_Qualified

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	None
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.Routine.Comment_Ratio.LOC_Qualified

Metric-Lines.Routine.LOC

Lines of Code in a Routine (Empty, Comment, Code).

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lines of Code in a Routine (Empty, Comment, Code)
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	100
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Lines.Routine.LOC

Metric-LogNPath_Ceiling

LogNPath_Ceiling.

Input: RFG

Details

Computes ceiling(log2[number of maximal acyclic execution paths]) within a function. The rules for each syntactical construct can be found in "NPATH: a measure of execution path complexity and its applications" (Brian A. Nejmeh, Communications of the ACM, Volume 31 Issue 2, Feb. 1988).

The values of this metric can be very high and will be cut off at $2^{31} - 1$ (the highest possible value for any integer metric in the RFG).

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	LogNPath_Ceiling
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.LogNPath_Ceiling

Metric-LogNPath_Floor

LogNPath_Floor.

Input: RFG

Details

Computes floor(log2[number of maximal acyclic execution paths]) within a function. The rules for each syntactical construct can be found in "NPATH: a measure of execution path complexity and its applications" (Brian A. Nejmeh, Communications of the ACM, Volume 31 Issue 2, Feb. 1988).

The values of this metric can be very high and will be cut off at $2^{31} - 1$ (the highest possible value for any integer metric in the RFG).

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	LogNPath_Floor
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.LogNPath_Floor

Metric-MI_Per_Routine

Input: RFG

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
attr_name_system	Name of the node attribute storing the maintainability index in the system node.	Metric.MI
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	None
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.MI_Per_Routine

Metric-Maximum_Extended_Nesting

Input: RFG

Details

Counts the maximum nesting level of syntactic levels inside a routine.

Maximum Extended Nesting differs from Metric.Maximum_Nesting by its treatment of else if-statements: Maximum Extended Nesting increases the nesting level by one for each else if-statement.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Maximum Extended Nesting
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Maximum_Extended_Nesting

Metric-Maximum_Nesting

Maximum Nesting.

Input: RFG

Details

Counts the maximum nesting level of syntactic levels inside a routine.

Maximum Nesting differs from Metric.Maximum_Extended_Nesting by its treatment of else if-statements: Maximum Nesting does not increase the nesting level by one for each else if-statement.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Maximum Nesting
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	7
min_value	Minimal allowed value. None if unlimited.	1
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Maximum_Nesting

Metric-McCabe_Complexity

McCabe Cyclomatic Complexity.

Input: RFG

Details

Computes the number of nodes in the control flow graph of a routine that have more than one outgoing edge.

For nodes originating in switch-statements, each nonempty case section is counted. It is also known as cyclomatic complexity.

Roughly speaking, the value McCabe Cyclomatic Complexity is computed by summing up

- the number of if-statements,
- the number of loop statements (e.g. for and while),
- the number of conditional (?) operators,
- the number of catch clauses, and
- the number of nonempty case-sections
- plus the value one.

So McCabe Cyclomatic Complexity is always greater zero.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	McCabe Cyclomatic Complexity
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	20
min_value	Minimal allowed value. None if unlimited.	1
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.M McCabe_Complexity

Metric-NPath

NPath.

Input: RFG

Details

Computes the number of maximal acyclic execution paths within a function. The rules for each syntactical construct can be found in "NPATH: a measure of execution path complexity and its applications" (Brian A. Nejmeh, Communications of the ACM, Volume 31 Issue 2, Feb. 1988).

The values of this metric can be very high and will be cut off at $2^{31} - 1$ (the highest possible value for any integer metric in the RFG).

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	NPath
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.NPath

Metric-Number_Of_Called_Routines

Number of functions this function is calling.

Input: RFG

Details

Counts for a given routine r the number of distinct routines that r calls.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Number of functions this function is calling
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	7
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Number_Of_Called_Routines

Metric-Number_Of_Calling_Routines

Number of functions calling this function.

Input: RFG

Details

Counts for a given routine r the number of distinct routines that call r .

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Number of functions calling this function
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	5
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Number_Of_Calling_Routines

Metric-Number_Of_Gotos

Number of Gotos.

Input: RFG

Details

Count of goto-statements in routine.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Number of Gotos
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	0
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Number_Of_Gotos

Metric-Number_Of_Invocations

Number of Invocations.

Input: RFG

Details

Counts number of different calls within a routine.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Number of Invocations
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	1000
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Number_Of_Invocations

Metric-Number_Of_Parameters

Number of Parameters.

Input: RFG

Details

This metric counts the number of parameters of a function/method as to give an indication for the size of the interface.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Number of Parameters
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	8
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Number_Of_Parameters

Metric-Number_Of_Returns

Number of Returns.

Input: RFG

Details

Count of `return` statements in routine.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Number of Returns
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	10
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Number_Of_Returns

Metric-Number_Of_SCs

Number of recursions (ap_cg_cycle).

Input: RFG

Details

This metric counts the number of call-graph recursions.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Number of recursions (ap_cg_cycle)
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	0
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Number_Of_SCs

Metric-Number_Of_Statements

Number of Statements.

Input: RFG

Details

Count of all statements in routine.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
allow_empty_constructor	Do not treat an empty constructor as violation.	False
allow_empty_destructor	Do not treat an empty destructor as violation.	False
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Number of Statements
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	1000
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Number_Of_Statements

Metric-OO.CBO

Coupling between object classes.

Input: RFG

Details

Number of classes that are coupled to a class C, i.e., the size of the union set of classes that are using fields and methods of C and the classes that are used by C's fields and methods (so both directions count here).

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Coupling between object classes
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.00.CBO

Metric-00.CB0a

Coupling between objects (excluding ancestors).

Input: RFG

Details

Number of classes that are coupled to a class C, i.e., the size of the union set of classes that are using fields and methods of C and the classes that are used by C's fields and methods (so both directions count here).

Ancestor classes of a class are not taken into account.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Coupling between objects (excluding ancestors)
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	5
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.00.CB0a

Metric-00.CB0s

Coupling between objects (excluding statics).

Input: RFG

Details

Number of classes that are coupled to a class C, i.e., the size of the union set of classes that are using fields and methods of C and the classes that are used by C's fields and methods (so both directions count here).

Static data members are not taken into account when looking at the dependencies to and from other classes.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Coupling between objects (excluding statics)
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	5
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.00.CBOs

Metric-00.DAC

Data abstraction coupling.

Input: RFG

Details

Number of attributes in a class that have another class as their type.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Data abstraction coupling
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	7
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.00.DAC

Metric-00.DIT

Depth of Inheritance Tree.

Input: RFG

Details

DIT of a class is the length of the longest path from class to the root in the inheritance hierarchy. Classes that do not inherit anything have a DIT of 0.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Depth of Inheritance Tree
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	5
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.00.DIT

Metric-00.LCOM

Lack of cohesion in methods.

Input: RFG

Details

Count of function member pairs of a class that have no similarity regarding the common use of data members.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lack of cohesion in methods
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
ignore_artificial_methods	Do not count artificial methods.	True
ignore_deleted_methods	Do not count deleted methods.	True
max_value	Maximal allowed value. None if unlimited.	5
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.00.LCOM

Metric-00.LCOMs

Lack of cohesion in non-static methods.

Input: RFG

Details

Count of function member pairs of a class that have no similarity regarding the common use of data members. Do not count static data members.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Lack of cohesion in non-static methods
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
ignore_artificial_methods	Do not count artificial methods.	True
ignore_deleted_methods	Do not count deleted methods.	True
max_value	Maximal allowed value. None if unlimited.	5
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.OO.LCOMs

Metric-OO.NIVOC

Number of invocations.

Input: RFG

Details

Calculates the number of routines called inside a routine.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Number of invocations
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	1000
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.00.NIVOC

Metric-00.NOA

Number of ancestor classes.

Input: RFG

Details

Number of classes that a given class directly or indirectly inherits from.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Number of ancestor classes
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	20
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.00.NOA

Metric-00.NOCC

Number of child classes.

Input: RFG

Details

Number of classes that directly inherit from a given class.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Number of child classes
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	10
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.00.NOCC

Metric-00.NOFA

Number of fields added.

Input: RFG

Details

Calculates number of added fields (NOFA).

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Number of fields added
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.00.NOFA

Metric-00.NOMA

Number of methods added.

Input: RFG

Details

Number of function members (including operators, constructors, and destructors) that a given class adds.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Number of methods added
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	10
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.00.NOMA

Metric-OO.NOMO

Number of methods overridden.

Input: RFG

Details

Number of function members that a given class overrides.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Number of methods overridden
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	10
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.OO.NOMO

Metric-OO.NOPC

Number of parent classes.

Input: RFG

Details

Number of classes that a given class directly inherits from.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Number of parent classes
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	5
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.00.NOPC

Metric-OO.RFC

Response for a class.

Input: RFG

Details

The response of a class is the count of methods of the class itself and all methods/functions called by the methods of the class (i.e., we take one level of the call tree into account). Inherited but not overridden methods are not taken into account; the potential fan out generated by calls to virtual functions is also not taken into account.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Response for a class
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
ignore_artificial_methods	Do not count artificial methods.	True
ignore_deleted_methods	Do not count deleted methods.	True
max_value	Maximal allowed value. None if unlimited.	50
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.00.RFC

Metric-00.WMC.McCabe_Complexity

Weighted Methods per Class based on McCabe Complexity.

Input: RFG

Details

Sum over all weighted defined or overridden methods of a class where the metric Metric.McCabe_Complexity is used as weight. Inherited methods are not counted.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Weighted Methods per Class based on McCabe Complexity
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
ignore_artificial_methods	Do not count artificial methods.	True
ignore_deleted_methods	Do not count deleted methods.	True
max_value	Maximal allowed value. None if unlimited.	200
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.00.WMC.M McCabe_Complexity
weight_attribute	RFG node attribute used as weight for the method. If None, a weight of 1 is used for every method.	Metric.M McCabe_Complexity

Metric-00.WMC.One

Weighted Methods per Class based on 1.

Input: RFG

Details

Sum over all weighted defined or overridden methods of a class where 1 is used as weight. Inherited methods are not counted.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Weighted Methods per Class based on 1
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
ignore_artificial_methods	Do not count artificial methods.	True
ignore_deleted_methods	Do not count deleted methods.	True
max_value	Maximal allowed value. None if unlimited.	20
min_value	Minimal allowed value. None if unlimited.	0
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.OO.WMC.One
weight_attribute	RFG node attribute used as weight for the method. If None, a weight of 1 is used for every method.	None

Metric-Switch_Complexity

Switch Complexity.

Input: RFG

Details

Computes the number of nodes in the control flow graph of a routine that have more than one outgoing edge.

For nodes originating in switch-statements, each nonempty case section is counted. It is also known as cyclomatic complexity.

This metrics *not* count short circuit logical operators. In addition, the case-sections of a switch-section are not counted. Instead, each occurring switch-statement as a whole increases the value by one.

Roughly speaking, the value is computed by summing up

- the number of if-statements,
- the number of loop statements (e.g. for and while),
- the number of conditional (?) operators,
- the number of catch clauses, and
- the number of switch-statements
- plus the value one.

So value of this metric is always greater zero.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict{...}
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Description of the metric shown in the dashboard.	Switch Complexity
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Maximal allowed value. None if unlimited.	None
min_value	Minimal allowed value. None if unlimited.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Name of the node attribute storing the metric value in the RFG.	Metric.Switch_Complexity

Metric-TokenFileMetric

Lines containing certain configurable tokens.

Input: RFG

Details

Counts lines that contain one or more times the configured tokens (inside a comment or as language token). The lookup itself is case insensitive, so that FooBar, FOOBAR, and foobar are all accounted.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	Include
display_name	Used internal. DO NOT USE IT - use tokens['token-name']['display_name'] instead.	None
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Used internal. DO NOT USE IT - use tokens['token-name']['max_value'] instead.	None
min_value	Used internal. DO NOT USE IT - use tokens['token-name']['min_value'] instead.	None
node_type_name		File
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Used internal. DO NOT USE IT.	
tokens	Dictionary string --> {'display_name' --> string, 'dashboard_name' --> string None, 'min_value' --> float, 'max_value' --> float, 'disable' --> boolean}. * The key is the name of the token. * display_name: description of the metric shown in dashboard. * dashboard_name: Value of column Metric in dashboard (by default hidden). If None: 'Metric.Lines.' + key * min_value: minimal number of allowed lines containing the token. * max_value: maximal number of allowed lines containing the token. * disable: disables this token type (optional).	dict(...)

Metric-TokenMetric

Lines containing certain configurable tokens.

Input: RFG

Details

Counts lines that contain one or more times the configured tokens (inside a comment or as language token). The lookup itself is case insensitive, so that FooBar, FOOBAR, and foobar are all accounted.

Configuration

Name	Explanation	Value
aggregation_names	Display names (descriptions) of the aggregated metric values as format strings. {0} is replaced by the display name of the metric.	dict(...)
base_view_name	Name of the base view used to calculate/read the metric values. If None, the value of CONFIG.Base_View_Name is used.	None
display_name	Used internal. DO NOT USE IT - use tokens['token-name']['display_name'] instead.	None
hierarchy_edge_name	Name of the edge used to traverse the hierarchy view. If None, the value of CONFIG.Hierarchy_Edge_Name is used.	None
hierarchy_view_name	Name of the hierarchy view used to propagate the metric values. If None, the value of CONFIG.Hierarchy_View_Name is used.	None
max_value	Used internal. DO NOT USE IT - use tokens['token-name']['max_value'] instead.	None
min_value	Used internal. DO NOT USE IT - use tokens['token-name']['min_value'] instead.	None
propagate	Propagate and aggregate the metric values (leaf values) up through the hierarchy view. If None, the value of CONFIG.Propagate_Metrics is used.	None
report_all_values	Report all metric values. If None, the value of CONFIG.Report_All_Metric_Values is used.	None
report_propagated_values	Report propagated metric values. 'no': Don't report (only store in RFG). 'root': Report value at hierarchy root node. 'all': Report all propagated values. If None, the value of CONFIG.Report_Propagated_Metric_Values is used.	None
rfg_metric_name	Used internal. DO NOT USE IT.	
tokens	Dictionary string --> {'display_name' --> string, 'dashboard_name' --> string None, 'min_value' --> float, 'max_value' --> float, 'disable' --> boolean}. * The key is the name of the token. * display_name: description of the metric shown in dashboard. * dashboard_name: Value of column Metric in dashboard (by default hidden). If None: 'Metric.Lines.' + key * min_value: minimal number of allowed lines containing the token. * max_value: maximal number of allowed lines containing the token. * disable: disables this token type (optional).	dict(...)

Rules in Group MisraC

MisraC-1.1

All code shall conform to ISO/IEC 9899:1990 "Programming languages -- C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
limits	Translation limits, defaults to C90-guaranteed limits. You can change individual limits, e.g. from bauhaus.ir.common.languages import translation_limits followed by RULES['...'].limits[translation_limits.enumerators] = 400000 The dict-keys usable here can be found in the IR reference guide where the module bauhaus.ir.languages.translation_limits is described. You can also assign your own translation limit class when it follows the standard ones from the translation_limits module: RULES['...'].limits = MyCompilerLimits()	C90_Limits
show_include_path	If True, violations regarding nested includes print the #include path.	False
standards	List of allowed frontend command-line options to select language standard.	['--c89']

Possible Messages

Name	Message
access_control	Number of access control declarations is higher than translation limit (which is {})
at_quick_exit_functions	Number of functions registered with at_quick_exit() is higher than translation limit (which is {})
atexit_functions	Number of functions registered with atexit() is higher than translation limit (which is {})
base_classes	Number of direct and indirect base classes is higher than translation limit (which is {})
call_arguments	Number of arguments is higher than translation limit (which is {})

catch_handlers	Number of catch handlers is higher than translation limit (which is {})
chars_in_line	Number of characters in logical line is higher than translation limit (which is {})
chars_in_string	Number of characters in string is higher than translation limit (which is {})
cpp_members	Number of members is higher than translation limit (which is {})
data_members	Number of data members is higher than translation limit (which is {})
direct_bases	Number of direct base classes is higher than translation limit (which is {})
enumerators	Number of enumerators is higher than translation limit (which is {})
expr_in_core_constant	Number of full expressions evaluated within a core constant expression is higher than translation limit (which is {})
external_identifiers	Number of external identifiers in this unit is higher than translation limit (which is {})
final_virtual_functions	Number of final overriding virtual functions is higher than translation limit (which is {})
friends	Number of friend declarations is higher than translation limit (which is {})
function_parameters	Number of parameters is higher than translation limit (which is {})
macro_arguments	Number of macro arguments is higher than translation limit (which is {})
macro_parameters	Number of macro parameters is higher than translation limit (which is {})
macros	Number of macro definitions in this unit is higher than translation limit (which is {})
member_initializers	Number of member initializers is higher than translation limit (which is {})
nested_blocks	Number of nested blocks is higher than translation limit (which is {})
nested_composites	Nesting level of structs and unions is higher than translation limit (which is {})
nested_declarators	Number of pointer/array/function declarators is higher than translation limit (which is {})
nested_externals	Number of nested external specifications is higher than translation limit (which is {})
nested_includes	Nesting level of #includes is higher than translation limit (which is {})
nested_preprocessor_ifs	Nesting level of #if is higher than translation limit (which is {})
nested_template_instantiations	Number of nested template instantiations is higher than translation limit (which is {})
object_size	Number of bytes in object is higher than translation limit (which is {})
parenthesized_declarators	Number of parenthesized declarators is higher than translation limit (which is {})
parenthesized_expressions	Number of parenthesized subexpressions is higher than translation limit (which is {})
placeholders	Number of placeholders is higher than translation limit (which is {})
recursive_constexpr	Number of recursive constexpr function invocations is higher than translation limit (which is {})
scope_qualifications	Number of scope qualifications is higher than translation limit (which is {})
significant_chars_external	Number of significant initial characters for external identifier is higher than translation limit (which is {})
significant_chars_internal	Number of significant initial characters for internal identifier is higher than translation limit (which is {})
static_members	Number of static members is higher than translation limit (which is {})
std_compile	You have to compile with option {}
switch_cases	Number of cases is higher than translation limit (which is {})
template_params	Number of template parameters is higher than translation limit (which is {})
throw_spec	Number of throw specifications is higher than translation limit (which is {})
variables_in_block	Number of variables in block is higher than translation limit (which is {})
virtual_bases	Number of virtual base classes is higher than translation limit (which is {})

MisraC-1.2

No reliance shall be placed on undefined or unspecified behaviour.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_extra_args	Whether to allow additional arguments that are not used by the format string.	True
allow_gnu_extensions	Whether to allow the GNU extensions to format specifications.	False
allow_unknown_specs	Whether to allow unknown format specifications. It may be necessary to set this option when using implementation-specific extensions. Arguments are not checked when the format string contains unknown format specifications.	False
functions	A dictionary mapping the names of the functions to check, to a triple {function_kind, fmt_param_index, arg_start_index} where function_kind is either 'printf' or 'scanf', fmt_param_index is the index of the format-string parameter, and arg_start_index is the index of the first variadic argument.	dict(...)

Possible Messages

Name	Message
arg_type_mismatch	{} expects argument of type '{}', but argument {} has type '{}'
buffer_too_small	{} may write up to {} characters to buffer of size {}.
empty_struct	Empty struct has undefined behavior
empty_union	Empty union has undefined behavior
invalid_conversion	Invalid or non-standard conversion specification
matching_arg_expected	{} expects a matching '{}' argument
missing_inline_def_in_multiple_units	Inline function is missing definition in {} compilation units (e.g. '{}')
missing_inline_def_in_single_unit	Inline function is missing definition in compilation unit '{}'
precision_for_conversion	Precision must not be used with %{} conversion specifier
struct_without_named	struct without named members has undefined behavior
too_many_args	Too many arguments for format.
union_without_named	union without named members has undefined behavior
unknown_buffer_size	Potential buffer overflow: {} used with buffer of unknown size.
unlimited_read	Potential buffer overflow: {} has no limit on amount of characters read.
unsupported_assignment_suppression	%n does not support assignment suppression
unsupported_field_width	%n does not support field width
unsupported_flags	%n does not support flags
unsupported_flags_modifiers	Cannot use any flags or modifiers with '%%'
unsupported_hash	%{} does not support the '#' flag
unsupported_i_flag	%{} does not support the 'l' flag
unsupported_length_modifier	%{} does not support the '{}' length modifier
unsupported_tick	%{} does not support the "" flag
unsupported_zero	%{} does not support the '0' flag

MisraC-2.1

Assembly language shall be encapsulated and isolated.

Input: IR
Source languages: Assembler, C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
asm_not_encapsulated	Assembly language shall be encapsulated and isolated.

MisraC-2.2

Source code shall only use /* ... */ style comments.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
files_to_check	Files to apply this check to (Primary_File / User_Include_File / System_Include_File)	('Primary_File', 'User_Include_File')

Possible Messages

Name	Message
cpp_comment	Use of C99 style comment.

MisraC-2.3

The character sequence /* shall not be used within a comment.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
check_cpp_comments	Whether to look for /* in C++-style comments as well.	False
files_to_check	Files to apply this check to (Primary_File / User_Include_File / System_Include_File)	('Primary_File', 'User_Include_File')

Possible Messages

Name	Message
nested_c_comment	C-style comment containing /* sequence.

MisraC-2.4

Sections of code should not be "commented out".

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
files_to_check	Files to be checked, e.g. Primary_File or File.	Primary_File
normal_text	Tokens that indicate non-code comments. This is the first step of checking a comment. If any of these words exist in the comment text, it is regarded as not containing code.	{' and ', ' or ', ' but ', ' now ', ' to ', ' is ', ' are ', ' only ', ' be ', ' has ', ' the ', ' with ', ' because ', ' when ', ' oder ', ' und ', ' :// ', '#', 'AXIVION', '++++', '----', '===='}
normal_text_regex	Regular expression to match normal non-code text. This is the second step of checking a comment. If this regex matches the comment text, the comment is regarded as not containing code.	\b[\w\d_]+\s+[\w\d_]+\s+[\w\d_]+\b
suspicious	Tokens that indicate code in comments. This is the final step of checking a comment. If the first two steps did not exclude the comment from further inspection, the comment text is searched for any of these suspicious tokens. The comment is regarded as containing code, if any suspicious token is found.	{' = ', ' == ', ' >= ', ' <= ', '[', ';', ':', '>', '>*', ' ::*', 'if', 'while', 'for', 'if', 'while', 'for', '#pragma', '#else', '#endif', '#if', '#include', '++', '--'}

Possible Messages

Name	Message
code_in_c_comment	Sections of code shall not be commented out using C-style comments.
code_in_cpp_comment	Sections of code should not be commented out using C++ comments.

MisraC-3.1

All usage of implementation-defined behaviour shall be documented.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
files_to_check	Files to be checked for scanner-based part of the rule, e.g. Primary_File or File.	Primary_File
show_include_path	If True, violations regarding nested includes print the #include path.	False

Possible Messages

Name	Message
access_control	Number of access control declarations is higher than translation limit (which is {})
at_quick_exit_functions	Number of functions registered with at_quick_exit() is higher than translation limit (which is {})
atexit_functions	Number of functions registered with atexit() is higher than translation limit (which is {})
base_classes	Number of direct and indirect base classes is higher than translation limit (which is {})
call_arguments	Number of arguments is higher than translation limit (which is {})
catch_handlers	Number of catch handlers is higher than translation limit (which is {})
chars_in_line	Number of characters in logical line is higher than translation limit (which is {})
chars_in_string	Number of characters in string is higher than translation limit (which is {})
cpp_members	Number of members is higher than translation limit (which is {})
data_members	Number of data members is higher than translation limit (which is {})
direct_bases	Number of direct base classes is higher than translation limit (which is {})
enumerators	Number of enumerators is higher than translation limit (which is {})
expr_in_core_constant	Number of full expressions evaluated within a core constant expression is higher than translation limit (which is {})
external_identifiers	Number of external identifiers in this unit is higher than translation limit (which is {})
final_virtual_functions	Number of final overriding virtual functions is higher than translation limit (which is {})

friends	Number of friend declarations is higher than translation limit (which is {})
function_parameters	Number of parameters is higher than translation limit (which is {})
macro_arguments	Number of macro arguments is higher than translation limit (which is {})
macro_parameters	Number of macro parameters is higher than translation limit (which is {})
macros	Number of macro definitions in this unit is higher than translation limit (which is {})
member_initializers	Number of member initializers is higher than translation limit (which is {})
multi_character_char_literal	More than one character in character literal
nested_blocks	Number of nested blocks is higher than translation limit (which is {})
nested_composites	Nesting level of structs and unions is higher than translation limit (which is {})
nested_declarators	Number of pointer/array/function declarators is higher than translation limit (which is {})
nested_externals	Number of nested external specifications is higher than translation limit (which is {})
nested_includes	Nesting level of #includes is higher than translation limit (which is {})
nested_preprocessor_ifs	Nesting level of #if is higher than translation limit (which is {})
nested_template_instantiations	Number of nested template instantiations is higher than translation limit (which is {})
nonbasic_char_in_comment	Using characters which are not from the basic source character set relies on implementation-defined behaviour
nonbasic_char_in_filename	Using characters which are not from the basic source character set relies on implementation-defined behaviour
nonbasic_char_inPragma	Using characters which are not from the basic source character set relies on implementation-defined behaviour
nonbasic_char_in_string	Using characters which are not from the basic source character set relies on implementation-defined behaviour
object_size	Number of bytes in object is higher than translation limit (which is {})
parenthesized_declarators	Nesting level of parenthesized declarators is higher than translation limit (which is {})
parenthesized_expressions	Nesting level of parenthesized subexpressions is higher than translation limit (which is {})
placeholders	Number of placeholders is higher than translation limit (which is {})
pragma	Using #pragma relies on implementation-defined behaviour
recursive_constexpr	Number of recursive constexpr function invocations is higher than translation limit (which is {})
scope_qualifications	Number of scope qualifications is higher than translation limit (which is {})
shift_right_negative	If first operand is negative, using this bitwise operator relies on implementation-defined behaviour
significant_chars_external	Number of significant initial characters for external identifier is higher than translation limit (which is {})
significant_chars_internal	Number of significant initial characters for internal identifier is higher than translation limit (which is {})
static_members	Number of static members is higher than translation limit (which is {})
switch_cases	Number of cases is higher than translation limit (which is {})
template_params	Number of template parameters is higher than translation limit (which is {})
throw_spec	Number of throw specifications is higher than translation limit (which is {})
variables_in_block	Number of variables in block is higher than translation limit (which is {})
virtual_bases	Number of virtual base classes is higher than translation limit (which is {})

MisraC-3.4

All uses of the #pragma directive shall be documented and explained.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
pragma	Use of #pragma

MisraC-4.1

Only those escape sequences that are defined in the ISO C standard shall be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_hex_and_octal	Whether hexadecimal and octal escape sequences should be accepted without further check.	False
allowed	List of allowed characters after backslash.	"\"?\\abfnrtv0

Possible Messages

Name	Message
hex_escape_sequence	Use of hexadecimal escape sequence.
nonstandard_escape_sequence	Use of non-standard escape sequence.
octal_escape_sequence	Use of octal escape sequence.

MisraC-4.2

Trigraphs shall not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_in_comments	Allow the use of trigraphs in comments	True

Possible Messages

Name	Message
trigraph_use	Use of trigraph.

MisraC-5.1

Identifiers (internal and external) shall not rely on the significance of more than 31 characters.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
exclude_identifiers_sharing_significant_characters	If True, only long identifiers not sharing all significant characters with other identifiers are reported.	True
maxlen	Number of significant characters to consider.	31
report_external_identifiers	Whether external identifiers should be compared to each other.	True
report_internal_identifiers	Whether internal identifiers should be compared to each other (including macros).	True
report_long_identifiers	If True, reports all identifiers longer than the limit. If False, only identifiers sharing too many characters are reported.	False
report_short_identifiers	If True, identifiers shorter than maxlen are considered as well.	True

Possible Messages

Name	Message
external_identifiers_not_distinct	External identifiers not distinct.
external_identifiers_sharing	External identifiers sharing first {} characters.
identifier_too_long	Identifier is longer than {} characters.
internal_identifiers_not_distinct	Internal identifiers not distinct.
internal_identifiers_sharing	Internal identifiers sharing first {} characters.

MisraC-5.2

Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allow_in_classes	Whether identifiers in local classes can hide surrounding ones	False
allow_in_lambdas	Whether identifiers in lambdas can hide surrounding ones	False
allow_in_local_classes	Whether identifiers in local classes can hide surrounding ones	False
allow_in_namespaces	Whether identifiers in namespaces can hide surrounding ones	False
allow_parameter_in_function_declaration	Whether identifiers of parameters in function declarations may hide surrounding ones, given the function declaration uses a different identifier for the same parameter that does not hide a surrounding one.	False
check_inheritance_hiding	Whether to check for hiding by inheritance.	False
check_nested_classes	Whether symbols from inner classes should be reported as hiding symbols of outer classes.	False
distinguish_name_spaces	Whether C name spaces should be distinguished.	True
hiding_entities	LIR node types to consider symbols that hide other symbols.	{'Variable', 'Typedef_Type', 'Field', 'Parameter'}
maxlen	Number of significant characters (or None)	None
tolerate_constructor_parameters_hiding_fields	Allow constructor parameters to hide fields	True
tolerate_function_prototype_argument_hiding	Whether to tolerate arguments of a function prototype, that itself is a function argument, that have the same identifier as a argument of the outer function. If 'True' is given, the rule will still report cases, where argument identifiers match up and the function prototype refers to arguments of the outer function (e.g. by `decltype()`).	False
tolerate_macro_local_identifier	Whether an identifier being local to a macro is allowed to hide identifiers from outside the macro	False
unchecked_types	Deprecated legacy option, kept to avoid breaking old configurations.	{'Routine_Call', 'Stack_Object_Definition'}

Possible Messages

Name	Message
hiding	{ } hides { }

MisraC-5.3

A typedef name shall be a unique identifier.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
tolerate_macros	Whether #define and #undef using the typedef's name is allowed	False
tolerate_typedef_entity	Whether the entity forming the typedef's underlying type can have the same name.	False

Possible Messages

Name	Message
reused_typedef	Typedef name reused.

MisraC-5.4

A tag name shall be a unique identifier.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
check_types_only_with_external_linkage	Whether the types should only be checked when they have external linkage	False
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	If provided, only messages of these types are reported.	[469]
reported_severities	List of severities to display.	('error', 'warning', 'remark')
tolerate_typedef	Whether a typedef to the tag may have the same name.	False
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

Possible Messages

Name	Message
composite_defined_multiple_times	Composite type has multiple definitions.
enum_defined_multiple_times	Enumeration type has multiple definitions.
reused_tag	Tag name reused.

MisraC-5.5

No object or function identifier with static storage duration should be reused.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allow_static_no_linkage	Whether to allow same identifiers for static objects without linkage (e.g. function local static objects)	False
func_filter	Restricts which functions are considered.	None
use_unqualified_names	If set to True, ns1::Type is considered equal to ns2::Type.	True
var_filter	Restricts which variables are considered.	None

Possible Messages

Name	Message
reused_global_function_name	Name of object or function with static storage duration reused.
reused_global_variable_name	Name of object or function with static storage duration reused.

MisraC-5.6

No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure member and union member names.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
exclude_system_headers	Whether to exclude identifiers appearing in system headers	False

Possible Messages

Name	Message
name_reused_in_different_c_name_space	Name reused in different name space.

MisraC-5.7

No identifier name should be reused.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
check_locals	Whether local variables should be considered identifiers to check.	True
check_macros	Whether macro names should be considered identifiers to check.	False
check_pragmas	Whether pragma names should be considered identifiers to check.	False
exclude_system_headers	Whether to exclude identifiers appearing in system headers	False
minlen	Minimum length of identifiers: Shorter ones will not be considered.	1

Possible Messages

Name	Message
reused_name	Identifier name reused.

MisraC-6.1

The plain char type shall be used only for storage and use of character values.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
char_used_numerically	Plain char datatype used for non-character data.

MisraC-6.2

signed and unsigned char type shall be used only for the storage and use of numeric values.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
numeric_char_used_as_character	Signed/unsigned char datatype used for character data.

MisraC-6.3

"Typedefs" that indicate size and signedness should be used in place of the basic numerical types.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allow_floating_point	Whether floating point types without should not be reported.	False
ignore_inherited	If true, missing typedefs in inherited methods are not reported.	False
treat_plain_char_as_numerical	Whether the char type without any signedness should be reported.	False

Possible Messages

Name	Message
missing_integer_typedef	Use of base type outside typedef.
wrong_integer_typedef	Use of badly named typedef for base type.

MisraC-6.4

Bit fields shall only be defined to be of type unsigned int or signed int.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
nonint_bitfield	Bit-field type shall be unsigned int or signed int.

MisraC-6.5

Bit fields of signed type shall be at least 2 bits long.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
signed_single_bitfield	Signed bit field shall be at least 2 bits long.

MisraC-7.1

Octal constants (other than zero) and octal escape sequences shall not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
octal_escape_sequence	Use of octal escape sequence.
octal_literal	Use of octal literal.

MisraC-8.1

Functions shall have a prototype declaration and that prototype shall be visible at both the definition and at call.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_entry_points_without_prototype	If True, main() and additional entry functions need no prototype declaration.	False
allow_internal_functions_without_prototype	Whether definitions of functions with internal linkage are allowed without a separate prototype declaration	False

Possible Messages

Name	Message
missing_function_prototype	Functions shall have a prototype declaration
missing_parameter_type	Functions shall have a prototype declaration
parameterless_func_without_void_param	Function with no parameters shall be declared with (void)
prototype_not_visible_at_funcdef	Prototype declaration shall be visible at definition
reference_to_missing_function_prototype	Referenced function needs prototype declaration
reference_to_undeclared_function	Referenced function needs a declaration

MisraC-8.2

Whenever an object or function is declared or defined, its type shall be explicitly stated.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
implicit_int	Type shall be explicitly stated

MisraC-8.3

For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_parameter_type_qualifier_differences	If enabled, do not report a violation for different top-level type qualifiers in parameters: int f(int p); int f(const int p) {} Top-level qualifiers are only relevant for the implementation of the function and have no effect on callers, so it is not necessary to replicate them in the header file.	False
check_undefined	Whether only-declared routines should also be checked.	False
require_exact_match	Whether to check for identical or compatible types.	True

Possible Messages

Name	Message
decl_return_type_mismatch	Return type at declarations differs
parameter_type_mismatch	Parameter type at definition differs from type at declaration
return_type_mismatch	Return type at definition differs from type at declaration

MisraC-8.4

If objects or functions are declared more than once their types shall be compatible.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_parameter_type_qualifier_differences	If enabled, do not report a violation for different top-level type qualifiers in parameters: int f(int p); int f(const int p) {} Top-level qualifiers are only relevant for the implementation of the function and have no effect on callers, so it is not necessary to replicate them in the header file.	False
check_undefined	Whether only-declared routines and variables should also be checked.	True
require_exact_match	Whether to check for identical or compatible types (for routine return types).	False

Possible Messages

Name	Message
decl_return_type_mismatch	Return type at declarations differs
decl_type_mismatch	Type at declarations differs
parameter_type_mismatch	Parameter type at definition differs from type at declaration
return_type_mismatch	Return type at definition differs from type at declaration
type_mismatch	Type at definition differs from type at declaration

MisraC-8.5

There shall be no definitions of objects or functions in a header file.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
accept_const_fields	Whether const-qualified static fields in header files should be tolerated.	True
accept_const_variables	Whether global const variables in header files should be tolerated.	True
find_non_external_function_declaration	Whether function declarations with non-external linkage should be reported.	False

Possible Messages

Name	Message
function_definition_in_header	Definition in header file.
static_field_def_in_header	Definition in header file.
variable_definition_in_header	Definition in header file.

MisraC-8.6

Functions shall be declared at file scope.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
local_function_declaration	Functions shall not be declared at block scope.

MisraC-8.7

Objects shall be defined at block scope if they are only accessed from within a single function.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
move_global_const_into_function	Controls suggestions for global constants that could be declared locally in a function.	True
only_check_unit_locals	Whether only global static variables should be checked.	False

Possible Messages

Name	Message
locality_function	Global {} can be declared inside function.

MisraC-8.8

An external object or function shall be declared in one and only one file.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allow_only_one_decl	Enables message if there are multiple declarations but all in same file.	False

Possible Messages

Name	Message
function_declared_in_multiple_files	Function declared in more than one file.
function_declared_multiple_times	Function declared more than once.
type_declared_in_multiple_files	Type declared in more than one file.
type_declared_multiple_times	Type declared more than once.
variable_declared_in_multiple_files	Variable declared in more than one file.
variable_declared_multiple_times	Variable declared more than once.

MisraC-8.9

An identifier with external linkage shall have exactly one external definition.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allow_undefined_specialized_template	Whether undefined specialized templates are tolerated when applied to C++ code.	True
allow_unselected_declarations	Select whether to report declarations with no definition if they are unused.	False
allowed_undefined_functions	Regular expressions for functions which are tolerated without a definition, e.g. compiler intrinsics.	['_.*']
allowed_undefined_variables	Regular expressions for variables which are tolerated without a definition.	['_.*']

Possible Messages

Name	Message
function_defined_multiple_times	Type, object or function has multiple definitions.
undefined_function	Type, object or function without definition
undefined_variable	Type, object or function without definition
variable_defined_multiple_times	Type, object or function has multiple definitions.

MisraC-8.10

All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allow_moving_to_other_primary_file	When a variable is only used in a single file X, but currently implemented in a different file Y, this controls whether the check suggests to move it into X and make it static there.	False
exclude_dllexport	If True, no suggestions will be produced to make functions marked as dllexport or dllimport static in a primary file.	True
exclude_function_locals	If True, variables that could even be function-local are not reported (this avoids overlapping messages with rule 8.7).	True
exclude_undefined	Whether only-declared symbols should be reported as well	True
template_args_can_be_static	Whether functions whose address is used as template argument can be made static (some compilers, like Microsoft's, don't allow it then).	False

Possible Messages

Name	Message
function_file_static	{ } can be declared static in primary file.
var_file_static	{ } can be declared static in primary file.

MisraC-8.11

The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
function_decl_missing_static	Declaration should use static storage class specifier
function_def_missing_static	Definition should use static storage class specifier
variable_decl_missing_static	Declaration should use static storage class specifier
variable_def_missing_static	Definition should use static storage class specifier

MisraC-8.12

When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialisation.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
only_extern	Whether to consider only extern declared arrays	False
report_definitions	Whether definitions of array variables should also be reported	False

Possible Messages

Name	Message
unbounded_array	Array declaration with unknown size.

MisraC-9.1

All automatic variables shall have been assigned a value before being used.

Input: IR

Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
perform_global_analysis	Whether global analysis (using iranalysis) should be done, or just a local analysis.	True

Possible Messages

Name	Message
pass_as_pointer_to_const_param	Passing uninitialized variable by pointer as function parameter with pointer-to-const type
possible_uninit	Use of possibly uninitialized variable
uninit	Use of uninitialized variable

MisraC-9.2

Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_any_zero_initialization	Whether {0} is allowed in nested aggregates as well.	False
allow_toplevel_zero_initialization	Whether {0} is allowed as top-level initialization for the complete aggregate	True
allowed_classes_without_braces	For the listed class types, don't require braces. This is useful for classes that transparently wrap a single contained value. For example, with this option "std::array<int, 3> a = {1,2,3};" is OK even though there could be two sets of braces (one for "std::array", the other and for its inner "int[3]" array).	['std::array']

Possible Messages

Name	Message
incomplete_aggregate_initialization	Provide explicit initializer for each part of initialized entity
missing_aggregate_initialization_braces	Use braces to indicate structure of initialized entity

MisraC-9.3

In an enumerator list, the "=" construct shall not be used to explicitly initialise members other than the first, unless all items are explicitly initialised.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
incomplete_enum_init	Do not initialize enumerators other than the first, or initialize all

MisraC-10.1

The value of an expression of integer type shall not be implicitly converted to a different underlying type if: (a) it is not a conversion to a wider integer type of the same signedness, or (b) the expression is complex, or (c) the expression is not constant and is a function argument, or (d) the expression is not constant and is a return expression.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_constant_conversions	If this option is enabled, the rule is relaxed to allow implicit conversions of constant integer expressions whenever the constant value fits into the target type.	False
allow_null	Allow converting the 0 literal to other types (e.g. pointer types)	True
category_changes	List of [from, to] type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes], [SignedTypes, UnsignedTypes, FloatingTypes, CharacterTypes, BoolTypes, EnumTypes, VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes, NullPointerTypes, OtherTypes, VoidTypes]]]
check_explicit_casts	Whether explicit casts should be checked.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
report_narrowing	Whether narrowing casts should be reported.	True
report_widening	Whether widening casts should be reported.	True
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
size_source_category	Selection of type categories of the operand being cast.	[SignedTypes, UnsignedTypes]
size_target_category	Selection of type categories appearing as target types of the cast.	[SignedTypes, UnsignedTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Implicit conversion to different underlying type
narrowing_cast	Implicit narrowing conversion of integer type
widening_cast	Implicit widening conversion of integer type

MisraC-10.2

The value of an expression of floating type shall not be implicitly converted to a different type if: (a) it is not a conversion to a wider floating type, or (b) the expression is complex, or (c) the expression is a function argument, or (d) the expression is a return expression.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[FloatingTypes], [SignedTypes, UnsignedTypes, FloatingTypes, CharacterTypes, BoolTypes, EnumTypes, VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes, NullPointerTypes, OtherTypes, VoidTypes]]]
check_explicit_casts	Whether explicit casts should be checked.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
report_narrowing	Whether narrowing casts should be reported.	True
report_widening	Whether widening casts should be reported.	True
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
size_source_category	Selection of type categories of the operand being cast.	FloatingTypes
size_target_category	Selection of type categories appearing as target types of the cast.	FloatingTypes
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Implicit conversion from floating type to different type
narrowing_cast	Implicit narrowing conversion of floating type
widening_cast	Implicit widening conversion of floating type

MisraC-10.3

The value of a complex expression of integer type shall only be cast to a type of the same signedness that is no wider than the underlying type of the expression.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, BoolTypes, CharacterTypes, EnumTypes], [VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes, NullPointerTypes, FloatingTypes, CharacterTypes, BoolTypes, EnumTypes, OtherTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
report_narrowing	Whether narrowing casts should be reported.	False
report_signed_to_unsigned	Whether conversion from signed to unsigned shall be reported.	True
report_unsigned_to_signed	Whether conversion from unsigned to signed shall be reported.	True
report_widening	Whether widening casts should be reported.	True
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
size_source_category	Selection of type categories of the operand being cast.	[SignedTypes, UnsignedTypes, BoolTypes, CharacterTypes, EnumTypes]
size_target_category	Selection of type categories appearing as target types of the cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Explicit cast of complex expression to unrelated type
cast_from_signed_to_unsigned	Explicit cast of complex expression from signed to unsigned type
cast_from_unsigned_to_signed	Explicit cast of complex expression from unsigned to signed type
narrowing_cast	Conversion to smaller type
widening_cast	Explicit cast of complex expression to larger type

MisraC-10.4

The value of a complex expression of floating type shall only be cast to a floating type that is narrower or of the same size.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[FloatingTypes], [VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes, NullPointerTypes, SignedTypes, UnsignedTypes, CharacterTypes, BoolTypes, EnumTypes, OtherTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
report_narrowing	Whether narrowing casts should be reported.	False
report_widening	Whether widening casts should be reported.	True
show_operand_in_entity	Whether entity should be "from->to" or "{from->to}operand"	False
size_source_category	Selection of type categories of the operand being cast.	[FloatingTypes]
size_target_category	Selection of type categories appearing as target types of the cast.	[FloatingTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Explicit cast of complex float expression to unrelated type
narrowing_cast	Conversion to smaller type
widening_cast	Explicit cast of complex float expression to larger type

MisraC-10.5

If the bitwise operators ~ and << are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
bitop_small_without_cast	Bitwise operator requires cast to underlying type on result

MisraC-10.6

A "U" suffix shall be applied to all constants of unsigned type.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
check_only_hex_and_octal	Whether only hex and octal literals should be checked.	False
consider_conversion	Selects whether literals with an implicit conversion from signed to unsigned are reported.	False
consider_pp_expressions	Whether literals in #if conditions are checked.	False
report_all_unsuffixed_hex_and_octal	Whether hex and octal literals should always be reported when they are unsuffixed, independent of the environment	False

Possible Messages

Name	Message
missing_hex_suffix	Hexadecimal and octal constants require "U" suffix
missing_u_suffix	Constant of unsigned type requires "U" suffix

MisraC-11.1

Conversions shall not be performed between a pointer to a function and any type other than an integral type.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[FunctionPointerTypes], [VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes, NullPointerTypes, FloatingTypes, BoolTypes, OtherTypes]], [[FloatingTypes, BoolTypes, VoidTypes, OtherTypes, VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes], [FunctionPointerTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	True
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "(from->to)operand"	False
type_category	Selection of type categories to consider for type changes inside same category.	[FunctionPointerTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion between function pointer and other type
cast_changes_type_inside_category	Conversion between incompatible function pointer types

MisraC-11.2

Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[ObjectPointerTypes], [FloatingTypes, OtherTypes, FunctionPointerTypes, BoolTypes]], [[FloatingTypes, OtherTypes, FunctionPointerTypes, BoolTypes, VoidTypes], [ObjectPointerTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	True
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "{from->to}operand"	False
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion between object pointer and other type

MisraC-11.3

A cast should not be performed between a pointer type and an integral type.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes], [SignedTypes, UnsignedTypes, BoolTypes, EnumTypes, CharacterTypes]], [[SignedTypes, UnsignedTypes, BoolTypes, EnumTypes, CharacterTypes], [VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "{from->to}operand"	False
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion between pointer and integral type

MisraC-11.4

A cast should not be performed between a pointer to object type and a different pointer to object type.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "{from->to}operand"	False
type_category	Selection of type categories to consider.	[ObjectPointerTypes, IncompletePointerTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_inside_category	Cast between object pointer and other object pointer type

MisraC-11.5

A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
check_const	Whether casts affecting const-qualifiers should be reported	True
check_volatile	Whether casts affecting volatile-qualifiers should be reported	True
only_top_level	Whether the check only applies to the top-level pointee or recursively to deeper pointer levels as well	False

Possible Messages

Name	Message
cast_removes_const	Cast removes const qualification
cast_removes_volatile	Cast removes volatile qualification
implicit_cast_removes_const	Implicit cast removes const qualification
implicit_cast_removes_volatile	Implicit cast removes volatile qualification

MisraC-12.1

Limited dependence should be placed on C's operator precedence rules in expressions.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allow_algebraic_order	Whether parens can be omitted for multiplicative operators as operands in additive operators	False
allow_redundant_parentheses_in_logical_operator	Allow redundant parentheses around unary operators when they appear as operand of '&&' or ' '. This option can be used to avoid a conflict with MisraC++-5.2.1, which requires parentheses in these instances.	False
report_unnecessary_parentheses	Controls whether unnecessary use of parentheses on the right side of assignments or around unary operators are reported.	True

Possible Messages

Name	Message
assignment_rhs_in_parens	No parentheses required for right-hand side of assignment
missing_parens_depends_on_precedence	Parentheses should be used to avoid dependence on precedence rules
unary_op_in_parens	No parentheses required for unary operator

MisraC-12.2

The value of an expression shall be the same under any order of evaluation that the standard permits.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
ignore_exceptions	Ignore possible exceptions in the computation of side effects. (this option only has an effect if report_calls is enabled)	False
report_calls	If True, unsequenced function calls are reported.	False

Possible Messages

Name	Message
unsequenced_call_volatile	Function calls unsequenced with volatile access
unsequenced_calls	Unsequenced function calls
unsequenced_read_write	Unsequenced read and write accesses
unsequenced_volatiles	Unsequenced volatile accesses
unsequenced_writes	Unsequenced writes

MisraC-12.3

The sizeof operator shall not be used on expressions that contain side effects.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
side_effect_in_sizeof	Operand of "sizeof" shall not contain side effects

MisraC-12.4

The right-hand operand of a logical && or || operator shall not contain side effects.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_volatile_access	Whether access to a volatile variable should be reported as side effect.	False
assume_effect_when_undefined	Whether all calls to undefined (external) functions should be seen as having a side-effect.	True

Possible Messages

Name	Message
call_with_side_effect	Call in right-hand operand of && or may have side-effect
modifies_local_var	Right-hand operand of && or modifies '{}'
side_effect	Right-hand operand of && or has side-effect

MisraC-12.5

The operands of a logical && or || shall be primary-expressions.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allowed	Can be used to name additional allowed IR node types for the operands.	None
require_postfix_eprression	Whether postfix or primary expressions are required as operands.	False

Possible Messages

Name	Message
missing_parens_for_logical_operator_operand	Operand of && or must be parenthesized.

MisraC-12.6

The operands of logical operators (&&, || and !) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (&&, ||, !=, ==, != and ?:).

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_sizeof	Allows the use of sizeof(bool_var). The Misra rule forbids all uses of expressions of type bool except for those explicitly allowed; and does not allow sizeof. However this is likely an oversight; there is no good reason to prevent the use of sizeof(bool_var). This option also allows alignof(bool_var) in C++11.	False
allow_unevaluated_operator	Allows the use of boolean expressions in contexts where the expression is not evaluated. As of C++17, these are: typeid(bool_var), noexcept(bool_var) and decltype(bool_var). sizeof(bool_var) is controlled by the separate option 'allow_sizeof'.	True
only_in_conditions	If True, only logical operators inside conditions are checked.	False

Possible Messages

Name	Message
bool_operand_in_bad_operator	Use of boolean operand in '{}' operator
nonbool_logical_operator_operand	Operand of logical operator shall be effectively boolean

MisraC-12.7

Bitwise operators shall not be applied to operands whose underlying type is signed.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
bitop_on_enum	Bitwise operator on enum underlying type
bitop_on_signed	Bitwise operator on signed underlying type

MisraC-12.8

The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
report_nonconst_cases	Whether a message should be printed for non-constant cases missing a range-checking if statement	False
use_essential_type	Whether essential type should be computed for allowed value range.	False

Possible Messages

Name	Message
excessive_shift	Right operand of shift operator has value {}, but should be in range 0 to {}
possible_excessive_shift	Right operand of shift operator should be checked to be in range 0 to {}

MisraC-12.9

The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
check_literals	If True, integer literals are also disallowed as operands.	False

Possible Messages

Name	Message
unary_minus_on_unsigned	Unary minus applied to unsigned

MisraC-12.10

The comma operator shall not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
use_of_comma	Use of comma sequence operator.

MisraC-12.11

Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
const_expression_overflow	Overflow during evaluation of constant expression.

MisraC-12.12

The underlying bit representations of floating-point values shall not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
float_bit_representation	Use of bit representation of a float value.

MisraC-12.13

The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
forbid_all_operators	If True, forbids mixing with any kind of operator; else only with arithmetic operators.	False

Possible Messages

Name	Message
increment_mixed_with_operator	Increment or decrement mixed with other operators

MisraC-13.1

Assignment operators shall not be used in expressions that yield a Boolean value.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_in_extra_parens	Whether if $\{(x = y)\}$ should be allowed (note the extra parens)	False
allow_in_relational_operator	Whether an assignment is allowed as operand of a comparison or relational operator, e.g. $(x = y) != 0$	False

Possible Messages

Name	Message
assignment_inside_bool	Assignment inside boolean expression.

MisraC-13.2

Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
tolerate_endless_loops	If True, while{1} and for{;;} are accepted.	True

Possible Messages

Name	Message
implicit_zero_comparison	Use explicit test for zero.

MisraC-13.3

Floating-point expressions shall not be tested for equality or inequality.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
float_equality_comparison	Testing a float for (in)equality.
float_indirect_equality_comparison	Indirectly testing a float for (in)equality.

MisraC-13.4

The controlling expression of a for statement shall not contain any objects of floating type.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
float_loop_condition	Floating type in controlling expression of a for loop.

MisraC-13.5

The three expressions of a for statement shall be concerned only with loop control.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
loop_counter_model		MisraCLoopCountersFromContinue

Possible Messages

Name	Message
complex_for_loop_expr	For-loop expressions shall be concerned only with loop control

MisraC-13.6

Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
loop_counter_model		MisraCLoopCounters

Possible Messages

Name	Message
modified_loop_counter	Counting expression of loop is modified.

MisraC-13.7

Boolean operations whose results are invariant shall not be permitted.

Input: IR
Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
report_do_while_false	Whether do ... while[0] should be reported.	False
report_while_true	Whether while[1] ... should be reported.	False

Possible Messages

Name	Message
dead_false_branch	Condition is always true: Else branch is dead
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Parameter condition is always true: Else branch is dead
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Parameter comparison to NULL is always true: Else branch is dead
dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Parameter comparison to NULL is always false: Then branch is dead
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Parameter condition is always false: Then branch is dead
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Condition is always false: Then branch is dead
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Variable condition is always true: Else branch is dead
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Variable condition is always false: Then branch is dead
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead

MisraC-14.1

There shall be no unreachable code.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
report_dead_if_branch_with_const_literal_condition	Whether bodies of if-branches with constant literal conditions should be reported.	True

Possible Messages

Name	Message
unreachable_code	Unreachable code

MisraC-14.2

All non-null statements shall either (a) have at least one side-effect however executed, or (b) cause control flow to change.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
assume_effect_for_all_calls	Whether all calls should be seen as having a side-effect.	False
assume_effect_when_undefined	Whether all calls to undefined [external] functions should be seen as having a side-effect.	True
tolerate_void_cast	Whether a [void] cast is accepted.	False

Possible Messages

Name	Message
no_effect	Non-null statement without side-effect

MisraC-14.3

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allow_empty_macros	Whether a macro invocation before the ; is allowed if it expands to nothing.	False
allow_nonempty_macros	Whether a non-empty macro invocation before the ; is allowed.	False

Possible Messages

Name	Message
null_statement_not_isolated	Null statement not on a line by itself

MisraC-14.4

The goto statement shall not be used.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
goto	Do not use goto.

MisraC-14.5

The continue statement shall not be used.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
continue	Do not use continue.

MisraC-14.6

For any iteration statement there shall be at most one break statement used for loop termination.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
inspect_gotos	Whether gotos exiting the loop should be counted.	False

Possible Messages

Name	Message
multiple_loop_breaks	More than one break terminates loop.

MisraC-14.7

A function shall have a single point of exit at the end of the function.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
multiple_returns	Function shall have a single point of exit at the end of the function.

MisraC-14.8

The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
loop_body_not_compound	Loop body shall be a statement sequence.
switch_body_not_compound	switch body shall be a statement sequence.

MisraC-14.9

An if {expression} construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
dangerous_else	Block or if statement required as else branch.
dangerous_then	Block required as then branch.

MisraC-14.10

All if ... else if constructs shall be terminated with an else clause.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
defensive_else	Controls if the else branch of if-cascade's should contain a suitable comment or not be empty.	True

Possible Messages

Name	Message
non_defensive_else	Last else in the if-cascade must not be empty or contain a comment.
unterminated_if_cascade	Last if in the if-cascade has no else.

MisraC-15.0

The MISRA C switch syntax shall be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
acceptable_end_items	This set characterizes allowed last items at the end of a switch case	Exit_Switch
acceptable_middle_items	This set characterizes allowed last items in the middle of a switch case	('Exit_Switch', 'Nondefault_Case_Label')
allow_as_first	Whether to allow the "default" case as the first case in a switch.	False
allow_complete_enum	Whether a switch having a case for all enumerators is tolerated without a default.	True
allow_empty_statements	Whether empty statements at the end of switch-case should be ignored.	False
allow_fallthrough_comment	Whether a comment with text "fall" and "through" is allowed to mark fallthrough.	True
allow_fallthrough_macro	Whether a macro called __fallthrough is allowed to mark fallthrough.	True
allow_if	Whether an if containing acceptable items at the end of both branches is accepted.	False
allow_noreturn_function_calls	Whether a function call to a [[noreturn]] function is considered a valid termination.	False
comment	The comment that indicates an intentional fallthrough. Can be a list of comments.	[]
disallow_multiple_compound_statements	Whether there can be more than one toplevel compound statement.	True
disallowed_nested_statements	Statement types not allowed inside compound statements within case-clause body.	None
disallowed_toplevel_statements	Statement types not allowed in case-clause body on toplevel.	('Named_Label', 'Unconditional_Branch', 'Declaration_Statement')
minimum_switch_cases	The number of cases a switch statement should at least have.	1

Possible Messages

Name	Message
case_label_nested	A switch label shall only be used in a switch.
default_not_last	"default" clause is not last in switch.
empty_default	Default clause should contain a statement or comment.
empty_switch	Switch has no "case" clause.
forbidden_compound_statement_in_case	At most one compound statement is allowed on toplevel in case-clauses.
forbidden_nested_statement_in_case	Statement is not allowed inside compound statement in case-clauses.
forbidden_toplevel_statement_in_case	Statement is not allowed outside a compound statement in case-clauses.
minimum_switch_cases	Switch has to little "case" clauses.
missing_default	Switch has no "default" clause.
switch_body_not_compound	A switch statement shall be a well-formed switch statement.
switch_not_well_formed	Switch has a statement before the first case label.
switch_over_bool	Switch expression is effectively Boolean.
unterminated_case	This switch-case is not terminated with {}, last statement is {}.
unterminated_empty_case	This switch-case is empty, not terminated with {}.

MisraC-15.1

A switch label shall only be used when the most closely-enclosing compound-statement is the body of a switch-statement.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
case_label_nested	A switch label shall only be used in a switch.

MisraC-15.2

An unconditional break statement shall terminate every non-empty switch clause.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
acceptable_end_items	This set characterizes allowed last items at the end of a switch case	Exit_Switch
acceptable_middle_items	This set characterizes allowed last items in the middle of a switch case	{'Exit_Switch', 'Nondefault_Case_Label'}
allow_empty_statements	Whether empty statements at the end of switch-case should be ignored.	False
allow_fallthrough_comment	Whether a comment with text "fall" and "through" is allowed to mark fallthrough.	True
allow_fallthrough_macro	Whether a macro called __fallthrough is allowed to mark fallthrough.	True
allow_if	Whether an if containing acceptable items at the end of both branches is accepted.	False
allow_noreturn_function_calls	Whether a function call to a [[noreturn]] function is considered a valid termination.	False
comment	The comment that indicates an intentional fallthrough. Can be a list of comments.	[]

Possible Messages

Name	Message
unterminated_case	This switch-case is not terminated with {}, last statement is {}.
unterminated_empty_case	This switch-case is empty, not terminated with {}.

MisraC-15.3

The final clause of a switch statement shall be the default clause.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_as_first	Whether to allow the "default" case as the first case in a switch.	False
allow_complete_enum	Whether a switch having a case for all enumerators is tolerated without a default.	True

Possible Messages

Name	Message
default_not_last	"default" clause is not last in switch.
empty_default	Default clause should contain a statement or comment.
missing_default	Switch has no "default" clause.

MisraC-15.4

A switch expression shall not represent a value that is effectively Boolean.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
switch_over_bool	Switch expression is effectively Boolean.

MisraC-15.5

Every switch statement shall have at least one case clause.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
minimum_switch_cases	The number of cases a switch statement should at least have.	1

Possible Messages

Name	Message
empty_switch	Switch has no "case" clause.
minimum_switch_cases	Switch has too little "case" clauses.

MisraC-16.1

Functions shall not be defined with variable numbers of arguments.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
ignore_declarations	Allow declarations of functions with variable number of arguments (e.g. for existing library functions); only report definitions.	True
ignore_inherited	Do not report functions inheriting a variable number of arguments.	False

Possible Messages

Name	Message
ellipsis_parameter	Function shall not be defined with a variable number of arguments.

MisraC-16.2

Functions shall not call themselves, either directly or indirectly.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
function_cycle	Cycle of recursive functions
recursive_function	Function is directly recursive

MisraC-16.3

Identifiers shall be given for all of the parameters in a function prototype declaration.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
unnamed_parameter	Function parameter shall have a name

MisraC-16.4

The identifiers used in the declaration and definition of a function shall be identical.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
ignore_casing	Whether to ignore casing for the check.	False
include_redefinitions	Whether declarations of redefining methods should be considered together with the redefined ones.	False
inspect_only_defined_routines	If False, re-declarations of undefined functions are compared to earlier declarations as well.	True

Possible Messages

Name	Message
parameter_name_mismatch	Parameter name at definition differs from name at declaration

MisraC-16.5

Functions with no parameters shall be declared and defined with the parameter list void.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
implicit_function_return_type	Function shall have explicit return type
parameterless_func_without_void_param	Function with no parameters shall be declared with (void)

MisraC-16.6

The number of arguments passed to a function shall match the number of parameters.

Input: IR
Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
use_pointer_analysis	Whether to use pointer analysis to verify indirect calls. Note: pointer analysis can only be used if CONFIG.Run_IRAnalysis_Checks is enabled.	True

Possible Messages

Name	Message
wrong_argument_number	Number of arguments at function call does not match number of parameters

MisraC-16.7

A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
check_all_variables	Whether non-parameter variables should be checked as well.	False
ignore_forwarding_references	Whether to allow C++11 forwarding references (universal references) even if all template instances would also work with a const lvalue reference.	True
ignore_move_constructors	Whether to allow non-const rvalue references on constructors.	True
report_overrides	Whether to report violations on overrides. For virtual methods, this rule will report a violation only if the parameter can be marked const in all related methods in the inheritance hierarchy. Thus, the same violation is reported for each method in the inheritance hierarchy. You can set this option to False to report such violations only for the base methods.	True

Possible Messages

Name	Message
parameter_can_point_to_const	{ } can be declared as pointer/reference to const.

MisraC-16.8

All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
lambda_return_missing_value	Return without value in non-void lambda expression
missing_return	Non-void function needs return with value at end.
missing_return_in_lambda	Non-void lambda expression needs return with value at end.
return_missing_value	Return without value in non-void function

MisraC-16.9

A function identifier shall only be used with either a preceding &, or with a parenthesised parameter list, which may be empty.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allowed	Qualified names of functions of which it is allowed to take the address implicitly, e.g. C++ I/O manipulators	[`std::endl`, `std::flush`, `std::boolalpha`, `std::noboolalpha`, `std::showbase`, `std::noshowbase`, `std::showpoint`, `std::noshowpoint`, `std::showpos`, `std::noshowpos`, `std::skipws`, `std::noskipws`, `std::uppercase`, `std::nouppercase`, `std::unitbuf`, `std::nounitbuf`, `std::internal`, `std::left`, `std::right`, `std::dec`, `std::hex`, `std::oct`, `std::fixed`, `std::scientific`, `std::hexfloat`, `std::defaultfloat`, `std::ws`, `std::ends`, `std::resetiosflag`, `std::setiosflag`, `std::setbase`, `std::setfill`, `std::setprecision`, `std::setw`, `std::get_money`, `std::put_money`, `std::get_time`, `std::put_time`, `std::quoted`]

Possible Messages

Name	Message
implicit_routine_address	Use of function identifier without &

MisraC-16.10

If a function returns error information, then that error information shall be tested.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
inspect_template_instances	Whether calls in template instances should be reported.	False
relevant_functions	If provided, only calls to these functions are inspected.	[]

Possible Messages

Name	Message
discarded_error_with_entity	Integral return value (possibly error code) discarded.
discarded_error_without_entity	Integral return value (possibly error code) discarded.

MisraC-17.1

Pointer arithmetic shall only be applied to pointers that address an array or array element.

Input: IR

Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
arithmetic_on_nonarray	Pointer arithmetic on {1} operates on non-array target {0}
arithmetic_out_of_bounds	Pointer arithmetic on {1} creates pointer outside array bounds of {0}
out_of_bounds	Access into array is out of bounds
possible_indirect_out_of_bounds	Pointer-indirect access through {1} might be out of bounds accessing {0}
possible_out_of_bounds	Access into array might be out of bounds

MisraC-17.2

Pointer subtraction shall only be applied to pointers that address elements of the same array.

Input: IR

Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
possible_unrelated_ptr_subtraction	Subtracting possibly unrelated pointers
unrelated_ptr_subtraction	Subtracting unrelated pointers

MisraC-17.3

>, >=, <, <= shall not be applied to pointer types except where they point to the same array.

Input: IR

Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
possible_unrelated_ptr_comparison	Comparing possibly unrelated pointers
unrelated_ptr_comparison	Comparing unrelated pointers

MisraC-17.4

Array indexing shall be the only allowed form of pointer arithmetic.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
indexing_only_on_identifiers	Report array indexing on pointers only for variables (`ptr[i]`), not for other pointer expressions (e.g. `get_ptr[i]`). This option is meant to suppress the violations introduced by the BAUHAUS-12021 bugfix in version 6.9.6.	False

Possible Messages

Name	Message
array_indexing_on_pointer	Array indexing only allowed for arrays
pointer_arithmetic	Pointer arithmetic not allowed
pointer_increment_decrement	Pointer arithmetic not allowed

MisraC-17.5

The declaration of objects should contain no more than 2 levels of pointer indirection.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
max_levels	Maximum number of allowed pointer-indirection levels.	2

Possible Messages

Name	Message
too_many_pointer_levels	More than {} level{} of pointer indirection

MisraC-17.6

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
global_reference_to_local_var	Address of local variable escapes via global variable.
parameter_reference_to_local_var	Address of local variable escapes via parameter.
returning_reference_to_local_var	Returning reference(pointer to local variable.

MisraC-18.1

All structure and union types shall be complete at the end of a translation unit.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
incomplete_composite	Composite type is incomplete at end of translation unit '{unit_name}'.
incomplete_composite_in_multiple_units	Composite type is incomplete at end of {num_units} translation units (e.g. '{unit_name}').

MisraC-18.2

An object shall not be assigned to an overlapping object.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
assignment_to_overlapping	An object shall not be assigned to an overlapping object

MisraC-18.4

Unions shall not be used.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allow_tagged_unions	Whether tagged unions (nested in a struct that has an enum discriminator) should be allowed	False

Possible Messages

Name	Message
union	Unions shall not be used

MisraC-19.1

#include statements in a file should only be preceded by other preprocessor directives or comments.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allow_extern_blocks	Whether to allow includes in `extern` blocks and not report a violation.	False

Possible Messages

Name	Message
include_after_code	#include should only be preceded by comments or preprocessor directives

MisraC-19.2

Non-standard characters should not occur in header file names in #include directives.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
check_quotation_mark	Whether system includes should be checked for containing a `".`	True
forbidden	The substrings to check for. `."` will be added for system-includes.	set(['/*', '""', '\\'])

Possible Messages

Name	Message
nonstandard_include_character	Non-standard character in #include directive

MisraC-19.3

The #include directive shall be followed by either a <filename> or "filename" sequence.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
nonstandard_include_form	#include shall be either <filename> or "filename"

MisraC-19.4

C macros shall only expand to a braced initialiser, a constant, a string literal, a parenthesised expression, a type qualifier, a storage class class specifier, or a do-while-zero construct.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
forbidden_c_macro_definition	Macro may only expand to certain types of expressions

MisraC-19.5

Macros shall not be #define'd or #undef'd within a block.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
local_macro	#define or #undef in block

MisraC-19.6

#undef shall not be used.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
undef	#undef shall not be used

MisraC-19.7

A function should be used in preference to a function-like macro.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
function_macro_definition	Prefer function over function-like macro

MisraC-19.8

A function-like macro shall not be invoked without all of its arguments.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
missing_function_macro_argument	Too few arguments in invocation of macro

MisraC-19.9

Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
directives	Preprocessing directives to check for in macro arguments.	('#if', '#ifdef', '#ifndef', '#elif', '#else', '#endif', '#pragma', '#warning', '#error', '#line', '#include', '#include_next', '#ident', '#region', '#endregion', '#asm', '#endasm', '#define', '#undef')

Possible Messages

Name	Message
pp_directive_as_macro_arg	Macro invocation argument looks like preprocessing directive

MisraC-19.10

In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
unsafe_macro_param	Macro parameter not enclosed in parentheses

MisraC-19.11

All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
undefined_macro_used	Zero used for undefined preprocessing identifier

MisraC-19.12

There shall be at most one occurrence of the # or ## operators in a single macro definition.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
report_only_concatenated_stringized	If True, only macros where a parameter is stringized and the result used for token concatenation are reported.	False

Possible Messages

Name	Message
concatenated_stringized	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator
more_than_one_macro_hash	Only one # or ## allowed in macro definition

MisraC-19.13

The # and ## operators should not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
hash_in_macro	The # and ## operators should not be used.

MisraC-19.14

The defined preprocessor operator shall only be used in one of the two standard forms.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
nonstandard_defined	Non-standard use of defined operator

MisraC-19.15

Precautions shall be taken in order to prevent the contents of a header file being included twice.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
macro_name_restrictions	Python iterable of functions with parameters {file, define, macro} to perform additional checks on the macro used as include guard in the given file. Should return None if macro is accepted, else a message to print for the violation. The parameters are: PIR File node for the header, token for the macro #define or #pragma once, and the name of the macro or the string 'pragma once'.	None

Possible Messages

Name	Message
include_guard_missing	Missing include guard.
include_guard_partial	Include guard does not cover complete file.
include_guard_with_wrong_define	Bad include guard: #define uses wrong macro, should be {}.
include_guard_without_define	Incomplete include guard: missing #define {}.

MisraC-19.16

Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
malformed_pp_directive	Malformed or unknown preprocessing directive

MisraC-19.17

All #else, #elif and #endif preprocessing directives shall reside in the same file as the #if or #ifdef directive to which they are related.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
missing_endif	The #endif for this directive is missing
missing_if	The #if for this directive is missing

MisraC-20.1

Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
additional_reserved_identifiers	Names listed here are seen as violations as well.	set(['assert', 'errno', 'defined'])
allow_reserved_identifier_as_include_guard	If True, macros used for include guards are not checked for being a reserved identifier.	False
check_for_keyword	Whether macro names being keywords should be reported as well.	True

Possible Messages

Name	Message
macro_having_reserved_name	Definition of reserved identifier or standard library element
undef_of_reserved_name	#undef of reserved identifier or standard library element

MisraC-20.2

The names of standard library macros, objects and functions shall not be reused.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_function_declarations	Whether nondefining function declarations with library names are allowed	False
check_locals	Whether parameters and local variables should also be checked	True
check_reserved_enum_identifier	Whether enumerator names should be checked to use a reserved identifier	False
check_reserved_function_identifier	Whether function names should be checked to use a reserved identifier	False
check_reserved_macro_identifier	Whether to report reserved names (e.g. names starting with underscore).	False
check_reserved_type_identifier	Whether type names should be checked to use a reserved identifier	False
check_reserved_variable_identifier	Whether variable names should be checked to use a reserved identifier	False
report_fields	Whether fields using a library name should be reported.	False

Possible Messages

Name	Message
enumerator_having_libname	The names of standard library macros, objects and functions, as well as reserved identifiers, shall not be reused.
field_having_libname	The names of standard library macros, objects and functions shall not be reused.
macro_having_libname	The names of standard library macros, objects and functions shall not be reused.
routine_having_libname	The names of standard library macros, objects and functions, as well as reserved identifiers, shall not be reused.
type_having_libname	The names of standard library macros, objects and functions shall not be reused.
variable_having_libname	The names of standard library macros, objects and functions, as well as reserved identifiers, shall not be reused.

MisraC-20.3

The validity of values passed to library functions shall be checked.

Input: IR

Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
argument_checks	Configuration of {dis}allowed values in addition to what iranalysis.config provides. This is a dict key -> dict with keys 'Functions', 'Parameter', and 'Allowed'/'Disallowed'. The parameter number counting starts at 0. Arguments passed in for this parameters are checked against the value specification given with '{Dis}allowed'.	dict{...}

Possible Messages

Name	Message
argumentViolation	Argument not within allowed values
possibleArgumentViolation	Argument possibly not within allowed values

MisraC-20.4

Dynamic heap memory allocation shall not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_base_classes_from	List of path globbing patterns to identify the location of base classes which (together with classes derived from them) should not be reported. For example, a globbing pattern like '/usr/include/*/qt*/Qt*' would allow classes derived from Qt classes if your Qt installation resides there.	[]

Possible Messages

Name	Message
cpp_new_delete	Builtin dynamic memory management operator used.
stdlib_memory_function_call	Dynamic memory management function from <stdlib.h> is called.

MisraC-20.5

The error indicator errno shall not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
symbol_header	Name of the header file of which the symbols should not be used.	{'errno', 'stdlib'}
symbols	Names of symbols which are forbidden.	{'errno'}
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

MisraC-20.6

The macro offsetof, in library <stddef.h>, shall not be used.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
symbol_header	Name of the header file of which the symbols should not be used.	stddef
symbols	Names of symbols which are forbidden.	{'offsetof'}
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

MisraC-20.7

The setjmp macro and the longjmp function shall not be used.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
symbol_header	Name of the header file of which the symbols should not be used.	setjmp
symbols	Names of symbols which are forbidden.	{'setjmp', 'longjmp'}
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

MisraC-20.8

The signal handling facilities of <signal.h> shall not be used.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header [used for symbols like NULL and size_t which are defined in multiple headers]	['NULL', 'size_t']
header	Name of the header file which should not be used.	signal

Possible Messages

Name	Message
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.

MisraC-20.9

The input/output library <stdio.h> shall not be used in production code.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header [used for symbols like NULL and size_t which are defined in multiple headers]	['NULL', 'size_t']
header	Name of the header file which should not be used.	stdio

Possible Messages

Name	Message
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.

MisraC-20.10

The library functions atof, atoi and atol from library <stdlib.h> shall not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
symbol_header	Name of the header file of which the symbols should not be used.	stdlib
symbols	Names of symbols which are forbidden.	['atof', 'atoi', 'atol']
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

MisraC-20.11

The library functions abort, exit, getenv, and system from library <stdlib.h> shall not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
symbol_header	Name of the header file of which the symbols should not be used.	stdlib
symbols	Names of symbols which are forbidden.	['abort', 'exit', 'getenv', 'system']
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

MisraC-20.12

The time handling functions of library <time.h> shall not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header (used for symbols like NULL and size_t which are defined in multiple headers)	['NULL', 'size_t']
header	Name of the header file which should not be used.	time

Possible Messages

Name	Message
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.

MisraC-21.1

Minimisation of run-time failures shall be ensured by the use of at least one of (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.

Input: IR

Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
abstract_interpretation_div_by_zero	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
abstract_interpretation_overflow	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
argument_checks	Configuration of [dis]allowed values in addition to what iranalysis.config provides. This is a dict key -> dict with keys 'Functions', 'Parameter', and 'Allowed'/'Disallowed'. The parameter number counting starts at 0. Arguments passed in for this parameters are checked against the value specification given with '[Dis]allowed'.	dict{...}
execute_checks_covered_in_other_rules	Whether iranalysis-based checks covered by other Misra-C:2004 rules should be executed here as well.	False
forbidden	Dict which lists forbidden operations per resource. The mapping gives each case a description which maps to a dict for key "Forbidden_Functions", "Mode_Parameter", "Mode".	dict{...}
modes	Describes when an allocating call opens a file in read or write mode. This is a dict resource -> dict with keys "Resource_Parameter", "Check_Write_Write", and "Modes" (being a dict with key "Read" and "Write")	dict{...}
perform_global_analysis	Whether global analysis (using iranalysis) should be done, or just a local analysis.	True
report_dead_initializations	Whether initialisations may be reported as dead.	True
report_do_while_false	Whether do ... while[0] should be reported.	True
report_while_true	Whether while[1] ... should be reported.	True
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict{...}

Possible Messages

Name	Message
argumentViolation	Argument not within allowed values
arithmetic_out_of_bounds	Pointer arithmetic on {1} creates pointer outside array bounds of {0}
asgn_not_an_enumerator	Assignment of value which does not correspond to an enumerator
asgn_out_of_enum_range	Assignment of value outside the range of the enumerators
cast_overflow	Cast on result of arithmetic computation may cause overflow
cast_truncate	Cast may truncate value
cast_underflow	Cast on result of arithmetic computation may cause underflow
conditional_unused_def	Result of assignment is not used along some path(s)
dead_false_branch	Condition is always true: Else branch is dead
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Parameter condition is always true: Else branch is dead
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Parameter comparison to NULL is always true: Else branch is dead
dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Parameter comparison to NULL is always false: Then branch is dead
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Parameter condition is always false: Then branch is dead
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead

dead_true_branch	Condition is always false: Then branch is dead
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Variable condition is always true: Else branch is dead
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Variable condition is always false: Then branch is dead
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead
division_by_zero	Division by zero
double_free	Dynamic memory released here was already released earlier
escaping_address	Escaping address of local variable
forbidden_operation	The way in which this resource was allocated forbids this operation
init_used_in_other_isr	Initialization is only used in some interrupt handler
memory_leak	Call allocates leaking memory
mode_conflict	Same file used for both reading and writing
modulo_by_zero	Modulo by zero
null_deref	Pointer is NULL at dereference
out_of_bounds	Access into array is out of bounds
overflow	Arithmetic computation may cause overflow
pass_as_pointer_to_const_param	Passing uninitialized variable by pointer as function parameter with pointer-to-const type
possible_argumentViolation	Argument possibly not within allowed values
possible_division_by_zero	Possible division by zero
possible_double_free	Dynamic memory released here possibly already released earlier
possible_indirect_out_of_bounds	Pointer-indirect access through {1} might be out of bounds accessing {0}
possible_memory_leak	Call allocates possibly leaking memory
possible_mode_conflict	Same file possibly used for both reading and writing
possible_modulo_by_zero	Possible modulo by zero
possible_null_deref	Pointer may be NULL at dereference
possible_out_of_bounds	Access into array might be out of bounds
possible_stack_free	{0} possibly released by call to {0} is a stack object
possible_uninit	Use of possibly uninitialized variable
possible_unrelated_ptr_comparison	Comparing possibly unrelated pointers
possible_unrelated_ptr_subtraction	Subtracting possibly unrelated pointers
possible_use_after_free	Dynamic memory possibly used after it was previously released
possible_wrong_release	Resource possibly released using wrong function (allocation used {0})
possiblyEscapingAddress	Possibly escaping address of local variable (as target of {1})
possiblyForbiddenOperation	The way in which this resource was allocated possibly forbids this operation
stack_free	{0} released by call to {0} is a stack object
underflow	Arithmetic computation may cause underflow

uninit	Use of uninitialized variable
unrelated_ptr_comparison	Comparing unrelated pointers
unrelated_ptr_subtraction	Subtracting unrelated pointers
unused_def	Result of assignment is not used
unused_init	Unused initialization
use_after_free	Dynamic memory used after it was previously released
used_in_other_isr	Result of assignment is only used in some interrupt handler
wrong_release	Resource released using wrong function {allocation used {0}}

Rules in Group MisraC++

MisraC++-0.1.1

There shall be no unreachable code.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
report_dead_if_branch_with_const_literal_condition	Whether bodies of if-branches with constant literal conditions should be reported.	True

Possible Messages

Name	Message
unreachable_code	Unreachable code

MisraC++-0.1.2

A project shall not contain infeasible paths.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
report_do_while_false	Whether do ... while(0) should be reported.	True
report_while_true	Whether while(1) ... should be reported.	True

Possible Messages

Name	Message
dead_false_branch	Condition is always true: Else branch is dead
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Parameter condition is always true: Else branch is dead
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Parameter comparison to NULL is always true: Else branch is dead
dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Parameter comparison to NULL is always false: Then branch is dead
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Parameter condition is always false: Then branch is dead
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Condition is always false: Then branch is dead
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Variable condition is always true: Else branch is dead
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Variable condition is always false: Then branch is dead
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead

MisraC++-0.1.3

A project shall not contain unused variables.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
ignore_types	Variables of types named here are ignored in this check. Globbing patterns are supported.	[]
only_check_unit_locals	Whether only global static variables and local variables should be checked.	False
report_global_constants	Whether unused global constants should be reported.	False
report_undefined_variables	Whether only-declared variables should be reported.	True
treat_initialization_as_use	Whether an explicit initialization should be considered a use of the variable.	True
treat_side_effect_constructors_as_use	Whether variables should be seen as used if they are of a class type and initialized through a call to a constructor having a side-effect, e.g. std::lock_guard	False

Possible Messages

Name	Message
unused_field	Unused field
unused_variable	Unused variable

MisraC++-0.1.4

A project shall not contain non-volatile POD variables having only one use.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_if_address_taken	Allow using a variable only once if that use involves taking the address of the variable.	False
report_fields	Select whether fields used only once should be reported as well.	True

Possible Messages

Name	Message
field_referenced_only_once	{ } referenced only once
unreferenced_initialized_field	{ } initialized but not referenced
unreferenced_initialized_variable	{ } initialized but not referenced
variable_used_only_once	{ } referenced only once

MisraC++-0.1.5

A project shall not contain unused type declarations.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
unused_type	Unused type declaration

MisraC++-0.1.6

A project shall not contain instances of non-volatile variables being given values that are never subsequently used.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
report_dead_initializations	Whether initialisations may be reported as dead.	True

Possible Messages

Name	Message
conditional_unused_def	Result of assignment is not used along some path(s)
init_used_in_other_isr	Initialization is only used in some interrupt handler
unused_def	Result of assignment is not used
unused_init	Unused initialization
used_in_other_isr	Result of assignment is only used in some interrupt handler

MisraC++-0.1.7

The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.

Input: IR
Source languages: C, C++

Configuration

Name	Explanation	Value
allowed_functions	Calls to these functions are ignored.	frozenset(['strncat', 'memmove', 'memset', 'strcat', 'strcpy', 'memcpy', 'strncpy'])
inspect_template_instances	Whether calls in template instances should be reported.	False

Possible Messages

Name	Message
discarded_return_with_entity	Return value of function discarded.
discarded_return_without_entity	Return value of function discarded.

MisraC++-0.1.8

All functions with void return type shall have external side effect(s).

Input: IR
Source languages: C, C++

Configuration

Name	Explanation	Value
assume_effect_when_undefined	Whether all calls to undefined (external) functions should be seen as having a side-effect.	True
exceptions	Names of functions that should be excluded from the check.	main
exclude_constructors	If True, tolerate constructors with no side-effect.	False
exclude_virtual_destructors	If True, tolerate virtual destructors with no side-effect.	False

Possible Messages

Name	Message
void_func_without_side_effect	Void function has no external side-effect

MisraC++-0.1.9

There shall be no dead code.

Input: IR
Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
allow_void_var	Whether {void}var; should be allowed or reported.	True
assume_effect_for_all_calls	Whether all calls should be seen as having a side-effect.	False
assume_effect_when_undefined	Whether all calls to undefined (external) functions should be seen as having a side-effect.	True
report_dead_initializations	Whether initialisations may be reported as dead.	True
report_do_while_false	Whether do ... while(0) should be reported.	True
report_while_true	Whether while(1) ... should be reported.	True
tolerate_void_cast	Whether a {void} cast is accepted.	True

Possible Messages

Name	Message
conditional_unused_def	Result of assignment is not used along some path(s) (disabled)
dead_false_branch	Redundant code, condition is always true
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Redundant code, parameter condition is always true
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Redundant code, parameter comparison to NULL is always true
dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Redundant code, parameter comparison to NULL is always false
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Redundant code, parameter condition is always false
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Redundant code, condition is always false
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Redundant code, variable condition is always true
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Redundant code, variable condition is always false
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead
init_used_in_other_isr	Initialization is only used in some interrupt handler
no_effect	Non-null statement without side-effect
removable_declaration	Declaration can be removed
removable_statement	Statement can be removed
unused_def	Dead (redundant) code: result of assignment is not used
unused_init	Unused initialization
used_in_other_isr	Result of assignment is only used in some interrupt handler

MisraC++-0.1.10

Every defined function shall be called at least once.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
ignore_copy_assignment_operators	Whether uncalled copy assignment operators are allowed. Other rules like the "rule of three" might require copy assignment operators as part of good class design, even if there are currently no callers of the copy assignment operator.	False
ignore_copy_constructors	Whether uncalled copy constructors are allowed. Other rules like the "rule of three" might require copy constructors as part of good class design, even if there are currently no callers of the copy constructor.	False
only_check_internal_or_private_methods	Whether only function with internal linkage or private functions should be checked.	False
only_check_static_or_private_methods	Whether only static functions or private functions should be checked.	False
only_check_unit_locals	Whether only global static functions should be checked.	False

Possible Messages

Name	Message
defined_function_not_called	Defined function is not called

MisraC++-0.1.11

There shall be no unused parameters (named or unnamed) in non-virtual functions.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
inspect_nonvirtual_functions	Whether parameters of non-virtual functions should be checked.	True
inspect_virtual_functions	Whether virtual functions should be checked.	False
report_unnamed_parameters	Whether unnamed unused parameters should be reported as well.	True

Possible Messages

Name	Message
unused_parameter	Unused parameter of non-virtual function

MisraC++-0.1.12

There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
inspect_nonvirtual_functions	Whether non-virtual functions should be checked.	False
inspect_virtual_functions	Whether parameters of virtual functions should be checked. Violations will be reported in the base class when none of the derived classes make use of the parameter.	True
report_unnamed_parameters	Whether unnamed unused parameters should be reported as well.	True

Possible Messages

Name	Message
unused_parameter	Unused parameter of virtual function

MisraC++-0.2.1

An object shall not be assigned to an overlapping object.

Input: IR
Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
assignment_to_overlapping	An object shall not be assigned to an overlapping object

MisraC++-0.3.1

Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.

Input: IR
Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
abstract_interpretation_div_by_zero	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
abstract_interpretation_overflow	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
argument_checks	Configuration of [dis]allowed values in addition to what iranalysis.config provides. This is a dict key -> dict with keys 'Functions', 'Parameter', and 'Allowed'/'Disallowed'. The parameter number counting starts at 0. Arguments passed in for this parameters are checked against the value specification given with '{Dis}allowed'.	dict(...)
execute_checks_covered_in_other_rules	Whether iranalysis-based checks covered by other Misra-C++:2008 rules should be executed here as well.	False
forbidden	Dict which lists forbidden operations per resource. The mapping gives each case a description which maps to a dict for key "Forbidden_Functions", "Mode_Parameter", "Mode".	dict(...)
modes	Describes when an allocating call opens a file in read or write mode. This is a dict resource -> dict with keys "Resource_Parameter", "Check_Write_Write", and "Modes" (being a dict with key "Read" and "Write")	dict(...)
perform_global_analysis	Whether global analysis (using iranalysis) should be done, or just a local analysis.	True
report_dead_initializations	Whether initialisations may be reported as dead.	True
report_do_while_false	Whether do ... while[0] should be reported.	True
report_while_true	Whether while[1] ... should be reported.	True
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)

Possible Messages

Name	Message
argumentViolation	Argument not within allowed values
arithmetic_out_of_bounds	Pointer arithmetic on {1} creates pointer outside array bounds of {0}
asgn_not_an_enumerator	Assignment of value which does not correspond to an enumerator
asgn_out_of_enum_range	Assignment of value outside the range of the enumerators
cast_overflow	Cast on result of arithmetic computation may cause overflow
cast_truncate	Cast may truncate value

cast_underflow	Cast on result of arithmetic computation may cause underflow
conditional_unused_def	Result of assignment is not used along some path(s)
dead_catch	Dead exception handler
dead_false_branch	Condition is always true: Else branch is dead
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Parameter condition is always true: Else branch is dead
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Parameter comparison to NULL is always true: Else branch is dead
dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Parameter comparison to NULL is always false: Then branch is dead
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Parameter condition is always false: Then branch is dead
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Condition is always false: Then branch is dead
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Variable condition is always true: Else branch is dead
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Variable condition is always false: Then branch is dead
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead
division_by_zero	Division by zero
double_free	Dynamic memory released here was already released earlier
escaping_address	Escaping address of local variable
forbidden_operation	The way in which this resource was allocated forbids this operation
init_used_in_other_isr	Initialization is only used in some interrupt handler
memory_leak	Call allocates leaking memory
mode_conflict	Same file used for both reading and writing
modulo_by_zero	Modulo by zero
null_deref	Pointer is NULL at dereference
out_of_bounds	Access into array is out of bounds
overflow	Arithmetic computation may cause overflow
pass_as_pointer_to_const_param	Passing uninitialized variable by pointer as function parameter with pointer-to-const type
possible_argumentViolation	Argument possibly not within allowed values
possible_division_by_zero	Possible division by zero
possible_double_free	Dynamic memory released here possibly already released earlier
possible_indirect_out_of_bounds	Pointer-indirect access through {1} might be out of bounds accessing {0}
possible_memory_leak	Call allocates possibly leaking memory

possible_mode_conflict	Same file possibly used for both reading and writing
possible_modulo_by_zero	Possible modulo by zero
possible_null_deref	Pointer may be NULL at dereference
possible_out_of_bounds	Access into array might be out of bounds
possible_stack_free	{ } possibly released by call to { } is a stack object
possible_uninit	Use of possibly uninitialized variable
possible_unrelated_ptr_comparison	Comparing possibly unrelated pointers
possible_unrelated_ptr_subtraction	Subtracting possibly unrelated pointers
possible_use_after_free	Dynamic memory possibly used after it was previously released
possible_wrong_release	Resource possibly released using wrong function (allocation used {0})
possiblyEscapingAddress	Possibly escaping address of local variable (as target of {1})
possiblyForbiddenOperation	The way in which this resource was allocated possibly forbids this operation
stack_free	{ } released by call to { } is a stack object
underflow	Arithmetic computation may cause underflow
uninit	Use of uninitialized variable
unrelated_ptr_comparison	Comparing unrelated pointers
unrelated_ptr_subtraction	Subtracting unrelated pointers
unused_def	Result of assignment is not used
unused_init	Unused initialization
use_after_free	Dynamic memory used after it was previously released
used_in_other_isr	Result of assignment is only used in some interrupt handler
wrong_release	Resource released using wrong function (allocation used {0})

MisraC++-0.3.2

If a function generates error information, then that error information shall be tested.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
inspect_template_instances	Whether calls in template instances should be reported.	False
relevant_functions	If provided, only calls to these functions are inspected.	[]

Possible Messages

Name	Message
discarded_error_with_entity	Integral return value (possibly error code) discarded.
discarded_error_without_entity	Integral return value (possibly error code) discarded.

MisraC++-0.4.2

Use of floating-point arithmetic shall be documented.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
float_arithmetic	Use of floating-point arithmetic

MisraC++-1.0.1

All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
show_include_path	If True, violations regarding nested includes print the #include path.	False
standards	List of allowed frontend command-line options to select language standard.	['--strict', '-A']

Possible Messages

Name	Message
access_control	Number of access control declarations is higher than translation limit (which is {})
at_quick_exit_functions	Number of functions registered with at_quick_exit() is higher than translation limit (which is {})
atexit_functions	Number of functions registered with atexit() is higher than translation limit (which is {})
base_classes	Number of direct and indirect base classes is higher than translation limit (which is {})
call_arguments	Number of arguments is higher than translation limit (which is {})
catch_handlers	Number of catch handlers is higher than translation limit (which is {})
chars_in_line	Number of characters in logical line is higher than translation limit (which is {})
chars_in_string	Number of characters in string is higher than translation limit (which is {})
cpp_members	Number of members is higher than translation limit (which is {})
data_members	Number of data members is higher than translation limit (which is {})
direct_bases	Number of direct base classes is higher than translation limit (which is {})
enumerators	Number of enumerators is higher than translation limit (which is {})
expr_in_core_constant	Number of full expressions evaluated within a core constant expression is higher than translation limit (which is {})
external_identifiers	Number of external identifiers in this unit is higher than translation limit (which is {})
final_virtual_functions	Number of final overriding virtual functions is higher than translation limit (which is {})
friends	Number of friend declarations is higher than translation limit (which is {})
function_parameters	Number of parameters is higher than translation limit (which is {})
macro_arguments	Number of macro arguments is higher than translation limit (which is {})
macro_parameters	Number of macro parameters is higher than translation limit (which is {})
macros	Number of macro definitions in this unit is higher than translation limit (which is {})
member_initializers	Number of member initializers is higher than translation limit (which is {})
nested_blocks	Number of nested blocks is higher than translation limit (which is {})
nested_composites	Nesting level of structs and unions is higher than translation limit (which is {})

nested_declarators	Number of pointer/array/function declarators is higher than translation limit (which is {})
nested_externals	Number of nested external specifications is higher than translation limit (which is {})
nested_includes	Nesting level of #includes is higher than translation limit (which is {})
nested_preprocessor_ifs	Nesting level of #if is higher than translation limit (which is {})
nested_template_instantiations	Number of nested template instantiations is higher than translation limit (which is {})
object_size	Number of bytes in object is higher than translation limit (which is {})
parenthesized_declarators	Nesting level of parenthesized declarators is higher than translation limit (which is {})
parenthesized_expressions	Nesting level of parenthesized subexpressions is higher than translation limit (which is {})
placeholders	Number of placeholders is higher than translation limit (which is {})
recursive_constexpr	Number of recursive constexpr function invocations is higher than translation limit (which is {})
scope_qualifications	Number of scope qualifications is higher than translation limit (which is {})
significant_chars_external	Number of significant initial characters for external identifier is higher than translation limit (which is {})
significant_chars_internal	Number of significant initial characters for internal identifier is higher than translation limit (which is {})
static_members	Number of static members is higher than translation limit (which is {})
std_compile	You have to compile with option {}
switch_cases	Number of cases is higher than translation limit (which is {})
template_params	Number of template parameters is higher than translation limit (which is {})
throw_spec	Number of throw specifications is higher than translation limit (which is {})
variables_in_block	Number of variables in block is higher than translation limit (which is {})
virtual_bases	Number of virtual base classes is higher than translation limit (which is {})

MisraC++-2.3.1

Trigraphs shall not be used.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_in_comments	Allow the use of trigraphs in comments	True

Possible Messages

Name	Message
trigraph_use	Use of trigraph.

MisraC++-2.5.1

Digraphs should not be used.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
digraph_use	Digraph used.

MisraC++-2.7.1

The character sequence /* shall not be used within a C-style comment.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
check_cpp_comments	Whether to look for /* in C++-style comments as well.	False
files_to_check	Files to apply this check to (Primary_File / User_Include_File / System_Include_File)	{'Primary_File', 'User_Include_File'}

Possible Messages

Name	Message
nested_c_comment	C-style comment containing /* sequence.

MisraC++-2.7.2

Sections of code shall not be "commented out" using C-style comments.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
files_to_check	Files to be checked, e.g. Primary_File or File.	Primary_File
normal_text	Tokens that indicate non-code comments. This is the first step of checking a comment. If any of these words exist in the comment text, it is regarded as not containing code.	{' and ', ' or ', ' but ', ' now ', ' to ', ' is ', ' are ', ' only ', ' be ', ' has ', ' the ', ' with ', ' because ', ' when ', ' oder ', ' und ', '://', '###', 'AXIVION', '++++', '----', '=====')}
normal_text_regex	Regular expression to match normal non-code text. This is the second step of checking a comment. If this regex matches the comment text, the comment is regarded as not containing code.	\b[\w\d_]+\s+[\w\d_]+\s+[\w\d_]+\b
suspicious	Tokens that indicate code in comments. This is the final step of checking a comment. If the first two steps did not exclude the comment from further inspection, the comment text is searched for any of these suspicious tokens. The comment is regarded as containing code, if any suspicious token is found.	{' = ', ' == ', ' >= ', ' <= ', '[', ']', '::', ' ->', ' ->*', ' ::*', 'if', 'if()', 'while', 'for', 'if', 'while', 'for', '#pragma', '#else', '#endif', '#if', '#include', '++', '--'}

Possible Messages

Name	Message
code_in_c_comment	Sections of code shall not be commented out using C-style comments.

MisraC++-2.7.3

Sections of code should not be "commented out" using C++ comments.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
files_to_check	Files to be checked, e.g. Primary_File or File.	Primary_File
normal_text	Tokens that indicate non-code comments. This is the first step of checking a comment. If any of these words exist in the comment text, it is regarded as not containing code.	{' and ', ' or ', ' but ', ' now ', ' to ', ' is ', ' are ', ' only ', ' be ', ' has ', ' the ', ' with ', ' because ', ' when ', ' oder ', ' und ', ' :// ', '###', 'AXIVION', '++++', '----', '===='}
normal_text_regex	Regular expression to match normal non-code text. This is the second step of checking a comment. If this regex matches the comment text, the comment is regarded as not containing code.	\b[\w\d_]+\s+[\w\d_]+\s+[\w\d_]+\b
suspicious	Tokens that indicate code in comments. This is the final step of checking a comment. If the first two steps did not exclude the comment from further inspection, the comment text is searched for any of these suspicious tokens. The comment is regarded as containing code, if any suspicious token is found.	{' = ', ' == ', ' >= ', ' <= ', '[', ']', ' :: ', ' ->', ' ->*', ' ::*', 'if', 'while', 'for', 'if (', 'while (', 'for (', '#pragma', '#else', '#endif', '#if', '#include', '++', '--')}

Possible Messages

Name	Message
code_in_cpp_comment	Sections of code should not be commented out using C++ comments.

MisraC++-2.10.1

Different identifiers shall be typographically unambiguous.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
distinguish_c_name_spaces	Whether to allow similar identifiers in different C name spaces (type tag/label/member/ordinary identifier)	True
distinguish_macro_names	Whether to allow macros to same similar names as other identifiers.	False
distinguish_type_names	Whether to allow types to have similar names as other identifiers. For example, a variable declaration "MyClass myClass;" is allowed by this option. Note: unlike the option 'distinguish_c_name_spaces', this option also distinguishes typedef names, not just struct/class tag names.	False
exclude_system_headers	Whether to exclude identifiers appearing in system headers	False
normalizations	Which pairs of characters should be seen as ambiguous	[('0', 'O'), ('1', 'I'), ('l', 'L'), ('i', 'I'), ('5', 'S'), ('2', 'Z'), ('8', 'B'), ('rn', 'm'), ('n', 'h'), ('_', '')]

Possible Messages

Name	Message
ambiguous	Identifiers are typographically ambiguous
casing	Identifiers only differ in casing

MisraC++-2.10.2

Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_in_classes	Whether identifiers in local classes can hide surrounding ones	False
allow_in_lambdas	Whether identifiers in lambdas can hide surrounding ones	False
allow_in_local_classes	Whether identifiers in local classes can hide surrounding ones	False
allow_in_namespaces	Whether identifiers in namespaces can hide surrounding ones	False
allow_parameter_in_function_declaration	Whether identifiers of parameters in function declarations may hide surrounding ones, given the function declaration uses a different identifier for the same parameter that does not hide a surrounding one.	False
check_inheritance_hiding	Whether to check for hiding by inheritance.	False
check_nested_classes	Whether symbols from inner classes should be reported as hiding symbols of outer classes.	False
distinguish_name_spaces	Whether C name spaces should be distinguished.	False
hiding_entities	LIR node types to consider symbols that hide other symbols.	{'Variable', 'Typedef_Type', 'Field', 'Parameter'}
maxlen	Number of significant characters (or None)	None
tolerate_constructor_parameters_hiding_fields	Allow constructor parameters to hide fields	False
tolerate_function_prototype_argument_hiding	Whether to tolerate arguments of a function prototype, that itself is a function argument, that have the same identifier as a argument of the outer function. If 'True' is given, the rule will still report cases, where argument identifiers match up and the function prototype refers to arguments of the outer function (e.g. by `decltype()`).	False
tolerate_macro_local_identifier	Whether an identifier being local to a macro is allowed to hide identifiers from outside the macro	False
unchecked_types	Deprecated legacy option, kept to avoid breaking old configurations.	{'Routine_Call', 'Stack_Object_Definition'}

Possible Messages

Name	Message
hiding	{ } hides { }

MisraC++-2.10.3

A typedef name (including qualification, if any) shall be a unique identifier.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
tolerate_macros	Whether #define and #undef using the typedef's name is allowed	False
tolerate_typedef_entity	Whether the entity forming the typedef's underlying type can have the same name.	False

Possible Messages

Name	Message
reused_typedef	Typedef name reused.

MisraC++-2.10.4

A class, union or enum name (including qualification, if any) shall be a unique identifier.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
check_types_only_with_external_linkage	Whether the types should only be checked when they have external linkage	False
tolerate_typedef	Whether a typedef to the tag may have the same name.	False

Possible Messages

Name	Message
composite_defined_multiple_times	Composite type has multiple definitions.
enum_defined_multiple_times	Enumeration type has multiple definitions.
reused_tag	Tag name reused.

MisraC++-2.10.5

The identifier name of a non-member object or function with static storage duration should not be reused.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_static_no_linkage	Whether to allow same identifiers for static objects without linkage (e.g. function local static objects)	False
func_filter	Restricts which functions are considered.	None
use_unqualified_names	If set to True, ns1::Type is considered equal to ns2::Type.	True
var_filter	Restricts which variables are considered.	None

Possible Messages

Name	Message
reused_global_function_name	Name of object or function with static storage duration reused.
reused_global_variable_name	Name of object or function with static storage duration reused.

MisraC++-2.10.6

If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
invert_findings	Whether to invert primary and secondary SLocs for violations	False

Possible Messages

Name	Message
reused_type	Type name reused in same scope

MisraC++-2.13.1

Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allowed	List of allowed characters after backslash.	"\"?\\abfnrtv

Possible Messages

Name	Message
nonstandard_escape_sequence	Use of non-standard escape sequence.

MisraC++-2.13.2

Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
octal_escape_sequence	Use of octal escape sequence.
octal_literal	Use of octal literal.

MisraC++-2.13.3

A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
check_only_hex_and_octal	Whether only hex and octal literals should be checked.	True
consider_conversion	Selects whether literals with an implicit conversion from signed to unsigned are reported.	False
consider_pp_expressions	Whether literals in #if conditions are checked.	False
report_all_unsuffixed_hex_and_octal	Whether hex and octal literals should always be reported when they are unsuffixed, independent of the environment	False

Possible Messages

Name	Message
missing_hex_suffix	Hexadecimal and octal constants require "U" suffix
missing_u_suffix	Constant of unsigned type requires "U" suffix

MisraC++-2.13.4

Literal suffixes shall be upper case.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
lowercase_suffix	Literal suffix should be upper case

MisraC++-2.13.5

Narrow and wide string literals shall not be concatenated.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
cpp11_mode	Use rules as defined in the C++11 standard.	False

Possible Messages

Name	Message
mixed_string_concatenation	Concatenation of mixed string encodings
narrow_wide_concat	Concatenation of narrow and wide string literal

MisraC++-3.1.1

It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
accept_const_fields	Whether const-qualified static fields in header files should be tolerated.	True
accept_const_variables	Whether global const variables in header files should be tolerated.	True
find_non_external_function_declaration	Whether function declarations with non-external linkage should be reported.	False

Possible Messages

Name	Message
function_definition_in_header	Definition in header file.
static_field_def_in_header	Definition in header file.
variable_definition_in_header	Definition in header file.

MisraC++-3.1.2

Functions shall not be declared at block scope.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
local_function_declaration	Functions shall not be declared at block scope.

MisraC++-3.1.3

When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
only_extern	Whether to consider only extern declared arrays	False
report_definitions	Whether definitions of array variables should also be reported	False

Possible Messages

Name	Message
unbounded_array	Array declaration with unknown size.

MisraC++-3.2.1

All declarations of an object or function shall have compatible types.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_parameter_type_qualifier_differences	If enabled, do not report a violation for different top-level type qualifiers in parameters: int f(int p); int f(const int p) {} Top-level qualifiers are only relevant for the implementation of the function and have no effect on callers, so it is not necessary to replicate them in the header file.	False
check_parameter_types	Whether parameter types should be compared	False
check_undefined	Whether only-declared routines and variables should also be checked.	True
require_exact_match	Whether to check for identical or compatible types (for routine return types).	False

Possible Messages

Name	Message
decl_return_type_mismatch	Return type at declarations differs
decl_type_mismatch	Type at declarations differs
parameter_type_mismatch	Parameter type at definition differs from type at declaration
return_type_mismatch	Return type at definition differs from type at declaration
type_mismatch	Type at definition differs from type at declaration

MisraC++-3.2.2

The One Definition Rule shall not be violated.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
class_struct_difference	{}
different_enumerators	{}
different_field_types	{}
different_fields	{}
general_odrViolation	{}

MisraC++-3.2.3

A type, object or function that is used in multiple translation units shall be declared in one and only one file.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_only_one_decl	Enables message if there are multiple declarations but all in same file.	False

Possible Messages

Name	Message
function_declared_in_multiple_files	Function declared in more than one file.
function_declared_multiple_times	Function declared more than once.
type_declared_in_multiple_files	Type declared in more than one file.
type_declared_multiple_times	Type declared more than once.
variable_declared_in_multiple_files	Variable declared in more than one file.
variable_declared_multiple_times	Variable declared more than once.

MisraC++-3.2.4

An identifier with external linkage shall have exactly one definition.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_undefined_specialized_template	Select whether undefined templates should be reported if specializations of them exist.	True
allow_unselected_declarations	Select whether to report declarations with no definition if they are unused.	True
allowed_undefined_functions	Regular expressions for functions which are tolerated without a definition, e.g. compiler intrinsics.	[_.*']
allowed_undefined_types	Regular expressions for types which are tolerated without a definition.	[_.*']
allowed_undefined_variables	Regular expressions for variables which are tolerated without a definition.	[_.*']
check_composite_types	Check class/struct/union types for having no definition even if they have no external linkage.	False
check_types_only_with_external_linkage	Whether the types should only be checked when they have external linkage	True

Possible Messages

Name	Message
composite_defined_multiple_times	Composite type has multiple definitions.
enum_defined_multiple_times	Enumeration type has multiple definitions.
function_defined_multiple_times	Type, object or function has multiple definitions.
undefined_function	Type, object or function without definition
undefined_type	Type without definition
undefined_variable	Type, object or function without definition
variable_defined_multiple_times	Type, object or function has multiple definitions.

MisraC++-3.3.1

Objects or functions with external linkage shall be declared in a header file.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
missing_function_declaration_in_header	Object or function with external linkage shall be declared in a header file
missing_variable_declaration_in_header	Object or function with external linkage shall be declared in a header file

MisraC++-3.3.2

If a function has internal linkage then all re-declarations shall include the static storage class specifier.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
function_decl_missing_static	Declaration should use static storage class specifier
function_def_missing_static	Definition should use static storage class specifier
variable_decl_missing_static	Declaration should use static storage class specifier
variable_def_missing_static	Definition should use static storage class specifier

MisraC++-3.4.1

An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_moving_to_other_primary_file	When a variable is only used in a single file X, but currently implemented in a different file Y, this controls whether the check suggests to move it into X and make it static there.	False
check_classes	Whether to report structs/classes/unions which are only used in a single function or file	True
check_loop_counter	Whether to report for-loop counters being declared before the loop that could be declared in the for-init part instead from C99 on	False
exclude_function_locals	If True, variables that could even be function-local are not reported.	True
exclude_undefined	Whether only-declared symbols should be reported as well	True
move_global_const_into_function	Controls suggestions for global constants that could be declared locally in a function.	True
only_check_unit_locals	Whether only global static variables should be checked.	False

Possible Messages

Name	Message
locality_block	{} can be declared in a more local scope.
locality_file	{} can be declared locally in primary file.
locality_function	Global {} can be declared inside function.
locality_loop_init	{} can be declared in the for-loop's initialization.
var_file_static	{} can be declared static in primary file.

MisraC++-3.9.1

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_missing_qualifiers	If True, tolerate differences in the use of explicit namespace/class qualifiers.	True

Possible Messages

Name	Message
parameter_type_tokens_mismatch	Type of redeclaration is not token-for-token identical
return_type_tokens_mismatch	Type of redeclaration is not token-for-token identical
variable_type_tokens_mismatch	Type of redeclaration is not token-for-token identical

MisraC++-3.9.2

Typedefs that indicate size and signedness should be used in place of the basic numerical types.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_floating_point	Whether floating point types without should not be reported.	False
ignore_inherited	If true, missing typedefs in inherited methods are not reported.	False
treat_plain_char_as_numerical	Whether the char type without any signedness should be reported.	False

Possible Messages

Name	Message
missing_integer_typedef	Use of base type outside typedef.
wrong_integer_typedef	Use of badly named typedef for base type.

MisraC++-3.9.3

The underlying bit representations of floating-point values shall not be used.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
float_bit_representation	Use of bit representation of a float value.

MisraC++-4.5.1

Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_sizeof	Allows the use of sizeof(bool_var). The Misra rule forbids all uses of expressions of type bool except for those explicitly allowed; and does not allow sizeof. However this is likely an oversight; there is no good reason to prevent the use of sizeof(bool_var). This option also allows alignof(bool_var) in C++11.	False
allow_unevaluated_operator	Allows the use of boolean expressions in contexts where the expression is not evaluated. As of C++17, these are: typeid(bool_var), noexcept(bool_var) and decltype(bool_var). sizeof(bool_var) is controlled by the separate option 'allow_sizeof'.	True

Possible Messages

Name	Message
bool_operand_in_bad_operator	Use of boolean operand in '{}' operator
bool_operand_outside_logical_and_relational_op	Use of boolean operand with integral promotion

MisraC++-4.5.2

Expressions with type enum shall not be used as operands to built-in operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
report_use_in_operator_calls	Whether enum arguments in calls to forbidden overloaded operators should be reported.	False

Possible Messages

Name	Message
enum_operand_outside_comparison	Use of enum operand in arithmetic or similar context

MisraC++-4.5.3

Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
char_operand_outside_comparison	Use of character operand in forbidden context

MisraC++-4.10.1

NULL shall not be used as an integer value.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
null_as_int	Use of NULL as integer value

MisraC++-4.10.2

Literal zero (0) shall not be used as the null-pointer-constant.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
require_nullptr	Require C++11 nullptr instead of macro NULL or literal zero (0).	False

Possible Messages

Name	Message
zero_as_null	Use of literal zero (0) as null-pointer-constant, use {} instead

MisraC++-5.0.1

The value of an expression shall be the same under any order of evaluation that the standard permits.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
ignore_exceptions	Ignore possible exceptions in the computation of side effects. (this option only has an effect if report_calls is enabled)	False
report_calls	If True, unsequenced function calls are reported.	False

Possible Messages

Name	Message
unsequenced_call_volatile	Function calls unsequenced with volatile access
unsequenced_calls	Unsequenced function calls
unsequenced_read_write	Unsequenced read and write accesses
unsequenced_volatile	Unsequenced volatile accesses
unsequenced_writes	Unsequenced writes

MisraC++-5.0.2

Limited dependence should be placed on C++ operator precedence rules in expressions.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_algebraic_order	Whether parens can be omitted for multiplicative operators as operands in additive operators	False
allow_redundant_parentheses_in_logical_operator	Allow redundant parentheses around unary operators when they appear as operand of '&&' or ' '. This option can be used to avoid a conflict with MisraC++-5.2.1, which requires parentheses in these instances.	True
report_unnecessary_parentheses	Controls whether unnecessary use of parentheses on the right side of assignments or around unary operators are reported.	True

Possible Messages

Name	Message
assignment_rhs_in_parens	No parentheses required for right-hand side of assignment
missing_parens_depends_on_precedence	Parentheses required to avoid dependence on precedence rules
unary_op_in_parens	No parentheses required for unary operator

MisraC++-5.0.3

A cvalue expression shall not be implicitly converted to a different underlying type.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes], [SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
type_category	Selection of type categories to consider for type changes inside same category.	[SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Result of cvalue expression implicitly converted to different underlying type
cast_changes_type_inside_category	Result of cvalue expression implicitly converted to different underlying type

MisraC++-5.0.4

An implicit integral conversion shall not change the signedness of the underlying type.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_constant_conversions	If this option is enabled, the rule is relaxed to allow implicit conversions of constant integer expressions whenever the constant value fits into the target type.	True
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
report_signed_to_unsigned	Whether conversion from signed to unsigned shall be reported.	True
report_unsigned_to_signed	Whether conversion from unsigned to signed shall be reported.	True
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_from_signed_to_unsigned	Implicit integral conversion changes signedness of underlying type
cast_from_unsigned_to_signed	Implicit integral conversion changes signedness of underlying type

MisraC++-5.0.5

There shall be no implicit floating-integral conversions.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes], [FloatingTypes]], [[FloatingTypes], [SignedTypes, UnsignedTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Implicit floating-integral conversion

MisraC++-5.0.6

An implicit integral or floating-point conversion shall not reduce the size of the underlying type.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	False
check_implicit_casts	Whether implicit casts should be checked and reported.	True
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
report_narrowing	Whether narrowing casts should be reported.	True
report_widening	Whether widening casts should be reported.	False
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
size_source_category	Selection of type categories of the operand being cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
size_target_category	Selection of type categories appearing as target types of the cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
narrowing_cast	Implicit conversion reduces size of underlying type
widening_cast	Conversion to larger type

MisraC++-5.0.7

There shall be no explicit floating-integral conversions of a cvalue expression.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, CharacterTypes], [FloatingTypes]], [[FloatingTypes], [SignedTypes, UnsignedTypes, CharacterTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Explicit floating-integral conversion of cvalue expression

MisraC++-5.0.8

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
report_narrowing	Whether narrowing casts should be reported.	False
report_widening	Whether widening casts should be reported.	True
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
size_source_category	Selection of type categories of the operand being cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
size_target_category	Selection of type categories appearing as target types of the cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
narrowing_cast	Conversion to smaller type
widening_cast	Explicit conversion increases size of underlying type of cvalue expression

MisraC++-5.0.9

An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
report_signed_to_unsigned	Whether conversion from signed to unsigned shall be reported.	True
report_unsigned_to_signed	Whether conversion from unsigned to signed shall be reported.	True
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_from_signed_to_unsigned	Explicit conversion changes signedness of underlying type of cvalue expression
cast_from_unsigned_to_signed	Explicit conversion changes signedness of underlying type of cvalue expression

MisraC++-5.0.10

If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
bitop_small_without_cast	Bitwise operator requires cast to underlying type on result

MisraC++-5.0.11

The plain char type shall only be used for the storage and use of character values.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
char_used_numerically	Plain char datatype used for non-character data.

MisraC++-5.0.12

signed char and unsigned char type shall only be used for the storage and use of numeric values.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
numeric_char_used_as_character	Signed/unsigned char datatype used for character data.

MisraC++-5.0.13

The condition of an if-statement and the condition of an iteration-statement shall have type bool.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
only_in_conditions	If True, only logical operators inside conditions are checked.	True
report_do_while_false	Whether do ... while(0) should be reported.	True
report_while_true	Whether while(1) ... should be reported.	True

Possible Messages

Name	Message
nonbool_if_condition	Condition must have type bool
nonbool_logical_operator_operand	Sub-condition must have type bool
nonbool_loop_condition	Condition must have type bool

MisraC++-5.0.14

The first operand of a conditional-operator shall have type bool.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
nonbool_conditional_operator_condition	Condition must have type bool

MisraC++-5.0.15

Array indexing shall be the only form of pointer arithmetic.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
indexing_only_on_identifiers	Report array indexing on pointers only for variables (`ptr[i]`), not for other pointer expressions (e.g. `get_ptr()[i]`). This option is meant to suppress the violations introduced by the BAUHAUS-12021 bugfix in version 6.9.6.	False

Possible Messages

Name	Message
array_indexing_on_pointer	Array indexing only allowed for arrays
pointer_arithmetic	Pointer arithmetic not allowed
pointer_increment_decrement	Pointer arithmetic not allowed

MisraC++-5.0.16

A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
arithmetic_on_nonarray	Pointer arithmetic on {1} operates on non-array target {0}
arithmetic_out_of_bounds	Pointer arithmetic on {1} creates pointer outside array bounds of {0}
out_of_bounds	Access into array is out of bounds
possible_indirect_out_of_bounds	Pointer-indirect access through {1} might be out of bounds accessing {0}
possible_out_of_bounds	Access into array might be out of bounds

MisraC++-5.0.17

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
possible_unrelated_ptr_subtraction	Subtracting possibly unrelated pointers
unrelated_ptr_subtraction	Subtracting unrelated pointers

MisraC++-5.0.18

>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
possible_unrelated_ptr_comparison	Comparing possibly unrelated pointers
unrelated_ptr_comparison	Comparing unrelated pointers

MisraC++-5.0.19

The declaration of objects should contain no more than two levels of pointer indirection.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
max_levels	Maximum number of allowed pointer-indirection levels.	2

Possible Messages

Name	Message
too_many_pointer_levels	More than {} level{} of pointer indirection

MisraC++-5.0.20

Non-constant operands to a binary bitwise operator shall have the same underlying type.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
bitop_with_different_operand_types	Non-constant operands of bitwise operator differ in underlying type
shortcut_bitop_with_different_operand_types	Non-constant operands of bitwise operator differ in underlying type

MisraC++-5.0.21

Bitwise operators shall only be applied to operands of unsigned underlying type.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
bitop_on_enum	Bitwise operator on enum underlying type
bitop_on_signed	Bitwise operator on signed underlying type

MisraC++-5.2.1

Each operand of a logical && or || shall be a postfix-expression.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allowed	Can be used to name additional allowed IR node types for the operands.	None
require_postfix_expression	Whether postfix or primary expressions are required as operands.	True

Possible Messages

Name	Message
missing_parens_for_logical_operator_operand	Operand of && or must be parenthesized.

MisraC++-5.2.2

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
cannot_cast_virtual_base	Cannot convert pointer to base class {} to pointer to derived class {} -- base class is virtual
missing_dynamic_cast_on_virtual_base	Use dynamic_cast on virtual base class

MisraC++-5.2.3

Casts from a base class to a derived class should not be performed on polymorphic types.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
cast_from_polybase_to_derived	Cast from polymorphic base class to derived class

MisraC++-5.2.4

C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value
allow_void_cast	If True, (void) is always allowed, else only for return value of calls.	False
allow_void_cast_on_call	If True, (void) is allowed, for return value of calls.	True

Possible Messages

Name	Message
c_cast	C-style cast
function_notation_cast	Functional notation cast

MisraC++-5.2.5

A cast shall not remove any const or volatile qualification from the type of a pointer or reference.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
check_const	Whether casts affecting const-qualifiers should be reported	True
check_volatile	Whether casts affecting volatile-qualifiers should be reported	True
only_top_level	Whether the check only applies to the top-level pointee or recursively to deeper pointer levels as well	False

Possible Messages

Name	Message
cast_removes_const	Cast removes const qualification
cast_removes_volatile	Cast removes volatile qualification
implicit_cast_removes_const	Implicit cast removes const qualification
implicit_cast_removes_volatile	Implicit cast removes volatile qualification

MisraC++-5.2.6

A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[FunctionPointerTypes], [VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes, NullPointerTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
type_category	Selection of type categories to consider for type changes inside same category.	[FunctionPointerTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion of function pointer to other type
cast_changes_type_inside_category	Conversion of function pointer to other function pointer type

MisraC++-5.2.7

An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[ObjectPointerTypes], [FunctionPointerTypes, IncompletePointerTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
type_category	Selection of type categories to consider for type changes inside same category.	[ObjectPointerTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion to unrelated pointer type
cast_changes_type_inside_category	Conversion to unrelated pointer type

MisraC++-5.2.8

An object with integer type or pointer to void type shall not be converted to an object with pointer type.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, VoidPointerTypes], [ObjectPointerTypes, IncompletePointerTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
look_through_casts	If True, [T*](void*)x will not be reported.	True
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
type_system	Which type system to use: compiler types, underlying types, essential types	UnderlyingTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion from void* or integer to pointer type

MisraC++-5.2.9

A cast should not convert a pointer type to an integral type.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories. [[[VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes], [SignedTypes, UnsignedTypes, BoolTypes, EnumTypes, CharacterTypes]]]	
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion between pointer and integral type

MisraC++-5.2.10

The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
forbid_all_operators	If True, forbids mixing with any kind of operator; else only with arithmetic operators.	False

Possible Messages

Name	Message
increment_mixed_with_operator	Increment or decrement mixed with other operators

MisraC++-5.2.11

The comma operator, && operator and the || operator shall not be overloaded.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
invalid	Selection of disallowed operator overloads by name.	['operator&&', 'operator ', 'operator,']
invalid_kinds	Selection of disallowed operator overloads by operator kind.	[]

Possible Messages

Name	Message
forbidden_operator_overload	Overloaded version of comma operator or && or

MisraC++-5.2.12

An identifier with array type passed as a function argument shall not decay to a pointer.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
decay	Array to pointer decay

MisraC++-5.3.1

Each operand of the ! operator, the logical && or the logical || operators shall have type bool.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
only_in_conditions	If True, only logical operators inside conditions are checked.	False

Possible Messages

Name	Message
nonbool_logical_operator_operand	Operand of logical operator shall be of type bool

MisraC++-5.3.2

The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
check_literals	If True, integer literals are also disallowed as operands.	True

Possible Messages

Name	Message
unary_minus_on_unsigned	Unary minus applied to unsigned

MisraC++-5.3.3

The unary & operator shall not be overloaded.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
invalid	Selection of disallowed operator overloads by name.	[]
invalid_kinds	Selection of disallowed operator overloads by operator kind.	[7]

Possible Messages

Name	Message
forbidden_operator_overload	Overloaded version of unary &

MisraC++-5.3.4

Evaluation of the operand to the sizeof operator shall not contain side effects.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
side_effect_in_sizeof	Operand of "sizeof" shall not contain side effects

MisraC++-5.8.1

The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
report_nonconst_cases	Whether a message should be printed for non-constant cases missing a range-checking if statement	False
use_essential_type	Whether essential type should be computed for allowed value range.	False

Possible Messages

Name	Message
excessive_shift	Right operand of shift operator has value {}, but should be in range 0 to {}
possible_excessive_shift	Right operand of shift operator should be checked to be in range 0 to {}

MisraC++-5.14.1

The right-hand operand of a logical && or || operator shall not contain side effects.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_volatile_access	Whether access to a volatile variable should be reported as side effect.	False
assume_effect_when_undefined	Whether all calls to undefined (external) functions should be seen as having a side-effect.	True

Possible Messages

Name	Message
call_with_side_effect	Call in right-hand operand of && or may have side-effect
modifies_local_var	Right-hand operand of && or modifies '{}'
side_effect	Right-hand operand of && or has side-effect

MisraC++-5.17.1

The semantic equivalence between a binary operator and its assignment operator form shall be preserved.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
check_call_relation	If True, checks whether there is a call relation between binary and assignment version.	True
ignore_stream_operators	If True, allows definitions of operator<<() and operator>>() without the corresponding assignment operator. Note this will also allow operator<<=() and operator>>=() as they no longer have an equivalent binary form.	False

Possible Messages

Name	Message
missing_assignment_version	Missing overload for corresponding assignment version of operator
missing_binary_version	Missing overload for corresponding binary version of operator
missing_call_to_assignment_version	There is no call relation between this operator and its assignment version to ensure semantic equivalence
missing_call_to_binary_version	There is no call relation between this operator and its binary version to ensure semantic equivalence

MisraC++-5.18.1

The comma operator shall not be used.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
use_of_comma	Use of comma sequence operator.

MisraC++-5.19.1

Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
const_expression_overflow	Overflow during evaluation of constant expression.

MisraC++-6.2.1

Assignment operators shall not be used in sub-expressions.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
assignment_result_used	Assignment inside sub-expression.

MisraC++-6.2.2

Floating-point expressions shall not be directly or indirectly tested for equality or inequality.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
float_equality_comparison	Testing a float for (in)equality.
float_indirect_equality_comparison	Indirectly testing a float for (in)equality.

MisraC++-6.2.3

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_empty_macros	Whether a macro invocation before the ; is allowed if it expands to nothing.	False
allow_nonempty_macros	Whether a non-empty macro invocation before the ; is allowed.	False

Possible Messages

Name	Message
null_statement_not_isolated	Null statement not on a line by itself

MisraC++-6.3.1

The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
loop_body_not_compound	Loop body shall be a statement sequence.
switch_body_not_compound	switch body shall be a statement sequence.

MisraC++-6.4.1

An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
dangerous_else	Block or if statement required as else branch.
dangerous_then	Block required as then branch.

MisraC++-6.4.2

All if ... else if constructs shall be terminated with an else clause.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
defensive_else	Controls if the else branch of if-cascade's should contain a suitable comment or not be empty.	True

Possible Messages

Name	Message
non_defensive_else	Last else in the if-cascade must not be empty or contain a comment.
unterminated_if_cascade	Last if in the if-cascade has no else.

MisraC++-6.4.3

A switch statement shall be a well-formed switch statement.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
acceptable_end_items	This set characterizes allowed last items at the end of a switch case	('Exit_Switch', 'Throw_Expression')
acceptable_middle_items	This set characterizes allowed last items in the middle of a switch case	('Exit_Switch', 'Nondefault_Case_Label', 'Throw_Expression')
allow_as_first	Whether to allow the "default" case as the first case in a switch.	False
allow_complete_enum	Whether a switch having a case for all enumerators is tolerated without a default.	True
allow_empty_statements	Whether empty statements at the end of switch-case should be ignored.	False
allow_fallthrough_comment	Whether a comment with text "fall" and "through" is allowed to mark fallthrough.	True
allow_fallthrough_macro	Whether a macro called __fallthrough is allowed to mark fallthrough.	True
allow_if	Whether an if containing acceptable items at the end of both branches is accepted.	False
allow_noreturn_function_calls	Whether a function call to a [[noreturn]] function is considered a valid termination.	False
comment	The comment that indicates an intentional fallthrough. Can be a list of comments.	[]
disallow_multiple_compound_statements	Whether there can be more than one toplevel compound statement.	True
disallowed_nested_statements	Statement types not allowed inside compound statements within case-clause body.	None
disallowed_toplevel_statements	Statement types not allowed in case-clause body on toplevel.	('Named_Label', 'Unconditional_Branch', 'Declaration_Statement')
minimum_switch_cases	The number of cases a switch statement should at least have.	1

Possible Messages

Name	Message
case_label_nested	A switch label shall only be used in a switch.
default_not_last	"default" clause is not last in switch.
empty_default	Default clause should contain a statement or comment.
empty_switch	Switch has no "case" clause.
forbidden_compound_statement_in_case	At most one compound statement is allowed on toplevel in case-clauses.
forbidden_nested_statement_in_case	Statement is not allowed inside compound statement in case-clauses.
forbidden_toplevel_statement_in_case	Statement is not allowed outside a compound statement in case-clauses.
minimum_switch_cases	Switch has to little "case" clauses.
missing_default	Switch has no "default" clause.
switch_body_not_compound	A switch statement shall be a well-formed switch statement.
switch_not_well_formed	Switch has a statement before the first case label.
switch_over_bool	Switch expression is effectively Boolean.
unterminated_case	This switch-case is not terminated with {}, last statement is {}.
unterminated_empty_case	This switch-case is empty, not terminated with {}.

MisraC++-6.4.4

A switch label shall only be used when the most closely-enclosing compound-statement is the body of a switch-statement.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
case_label_nested	A switch label shall only be used in a switch.

MisraC++-6.4.5

An unconditional throw or break statement shall terminate every non-empty switch-clause.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
acceptable_end_items	This set characterizes allowed last items at the end of a switch case	('Exit_Switch', 'Throw_Expression')
acceptable_middle_items	This set characterizes allowed last items in the middle of a switch case	('Exit_Switch', 'Nondefault_Case_Label', 'Throw_Expression')
allow_empty_statements	Whether empty statements at the end of switch-case should be ignored.	False
allow_fallthrough_comment	Whether a comment with text "fall" and "through" is allowed to mark fallthrough.	True
allow_fallthrough_macro	Whether a macro called __fallthrough is allowed to mark fallthrough.	True
allow_if	Whether an if containing acceptable items at the end of both branches is accepted.	False
allow_noreturn_function_calls	Whether a function call to a [[noreturn]] function is considered a valid termination.	False
comment	The comment that indicates an intentional fallthrough. Can be a list of comments.	[]

Possible Messages

Name	Message
unterminated_case	This switch-case is not terminated with {}, last statement is {}.
unterminated_empty_case	This switch-case is empty, not terminated with {}.

MisraC++-6.4.6

The final clause of a switch statement shall be the default clause.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_as_first	Whether to allow the "default" case as the first case in a switch.	False
allow_complete_enum	Whether a switch having a case for all enumerators is tolerated without a default.	True

Possible Messages

Name	Message
default_not_last	"default" clause is not last in switch.
empty_default	Default clause should contain a statement or comment.
missing_default	Switch has no "default" clause.

MisraC++-6.4.7

The condition of a switch statement shall not have bool type.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
switch_over_bool	Switch condition shall not have bool type.

MisraC++-6.4.8

Every switch statement shall have at least one case-clause.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
minimum_switch_cases	The number of cases a switch statement should at least have.	1

Possible Messages

Name	Message
empty_switch	Switch has no "case" clause.
minimum_switch_cases	Switch has too little "case" clauses.

MisraC++-6.5.1

A for loop shall contain a single loop-counter which shall not have floating type.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
loop_counter_model		MisraCppLoopCounters

Possible Messages

Name	Message
loop_float_counter	Loop-counter of for loop shall not have floating type
loop_missing_counter	For loop has no loop-counter
loop_multiple_counters	For loop shall have only a single loop-counter

MisraC++-6.5.2

If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
loop_counter_model		MisraCppLoopCounters

Possible Messages

Name	Message
stepping_loop_uses_equality_check	Loop-counter shall not be tested with equality operator if not modified by -- or ++

MisraC++-6.5.3

The loop-counter shall not be modified within condition or statement.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
loop_counter_model		MisraCppLoopCounters

Possible Messages

Name	Message
loop_counter_modified_in_condition	Loop-counter shall not be modified within condition
modified_loop_counter	Loop-counter shall not be modified within loop body

MisraC++-6.5.4

The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_constexpr	Allow constexpr as a constant <n>.	True
loop_counter_model		MisraCppLoopCounters

Possible Messages

Name	Message
nonconst_loop_increment	Loop-counter shall be modified by one of: --, ++, -=n, or +=n (with constant n)

MisraC++-6.5.5

A loop-control-variable other than the loop-counter shall not be modified within condition or expression.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
loop_counter_model		MisraCppLoopCounters

Possible Messages

Name	Message
modified_loop_control_variable	Loop-control variable (other than counter) shall not be modified within condition or expression

MisraC++-6.5.6

A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
loop_counter_model		MisraCppLoopCounters

Possible Messages

Name	Message
nonbool_loop_control_variable	Loop-control variable (other than counter) shall have type bool

MisraC++-6.6.1

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

Input: IR

Source languages: C, C++, C#

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
goto_label_block	Label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

MisraC++-6.6.2

The goto statement shall jump to a label declared later in the same function body.

Input: IR

Source languages: C, C++, C#

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
backwards_goto	Label referenced by a goto statement shall be declared later in same function.

MisraC++-6.6.3

The continue statement shall only be used within a well-formed for loop.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
loop_counter_model		MisraCppLoopCounters
report_outside_for_loops	Whether to report continue statements in do..while/while loops	True

Possible Messages

Name	Message
continue_in_bad_loop	The continue statement shall only be used within a well-formed for loop

MisraC++-6.6.4

For any iteration statement there shall be at most one break or goto statement used for loop termination.

Input: IR

Source languages: C, C++, C#

Configuration

Name	Explanation	Value
inspect_gotos	Whether gotos exiting the loop should be counted.	True

Possible Messages

Name	Message
multiple_loop_breaks	More than one break or goto terminates loop.

MisraC++-6.6.5

A function shall have a single point of exit at the end of the function.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
multiple_returns	Function shall have a single point of exit at the end of the function.

MisraC++-7.1.1

A variable which is not modified shall be const qualified.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
check_parameters	If False, rule is not applied to parameters.	True
ignore_pointer_variables	Whether variables of pointer type should be ignored.	False
only_check_unit_locals	Whether only local variables and global static variables should be checked.	False
only_immutable_data	Whether only declarations with immutable data should be checked.	False
report_only_at_definition	Report violations for non-const parameters only at the function definition, not the function declaration.	False

Possible Messages

Name	Message
parameter_missing_const	A parameter which is not modified shall be const qualified.
variable_missing_const	A variable which is not modified shall be const qualified.

MisraC++-7.1.2

A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
check_all_variables	Whether non-parameter variables should be checked as well.	False
ignore_forwarding_references	Whether to allow C++11 forwarding references (universal references) even if all template instances would also work with a const lvalue reference.	True
ignore_move_constructors	Whether to allow non-const rvalue references on constructors.	True
report_overrides	Whether to report violations on overrides. For virtual methods, this rule will report a violation only if the parameter can be marked const in all related methods in the inheritance hierarchy. Thus, the same violation is reported for each method in the inheritance hierarchy. You can set this option to False to report such violations only for the base methods.	True

Possible Messages

Name	Message
parameter_can_point_to_const	{ } can be declared as pointer/reference to const.

MisraC++-7.2.1

An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
asgn_not_an_enumerator	Assignment of value which does not correspond to an enumerator
asgn_out_of_enum_range	Assignment of value outside the range of the enumerators
conversion_creating_bad_enum_value	Expression does not correspond to an enumerator in { }

MisraC++-7.3.1

The global namespace shall only contain main, namespace declarations and extern "C" declarations.

Input: IR

Source languages: C++, C

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
global_symbol	Symbol not allowed in global namespace.

MisraC++-7.3.2

The identifier main shall not be used for a function other than the global function main.

Input: IR

Source languages: C, C++, C

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
nonglobal_function_named_main	The identifier main shall not be used for a function other than the global function main

MisraC++-7.3.3

There shall be no unnamed namespaces in header files.

Input: IR

Source languages: C, C++, C

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
unnamed_namespace_in_header	Unnamed namespaces in header file

MisraC++-7.3.4

Using-directives shall not be used.

Input: IR

Source languages: C, C++, C

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
using_directive	Using-directives shall not be used

MisraC++-7.3.5

Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
report_other_usages	Whether to find other usages (such as function calls, or taking a fp-reference) between multiple declarations for an identifier.	False

Possible Messages

Name	Message
declarations_surrounding_usage	Declarations straddle a usage
declarations_surrounding_using	Declarations straddle a using-declaration

MisraC++-7.3.6

using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
using_declaration_in_header	Using-declaration in header file
using_namespace_in_header	Using-directive in header file

MisraC++-7.4.1

All usage of assembler shall be documented.

Input: IR

Source languages: Assembler, C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
use_of_assembler	Usage of assembler shall be documented

MisraC++-7.4.2

Assembler instructions shall only be introduced using the asm declaration.

Input: IR

Source languages: Assembler, C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
pragma_asm	Assembler instructions shall only be introduced using the asm declaration

MisraC++-7.4.3

Assembly language shall be encapsulated and isolated.

Input: IR

Source languages: Assembler, C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
asm_not_encapsulated	Assembly language shall be encapsulated and isolated.

MisraC++-7.5.1

A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
global_reference_to_local_var	Address of local variable escapes via global variable.
parameter_reference_to_local_var	Address of local variable escapes via parameter.
returning_reference_to_local_var	Returning reference/pointer to local variable.

MisraC++-7.5.2

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_longer_living_local	Whether assignment to a longer-living local variable should be accepted.	False

Possible Messages

Name	Message
possibly_leaking_reference_to_local_variable	Address of local variable is assigned to longer-living object.

MisraC++-7.5.3

A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
only_const_reference	Whether only to report returns of parameters with reference to const.	False

Possible Messages

Name	Message
returning_reference_to_refparam	Returning reference/pointer to reference parameter.

MisraC++-7.5.4

Functions shall not call themselves, either directly or indirectly.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
function_cycle	Cycle of recursive functions
recursive_function	Function is directly recursive

MisraC++-8.0.1

An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_in_for_loop_init	Whether a multi-declaration is allowed in the for-init statement	False
allow_uninitialized_simple_type	Whether a multi-declaration of simple type is allowed when there is no initialization	False

Possible Messages

Name	Message
multi_declaration	Multiple declarators in single declaration

MisraC++-8.3.1

Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
redefinition_uses_different_default_argument	Default argument differs from the one in redefined method

MisraC++-8.4.1

Functions shall not be defined using the ellipsis notation.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
ignore_declarations	Allow declarations of functions with variable number of arguments (e.g. for existing library functions); only report definitions.	True
ignore_inherited	Do not report functions inheriting a variable number of arguments.	False

Possible Messages

Name	Message
ellipsis_parameter	Function definitions shall not use ellipsis

MisraC++-8.4.2

The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
ignore_casing	Whether to ignore casing for the check.	False
include_redefinitions	Whether declarations of redefining methods should be considered together with the redefined ones.	True
inspect_only_defined_routines	If False, re-declarations of undefined functions are compared to earlier declarations as well.	False

Possible Messages

Name	Message
parameter_name_mismatch	Different name used for parameter

MisraC++-8.4.3

All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
lambda_return_missing_value	Return without value in non-void lambda expression
missing_return	Non-void function needs return with value at end.
missing_return_in_lambda	Non-void lambda expression needs return with value at end.
return_missing_value	Return without value in non-void function

MisraC++-8.4.4

A function identifier shall either be used to call the function or it shall be preceded by &.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allowed	Qualified names of functions of which it is allowed to take the address implicitly, e.g. C++ I/O manipulators	['std::endl', 'std::flush', 'std::boolalpha', 'std::noboolalpha', 'std::showbase', 'std::noshowbase', 'std::showpoint', 'std::noshowpoint', 'std::showpos', 'std::noshowpos', 'std::skipws', 'std::noskipws', 'std::uppercase', 'std::nouppercase', 'std::unitbuf', 'std::nounitbuf', 'std::internal', 'std::left', 'std::right', 'std::dec', 'std::hex', 'std::oct', 'std::fixed', 'std::scientific', 'std::hexfloat', 'std::defaultfloat', 'std::ws', 'std::ends', 'std::resetiosflag', 'std::setiosflag', 'std::setbase', 'std::setfill', 'std::setprecision', 'std::setw', 'std::get_money', 'std::put_money', 'std::get_time', 'std::put_time', 'std::quoted']

Possible Messages

Name	Message
implicit_routine_address	Taking address of function without &

MisraC++-8.5.1

All variables shall have a defined value before they are used.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
init_functions	Names of functions to be inspected as well when called directly from constructor.	('Init', 'init')
inspect_directly_called_methods	Inspect all methods directly called from constructor.	False
only_member_initializer_list	Only inspect member initializer list and not the constructor body/methods.	False
perform_global_analysis	Whether global analysis (using iranalysis) should be done, or just a local analysis.	True
report_missing_base_constructors	Enables detection of constructors which rely on implicit base constructor calls.	False
report_missing_field_constructors	Enables detection of constructors which rely on implicit field constructor calls.	False

Possible Messages

Name	Message
implicit_field_init	Field is only implicitly initialized in constructor.
missing_base_class_init	Base class is not explicitly initialized in constructor.
missing_field_init	Field is not initialized in constructor.
pass_as_pointer_to_const_param	Passing uninitialized variable by pointer as function parameter with pointer-to-const type
possible_uninit	Use of possibly uninitialized variable
uninit	Use of uninitialized variable

MisraC++-8.5.2

Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_any_zero_initialization	Whether {0} is allowed in nested aggregates as well.	False
allow_toplevel_zero_initialization	Whether {0} is allowed as top-level initialization for the complete aggregate	True
allowed_classes_without_braces	For the listed class types, don't require braces. This is useful for classes that transparently wrap a single contained value. For example, with this option "std::array<int, 3> a = {1,2,3};" is OK even though there could be two sets of braces (one for "std::array", the other and for its inner "int[3]" array).	['std::array']

Possible Messages

Name	Message
incomplete_aggregate_initialization	Provide explicit initializer for each part of initialized entity
missing_aggregate_initialization_braces	Use braces to indicate structure of initialized entity

MisraC++-8.5.3

In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
incomplete_enum_init	Do not initialize enumerators other than the first, or initialize all

MisraC++-9.3.1

const member functions shall not return non-const pointers or references to class-data.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
include_copies_of_pointer_fields	Whether returning a pointer field should be reported as well.	True
inspect_only_const_methods	Whether all methods or only const methods should be checked.	True
only_report_references	Whether pointer and reference to field should be reported, or just references.	False
smart_pointer_names	Names that are considered smart pointers.	['unique_ptr', 'shared_ptr', 'weak_ptr']

Possible Messages

Name	Message
returning_nonconst_member_reference	Returning non-const reference/pointer to class data.

MisraC++-9.3.2

Member functions shall not return non-const handles to class-data.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
include_copies_of_pointer_fields	Whether returning a pointer field should be reported as well.	True
inspect_only_const_methods	Whether all methods or only const methods should be checked.	False
only_report_references	Whether pointer and reference to field should be reported, or just references.	True
smart_pointer_names	Names that are considered smart pointers.	['unique_ptr', 'shared_ptr', 'weak_ptr']

Possible Messages

Name	Message
returning_nonconst_member_reference	Returning non-const reference to class data.

MisraC++-9.3.3

If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
report_candidates_for_const	If False, avoid reporting methods that can be made const.	True
report_candidates_for_static	If False, avoid reporting methods that can be made static.	True
test_operators_for_static	If True, check whether a method can be made static is also applied to operator methods	False

Possible Messages

Name	Message
method_can_be_const	Method can be declared const.
method_can_be_static	Method can be declared static.

MisraC++-9.5.1

Unions shall not be used.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_tagged_unions	Whether tagged unions (nested in a struct that has an enum discriminator) should be allowed	False

Possible Messages

Name	Message
union	Unions shall not be used

MisraC++-9.6.1

When the absolute positioning of bits representing a bit-field is required, then the behaviour and packing of bit-fields shall be documented.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
bitfield	Usage of bit-fields shall be documented

MisraC++-9.6.2

Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
accept_system_typedef	Whether standard typedefs like int32_t are accepted if from a system header even when they do not use an explicit sign for the underlying type	True

Possible Messages

Name	Message
missing_bitfield_sign	Bit-field type should be an explicitly unsigned or signed integral type.

MisraC++-9.6.3

Bit-fields shall not have enum type.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
enum_bitfield	Bit-fields shall not have enum type.

MisraC++-9.6.4

Named bit-fields with signed integer type shall have a length of more than one bit.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
signed_single_bitfield	Signed bit field shall be at least 2 bits long.

MisraC++-10.1.1

Classes should not be derived from virtual bases.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
virtual_inheritance	Classes should not be derived from virtual bases.

MisraC++-10.1.2

A base class shall only be declared virtual if it is used in a diamond hierarchy.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
virtual_inheritance_outside_diamond	A base class shall only be declared virtual if it is used in a diamond hierarchy.

MisraC++-10.1.3

An accessible base class shall not be both virtual and non-virtual in the same hierarchy.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
base_class_being_virtual_and_nonvirtual	Has base class which is both virtual and non-virtual.

MisraC++-10.2.1

All accessible entity names within a multiple inheritance hierarchy should be unique.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
ambiguous_member	All accessible entity names within a multiple inheritance hierarchy should be unique
use_of_ambiguous_name	{ } is ambiguous

MisraC++-10.3.1

There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
redefinition_of_defined_virtual_function	Redefinition of already defined virtual function.

MisraC++-10.3.2

Each overriding virtual function shall be declared with the virtual keyword.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_override_without_virtual	Don't require the 'virtual' modifier if the C++11 'override' modifier is in use.	True

Possible Messages

Name	Message
redefinition_missing_virtual	Redefinition requires 'virtual' keyword.
redefinition_missing_virtual_or_override	Redefinition requires 'virtual' or 'override' keyword.

MisraC++-10.3.3

A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
pure_redefinition	Pure redefinition of non-pure virtual function.

MisraC++-11.0.1

Member data in non-POD class types shall be private.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_protected_members	If True, protected members are tolerated.	False
allowed	Specifies allowed fields as pairs (class name pattern, field name pattern).Example: {re.compile('.*'), re.compile('x')} to allow x in all classes.	[]
ignore_const_members	If True, non-private const members are tolerated.	False
ignore_pod	Whether fields in POD classes should be reported.	True
ignore_structs	Whether fields in structs should be reported.	False
ignore_templates	Whether fields in generic templates should be reported.	True

Possible Messages

Name	Message
protected_field	Member data in non-POD class types shall be private.
public_field	Member data in non-POD class types shall be private.

MisraC++-12.1.1

An object's dynamic type shall not be used from the body of its constructor or destructor.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
constructor_using_dynamic_cast	Dynamic cast used in constructor/destructor.
constructor_using_typeid	Typeid on polymorphic class used in constructor/destructor.
constructor_using_virtual_call	Virtual call used in constructor/destructor.

MisraC++-12.1.2

All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
missing_base_class_init	Base class is not explicitly initialized in constructor.

MisraC++-12.1.3

All constructors that are callable with a single argument of fundamental type shall be declared explicit.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
consider_only_fundamental_types	Whether this check should be limited to single arguments of fundamental type or should also be applied to user defined types.	True

Possible Messages

Name	Message
constructor_missing_explicit	Constructor shall be declared explicit

MisraC++-12.8.1

A copy constructor shall only initialize its base classes and the non-static members of the class of which it is a member.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
report_copy_constructor	Whether to report side effects on copy constructors.	True
report_move_constructor	Whether to report side effects on move constructors.	False

Possible Messages

Name	Message
copy_ctor_with_side_effect	Copy Constructor has side-effect
move_ctor_with_side_effect	Move Constructor has side-effect

MisraC++-12.8.2

The copy assignment operator shall be declared protected or private in an abstract class.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
report_compiler_generated_copy_assign	Report (implicitly public) compiler-generated assignment operator in abstract classes.	True

Possible Messages

Name	Message
compiler_asgn	Compiler-generated copy assignment operator shall be explicitly declared protected or private in an abstract class
public_asgn	Copy assignment operator shall be declared protected or private in an abstract class

MisraC++-14.5.1

A non-member generic function shall only be declared in a namespace that is not an associated namespace.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
template_function_in_associated_namespace	Generic function in associated namespace {}

MisraC++-14.5.2

A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
missing_copy_constructor_for_template	Class has template constructor but no copy constructor

MisraC++-14.5.3

A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
missing_copy_asgn_for_template	Class has template assignment operator but no copy assignment operator

MisraC++-14.6.1

In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
unqualified_dependent_member_access	Use qualifiers or this-> to select name that may be found in that dependent base

MisraC++-14.6.2

The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
call_to_later_decl	Call to function declared later in translation unit

MisraC++-14.7.1

All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
only_check_in_primary_file	Whether only templates defined in primary file should be checked.	False

Possible Messages

Name	Message
uninstantiated_method_in_template	Method in template is never instantiated through a call
uninstantiated_static_field_in_template	Static field in template is never instantiated
uninstantiated_template	Template is not instantiated

MisraC++-14.7.2

For any given template specialization, an explicit instantiation of the template with the template-arguments used in the specialization shall not render the program ill-formed.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
explicit_instantiation_would_be_illegal	Explicit instantiation could render the program ill-formed

MisraC++-14.7.3

All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_in_type_declaration_file	Also allow specializations in the same file of a user-defined type for which the specialization is declared.	False
relax_if_template_only_declared	Allow specializations that are not declared in the header file if the template itself is only declared but not defined in the header.	False

Possible Messages

Name	Message
template_specialization_in_different_file	Specialization not declared in same file as primary template

MisraC++-14.8.1

Overloaded function templates shall not be explicitly specialized.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
specialization_of_overloaded_template	Overloaded function templates shall not be explicitly specialized

MisraC++-14.8.2

The viable function set for a function call should either contain no function specializations, or only contain function specializations.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
template_nontemplate_callee	Viable function set for this call contains both non-template {} and {}

MisraC++-15.0.1

Exceptions shall only be used for error handling.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
locally_catched	Exceptions shall only be used for error handling.

MisraC++-15.0.2

An exception object should not have pointer type.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
throwing_pointer	Exception object of pointer type

MisraC++-15.0.3

Control shall not be transferred into a try or catch block using a goto or a switch statement.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
goto_into_try	Goto jumps into try or catch block
switch_into_try	Switch statement jumps into try or catch block

MisraC++-15.1.1

The assignment-expression of a throw statement shall not itself cause an exception to be thrown.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
throw_expression_raises_exception	Expression of throw may itself raise an exception

MisraC++-15.1.2

NULL shall not be thrown explicitly.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
throwing_null	NULL shall not be thrown explicitly

MisraC++-15.1.3

An empty throw (throw;) shall only be used in the compound-statement of a catch handler.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
rethrow_outside_catch	Rethrow outside any catch block

MisraC++-15.3.1

Exceptions shall be raised only after start-up and before termination of the program.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
report_only_one_exception_per_function	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
report_only_uncaught	Whether the check shall report all throws or just those not caught.	False

Possible Messages

Name	Message
exception_escaping_initialization	Uncaught exception raised in initialization or finalization
exception_raised_in_initialization	Exception raised in initialization or finalization

MisraC++-15.3.2

There should be at least one exception handler to catch all otherwise unhandled exceptions.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
external_base_exceptions	Sequence of external/third-party exception base-classes that should be caught	[]
inspect_thread_main	Whether to also inspect thread main functions.	False
std_base_exceptions	Sequence of cpp-std exception base-classes that should be caught	[]

Possible Messages

Name	Message
missing_catch_all	Catch-all required around main program body
missing_catch_handler	Handler for {} needed

MisraC++-15.3.3

Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
handler_uses_field	Handler of a function-try-block shall not reference non-static members from this class or its bases

MisraC++-15.3.4

Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
inspect_at_exit_handlers	Whether to also inspect at_exit() handlers-functions.	True
inspect_thread_main	Whether to also inspect thread main functions.	True
report_only_one_exception_per_function	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False

Possible Messages

Name	Message
exception_escaping_initialization	Uncaught exception raised in initialization or finalization
exception_escaping_main	Uncaught exception escaping from main or additional entry point

MisraC++-15.3.5

A class type exception shall always be caught by reference.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
only_class_types	Whether all types should be caught by reference or only class types.	True

Possible Messages

Name	Message
catch_without_reference	A class type exception shall always be caught by reference.

MisraC++-15.3.6

Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
wrong_catch_order	Catch handlers in wrong order.

MisraC++-15.3.7

Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
catch_all_not_last	Catch-all shall occur as last handler.

MisraC++-15.4.1

If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
exception_specification_mismatch	Mismatch in exception specification types.

MisraC++-15.5.1

A class destructor shall not exit with an exception.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
allowed_exceptions	Exceptions that are allowed to escape from destructors.	['bad_alloc', 'bad_cast', 'failure', 'runtime_error', 'system_error']
constructors	Whether to consider constructors.	False
destructors	Whether to consider destructors.	True
generateViolationPath	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True

Possible Messages

Name	Message
exceptionEscapingConstructor	Escaping exception from constructor.
exceptionEscapingDestructor	Escaping exception from destructor.

MisraC++-15.5.2

Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
generateViolationPath	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignoreConstructorDestructor	Whether to ignore escaping exceptions from constructors and destructors.	False
ignoreUnknownRoutines	Whether to ignore extern or only declared routines.	False

Possible Messages

Name	Message
exceptionSpecificationViolation	Exception violates function's exception-specification.

MisraC++-15.5.3

The terminate() function shall not be called implicitly.

Input: IR

Source languages: C, C++

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
allowed_exceptions	Exceptions that are allowed to escape from destructors.	['bad_alloc', 'bad_cast', 'failure', 'runtime_error', 'system_error']
constructors	Whether to consider constructors.	False
destructors	Whether to consider destructors.	True
generate_violation_path	Whether to compute a trace for the exception. This improves the usability of the violation description, but requires additional computing which might slow down the rule.	True
ignore_constructor_destructor	Whether to ignore escaping exceptions from constructors and destructors.	True
ignore_throwing_functions	Whether to ignore noexcept specification violations on function that actually throw an exception.	False
ignore_unkown_routines	Whether to ignore extern or only declared routines.	False
inspect_at_exit_handlers	Whether to also inspect at_exit() handlers-functions.	True
inspect_thread_main	Whether to also inspect thread main functions.	True
report_noexcept_falseViolations	Whether to report cases where the function is declared noexcept(false), but no exceptions are ever thrown (directly or indirectly).	False
report_only_one_exception_per_function	Report at most one uncaught exception per function. This suppresses issues at sites where an uncaught exception is thrown to get a faster execution of the check. Setting this parameter to True will result in false negatives: Real issues may not be detected anymore.	False
required	Dict which lists required operations per resource. The mapping gives each case a description which maps to a dict for key "Required_Functions", "Resource_Parameter_Empty".	dict(...)
resources	Configuration of resources and operations on them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)

Possible Messages

Name	Message
exceptionEscapingConstructor	Escaping exception from constructor.
exceptionEscapingDestructor	Escaping exception from destructor.
exceptionEscapingInitialization	Uncaught exception raised in initialization or finalization
exceptionEscapingMain	Uncaught exception escaping from main or additional entry point
exceptionSpecificationViolation	Exception violates function's exception-specification.
implicitNoexceptSpecViolationWithout	Function implicitly declared noexcept(false) but no exceptions will be thrown.
noexceptSpecViolationWith	Exception violates function's noexcept-specification.
noexceptSpecViolationWithout	Function declared noexcept(false) but no exceptions will be thrown.
possiblyRequiredOperation	This thread is possibly joinable on destructor call
requiredOperation	This thread is joinable on destructor call

MisraC++-16.0.1

#include directives in a file shall only be preceded by other preprocessor directives or comments.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_extern_blocks	Whether to allow includes in `extern` blocks and not report a violation.	False

Possible Messages

Name	Message
include_after_code	#include should only be preceded by comments or preprocessor directives

MisraC++-16.0.2

Macros shall only be #define'd or #undef'd in the global namespace.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
local_macro	#define or #undef not in global namespace

MisraC++-16.0.3

#undef shall not be used.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
undef	#undef shall not be used

MisraC++-16.0.4

Function-like macros shall not be defined.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
function_macro_definition	Function-like macros shall not be defined

MisraC++-16.0.5

Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
directives	Preprocessing directives to check for in macro arguments.	('#if', '#ifdef', '#ifndef', '#elif', '#else', '#endif', '#pragma', '#warning', '#error', '#line', '#include', '#include_next', '#ident', '#region', '#endregion', '#asm', '#endasm', '#define', '#undef')

Possible Messages

Name	Message
pp_directive_as_macro_arg	Preprocessing directive used in macro argument.

MisraC++-16.0.6

In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
unsafe_macro_param	Macro parameter not enclosed in parentheses

MisraC++-16.0.7

Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
undefined_macro_used	Zero used for undefined preprocessing identifier

MisraC++-16.0.8

If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
malformed_pp_directive	Malformed or unknown preprocessing directive

MisraC++-16.1.1

The defined preprocessor operator shall only be used in one of the two standard forms.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
nonstandard_defined	Non-standard use of defined operator

MisraC++-16.1.2

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
missing_endif	The #endif for this directive is missing
missing_if	The #if for this directive is missing

MisraC++-16.2.1

The pre-processor shall only be used for file inclusion and include guards.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
accept_conditional_includes	Whether to accept #ifs for conditional includes.	False

Possible Messages

Name	Message
function_macro_definition	Macros are only allowed for include guards
line_directive	The pre-processor shall only be used for file inclusion and include guards.
object_macro_definition	Macros are only allowed for include guards
pp_if	Conditional compilation is only allowed for include guards
pragma	The pre-processor shall only be used for file inclusion and include guards.
undef	The pre-processor shall only be used for file inclusion and include guards.

MisraC++-16.2.2

C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
forbidden_cpp_macro_definition	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers
function_macro_definition	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers

MisraC++-16.2.3

Include guards shall be provided.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
macro_name_restrictions	Python iterable of functions with parameters (file, define, macro) to perform additional checks on the macro used as include guard in the given file. Should return None if macro is accepted, else a message to print for the violation. The parameters are: PIR File node for the header, token for the macro #define or #pragma once, and the name of the macro or the string 'pragma once'.	None

Possible Messages

Name	Message
include_guard_missing	Missing include guard.
include_guard_partial	Include guard does not cover complete file.
include_guard_with_wrong_define	Bad include guard: #define uses wrong macro, should be {}.
include_guard_without_define	Incomplete include guard: missing #define {}.

MisraC++-16.2.4

The ', ', /* or // characters shall not occur in a header file name.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
check_quotation_mark	Whether system includes should be checked for containing a ".	True
forbidden	The substrings to check for. " will be added for system-includes.	set(['//', '/*', '""'])

Possible Messages

Name	Message
nonstandard_include_character	Non-standard character in #include directive

MisraC++-16.2.5

The \ character should not occur in a header file name.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
check_quotation_mark	Whether system includes should be checked for containing a ".	False
forbidden	The substrings to check for. " will be added for system-includes.	set(['\\'])

Possible Messages

Name	Message
nonstandard_include_character	Non-standard character in #include directive

MisraC++-16.2.6

The #include directive shall be followed by either a <filename> or "filename" sequence.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
nonstandard_include_form	#include shall be either <filename> or "filename"

MisraC++-16.3.1

There shall be at most one occurrence of the # or ## operators in a single macro definition.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
report_only_concatenated_stringized	If True, only macros where a parameter is stringized and the result used for token concatenation are reported.	False

Possible Messages

Name	Message
concatenated_stringized	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator
more_than_one_macro_hash	Only one # or ## allowed in macro definition

MisraC++-16.3.2

The # and ## operators should not be used.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
hash_in_macro	The # and ## operators should not be used.

MisraC++-16.6.1

All uses of the #pragma directive shall be documented.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
pragma	Use of #pragma

MisraC++-17.0.1

Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
additional_reserved_identifiers	Names listed here are seen as violations as well.	set(['assert', 'errno', 'defined'])
allow_reserved_identifier_as_include_guard	If True, macros used for include guards are not checked for being a reserved identifier.	False
check_for_keyword	Whether macro names being keywords should be reported as well.	True

Possible Messages

Name	Message
macro_having_reserved_name	Definition of reserved identifier or standard library element
undef_of_reserved_name	#undef of reserved identifier or standard library element

MisraC++-17.0.2

The names of standard library macros and objects shall not be reused.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
check_locals	Whether parameters and local variables should also be checked	True

Possible Messages

Name	Message
reused_macro_object_libname	The names of standard library macros and objects shall not be reused.

MisraC++-17.0.3

The names of standard library functions shall not be overridden.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_function_declarations	Whether nondefining function declarations with library names are allowed	False

Possible Messages

Name	Message
reused_routine_libname	The names of standard library functions shall not be overridden.

MisraC++-17.0.5

The setjmp macro and the longjmp function shall not be used.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
symbol_header	Name of the header file of which the symbols should not be used.	setjmp
symbols	Names of symbols which are forbidden.	['setjmp', 'longjmp']
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

MisraC++-18.0.1

The C library shall not be used.

Input: IR

Source languages: C++

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
c_lib_header	Include <{}> instead of <{}>.
cpp_lib_header_with_suffix	Include <{}> instead of <{}>.

MisraC++-18.0.2

The library functions atof, atoi and atol from library <cstdlib> shall not be used.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
symbol_header	Name of the header file of which the symbols should not be used.	stdlib
symbols	Names of symbols which are forbidden.	['atof', 'atoi', 'atol']
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

MisraC++-18.0.3

The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
symbol_header	Name of the header file of which the symbols should not be used.	stdlib
symbols	Names of symbols which are forbidden.	['abort', 'exit', 'getenv', 'system']
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

MisraC++-18.0.4

The time handling functions of library <ctime> shall not be used.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
symbol_header	Name of the header file of which the symbols should not be used.	time
symbols	Names of symbols which are forbidden.	['clock', 'difftime', 'mktime', 'time', 'asctime', 'ctime', 'gmtime', 'localtime', 'strftime']
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

MisraC++-18.0.5

The unbounded functions of library <cstring> shall not be used.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
symbol_header	Name of the header file of which the symbols should not be used.	string
symbols	Names of symbols which are forbidden.	['strcpy', 'strcmp', 'strcat', 'strchr', 'strspn', 'strcspn', 'strpbrk', 'strrchr', 'strstr', 'strtok', 'strlen']
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h} -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

MisraC++-18.2.1

The macro offsetof shall not be used.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
symbol_header	Name of the header file of which the symbols should not be used.	stddef
symbols	Names of symbols which are forbidden.	['offsetof']
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h} -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

MisraC++-18.4.1

Dynamic heap memory allocation shall not be used.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
allow_base_classes_from	List of path globbing patterns to identify the location of base classes which (together with classes derived from them) should not be reported. For example, a globbing pattern like '/usr/include/*/qt*/Qt*' would allow classes derived from Qt classes if your Qt installation resides there.	[]

Possible Messages

Name	Message
cpp_new_delete	Builtin dynamic memory management operator used.
stdlib_memory_function_call	Dynamic memory management function from <stdlib.h> is called.

MisraC++-18.7.1

The signal handling facilities of <csignal> shall not be used.

Input: IR
Source languages: C, C++

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header (used for symbols like NULL and size_t which are defined in multiple headers)	['NULL', 'size_t']
header	Name of the header file which should not be used.	signal

Possible Messages

Name	Message
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.

MisraC++-19.3.1

The error indicator errno shall not be used.

Input: IR
Source languages: C, C++

Configuration

Name	Explanation	Value
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
symbol_header	Name of the header file of which the symbols should not be used.	['errno', 'stdlib', 'stddef']
symbols	Names of symbols which are forbidden.	['errno']
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <cerrno>.

MisraC++-27.0.1

The stream input/output library <cstdio> shall not be used.

Input: IR
Source languages: C, C++

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header (used for symbols like NULL and size_t which are defined in multiple headers)	['NULL', 'size_t']
header	Name of the header file which should not be used.	stdio

Possible Messages

Name	Message
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.

Rules in Group MisraC2012

The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
limits	Translation limits of your compiler, defaults to C99-guaranteed limits. You can change individual limits, e.g. from bauhaus.ir.common.languages import translation_limits followed by RULES['...'].limits[translation_limits.enumerators] = 400000 The dict-keys usable here can be found in the IR reference guide where the module bauhaus.ir.languages.translation_limits is described. You can also assign your own translation limit class when it follows the standard ones from the translation_limits module: RULES['...'].limits = MyCompilerLimits()	C99_Limits
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
		set[[2, 3, 6, 7, 8, 9, 10, 11, 12, 13, 14, 17, 18, 19, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 44, 45, 46, 47, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 65, 66, 67, 68, 69, 70, 71, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 112, 114, 115, 116, 117, 118, 119, 120, 121, 122, 124, 125, 126, 127, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 157, 158, 159, 160, 165, 166, 167, 168, 169, 170, 171, 172, 173, 175, 179, 180, 181, 182, 183, 184, 192, 194, 195, 196, 220, 221, 222, 224, 225, 226, 227, 228, 229, 230, 235, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 262, 263, 264, 265, 266, 268, 269, 274, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 296, 297, 298, 299, 300, 302, 304, 305, 306, 307, 308, 309, 310, 311, 312, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 356, 357, 359, 360, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 375, 377, 378, 380, 382, 384, 386, 387, 389, 390, 391, 392, 393, 394, 395, 397, 398, 399, 400, 401, 403, 404, 405, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 424, 427, 429, 430, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 445, 449, 450, 451, 452, 457, 458, 459, 460, 461, 462, 463, 464, 466, 467, 468, 469, 470, 471, 472, 473, 475, 476, 478, 479, 481, 484, 485, 486, 487, 489, 490, 493, 494, 496, 497, 498, 500, 501, 502, 503, 504, 505, 506, 507, 508, 510, 511, 512, 513, 515, 516, 517, 518, 519, 520, 521, 523, 525, 526, 529, 530, 531, 532, 533, 534, 535, 536, 541, 543, 544, 545, 546, 548, 549, 551, 552, 553, 555, 556, 558, 559, 560, 598, 599, 601, 603, 604, 605, 606, 607, 608, 609, 611, 612, 617, 618, 619, 620, 624, 643, 644, 646, 647, 651, 654, 655, 656, 657, 658, 660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 673, 674, 676, 681, 682, 688, 689, 691, 692, 693, 694, 695, 696, 697, 698, 701, 702, 704, 705, 706, 707, 709, 710, 711, 713, 714, 715, 716, 717, 718, 719, 720, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 737, 738, 742, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 765, 766, 767, 768, 769, 771, 772, 773, 774, 775, 776, 777, 779, 782, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795, 799, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 810, 811, 812, 816, 817, 818, 819, 821, 822, 824, 825, 826, 827, 829, 830, 832, 833, 834, 835, 836, 837, 838, 840, 841, 842, 844, 845, 846, 847, 848, 849, 850, 851, 852, 854, 855, 857, 858, 859, 861, 862, 864, 865, 867, 868, 870, 871, 872, 873, 875, 876, 877, 878, 879, 880, 881, 882, 884, 885, 886, 887, 888, 890, 891, 892, 893, 894, 895, 896, 898, 901, 904, 905, 906, 907, 908, 909, 910, 911, 912, 916, 925, 928, 929, 930, 934, 935, 936, 937, 938, 939, 940, 942, 946, 948, 949, 951, 952, 953, 954, 955, 956, 960, 961, 962, 963, 964, 965, 966, 967, 968, 969, 971, 972, 975, 976, 977, 978, 979, 980, 982, 984, 985, 987, 988, 989, 990, 991, 993, 994, 995, 996, 997, 998, 999, 1000, 1001, 1006, 1007, 1009, 1010, 1011, 1012, 1013, 1014, 1015, 1017, 1018, 1019, 1020, 1021, 1022, 1023, 1028, 1029, 1030, 1031, 1032, 1033, 1034, 1035, 1036, 1037, 1038, 1039, 1040, 1041, 1042, 1043, 1044, 1045, 1047, 1049, 1051, 1052, 1054, 1055, 1061, 1065, 1066, 1067, 1068, 1069, 1070, 1071, 1072, 1075, 1076, 1077, 1081, 1084, 1086, 1087, 1088, 1089, 1090, 1091, 1094, 1096, 1097, 1098, 1099, 1101, 1102, 1103, 1105, 1107, 1109, 1110, 1112, 1113, 1115, 1116, 1118, 1120, 1121, 1122, 1123, 1124, 1127, 1128, 1129, 1130, 1131, 1132, 1133, 1134, 1135, 1136, 1137, 1138, 1139, 1142, 1143, 1144, 1146, 1147, 1148, 1149, 1151, 1152, 1153, 1154, 1155, 1158, 1159, 1161, 1162, 1163, 1166, 1167, 1168]

		2461, 2462, 2463, 2464, 2465, 2466, 2467, 2468, 2469, 2470, 2471, 2472, 2473, 2474, 2475, 2476, 2477, 2478, 2479, 2480, 2481, 2482, 2483, 2484, 2485, 2486, 2487, 2488, 2489, 2490, 2491, 2492, 2493, 2494, 2495, 2496, 2497, 2498, 2499, 2500, 2501, 2502, 2503, 2504, 2506, 2507, 2508, 2509, 2510, 2511, 2512, 2513, 2514, 2515, 2516, 2517, 2518, 2519, 2520, 2521, 2522, 2523, 2524, 2525, 2526, 2527, 2529, 2530, 2531, 2532, 2533, 2534, 2535, 2536, 2537, 2538, 2539, 2540, 2541, 2542, 2543, 2544, 2545, 2546, 2547, 2548, 2550, 2552, 2553, 2554, 2555, 2556, 2557, 2558, 2559, 2560, 2561, 2562, 2563, 2564, 2565, 2566, 2567, 2568, 2569, 2570, 2571, 2572, 2573, 2574, 2575, 2576, 2577, 2578, 2579, 2580, 2581, 2582, 2583, 2584, 2585, 2586, 2589, 2591, 2592, 2593, 2594, 2595, 2596, 2597, 2598, 2599, 2600, 2601, 2602, 2603, 2604, 2605, 2606, 2607, 2608, 2609, 2610, 2611, 2612, 2613, 2614, 2615, 2616, 2617, 2618, 2619, 2620, 2621, 2622, 2623, 2624, 2625, 2626, 2627, 2628, 2629, 2630, 2631, 2632, 2633, 2634, 2635, 2636, 2637, 2638, 2639, 2640, 2641, 2642, 2643, 2644, 2645, 2647, 2648, 2649, 2650, 2651, 2652, 2654, 2655, 2656, 2657, 2662, 2663, 2664, 2665, 2666, 2668, 2670, 2671, 2672, 2673, 2674, 2675, 2676, 2677, 2678, 2679, 2680, 2681, 2682, 2683, 2684, 2686, 2687, 2690, 2691, 2692, 2693, 2694, 2695, 2697, 2698, 2699, 2700, 2701, 2702, 2703, 2704, 2705, 2706, 2707, 2708, 2709, 2710, 2711, 2712, 2713, 2714, 2715, 2716, 2717, 2718, 2719, 2720, 2721, 2722, 2723, 2724, 2725, 2726, 2727, 2728, 2729, 2730, 2731, 2733, 2734, 2735, 2736, 2737, 2738, 2739, 2740, 2741, 2742, 2743, 2744, 2745, 2746, 2747, 2748, 2749, 2750, 2751, 2752, 2754, 2755, 2756, 2757, 2758, 2759, 2760, 2761, 2762, 2763, 2764, 2766, 2767, 2768, 2769, 2770, 2771, 2773, 2774, 2775, 2776, 2777, 2780, 2781, 2782, 2783, 2785, 2787, 2788, 2789, 2790, 2791, 2794, 2796, 2797, 2798, 2799, 2800, 2801, 2803, 2804, 2805, 2807, 2809, 2811, 2815, 2816, 2817, 2818, 2819, 2820, 2822, 2824, 2825, 2826, 2827, 2828, 2829, 2830, 2831, 2832, 2833, 2834, 2835, 2836, 2837, 2838, 2839, 2840, 2841, 2842, 2843, 2844, 2845, 2846, 2847, 2848, 2849, 2850, 2851, 2852, 2853, 2855, 2856, 2857, 2858, 2859, 2860, 2861, 2862, 2863, 2864, 2865, 2866, 2868, 2869, 2870, 2871, 2872, 2873, 2874, 2875, 2876, 2878, 2879, 2880, 2881, 2882, 2883, 2884, 2885, 2886, 2887, 2888, 2889, 2890, 2891, 2892, 2893, 2894, 2896]
reported_severities	Selection of compiler messages by severity.	['warning',]
show_include_path	If True, violations regarding nested includes print the #include path.	False
standards	List of allowed frontend command-line options to select language standard.	['--c99', '--c89']
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

Possible Messages

Name	Message
access_control	Number of access control declarations is higher than translation limit (which is {})
at_quick_exit_functions	Number of functions registered with at_quick_exit() is higher than translation limit (which is {})
atexit_functions	Number of functions registered with atexit() is higher than translation limit (which is {})
base_classes	Number of direct and indirect base classes is higher than translation limit (which is {})
call_arguments	Number of arguments is higher than translation limit (which is {})
catch_handlers	Number of catch handlers is higher than translation limit (which is {})
chars_in_line	Number of characters in logical line is higher than translation limit (which is {})
chars_in_string	Number of characters in string is higher than translation limit (which is {})
cpp_members	Number of members is higher than translation limit (which is {})
data_members	Number of data members is higher than translation limit (which is {})
direct_bases	Number of direct base classes is higher than translation limit (which is {})
enumerators	Number of enumerators is higher than translation limit (which is {})
expr_in_core_constant	Number of full expressions evaluated within a core constant expression is higher than translation limit (which is {})

external_identifiers	Number of external identifiers in this unit is higher than translation limit (which is {})
final_virtual_functions	Number of final overriding virtual functions is higher than translation limit (which is {})
friends	Number of friend declarations is higher than translation limit (which is {})
function_parameters	Number of parameters is higher than translation limit (which is {})
macro_arguments	Number of macro arguments is higher than translation limit (which is {})
macro_parameters	Number of macro parameters is higher than translation limit (which is {})
macros	Number of macro definitions in this unit is higher than translation limit (which is {})
member_initializers	Number of member initializers is higher than translation limit (which is {})
nested_blocks	Number of nested blocks is higher than translation limit (which is {})
nested_composites	Nesting level of structs and unions is higher than translation limit (which is {})
nested_declarators	Number of pointer/array/function declarators is higher than translation limit (which is {})
nested externals	Number of nested external specifications is higher than translation limit (which is {})
nested_includes	Nesting level of #includes is higher than translation limit (which is {})
nested_preprocessor_ifs	Nesting level of #if is higher than translation limit (which is {})
nested_template_instantiations	Number of nested template instantiations is higher than translation limit (which is {})
object_size	Number of bytes in object is higher than translation limit (which is {})
parenthesized_declarators	Number of parenthesized declarators is higher than translation limit (which is {})
parenthesized_expressions	Number of parenthesized subexpressions is higher than translation limit (which is {})
placeholders	Number of placeholders is higher than translation limit (which is {})
recursive_constexpr	Number of recursive constexpr function invocations is higher than translation limit (which is {})
scope_qualifications	Number of scope qualifications is higher than translation limit (which is {})
significant_chars_external	Number of significant initial characters for external identifier is higher than translation limit (which is {})
significant_chars_internal	Number of significant initial characters for internal identifier is higher than translation limit (which is {})
static_members	Number of static members is higher than translation limit (which is {})
std_compile	You have to compile with option {}
switch_cases	Number of cases is higher than translation limit (which is {})
template_params	Number of template parameters is higher than translation limit (which is {})
throw_spec	Number of throw specifications is higher than translation limit (which is {})
variables_in_block	Number of variables in block is higher than translation limit (which is {})
virtual_bases	Number of virtual base classes is higher than translation limit (which is {})

MisraC2012-1.2

Language extensions should not be used.

Input: IR

Source languages: Assembler, C

Configuration

Name	Explanation	Value
extensions	List of PIR classes which are considered a language extension.	('Imaginary_Literal', 'Complex_Literal', 'Fixed_Point_Literal', 'UPC_Literal', 'Address_Of_Label_Literal', 'Label_Difference_Literal', 'Asm_Operand', 'Statement_Expression', 'Min_Operator', 'Max_Operator', 'Assume_Operator', 'Real_Part', 'Imaginary_Part', 'Complex_Conjugation', 'MS_Gcnew_Operator', 'Instanceof_Operator', 'Uuidof_Operator', 'Event_Raise_Call', 'Local_Label_Declaration', 'Managed_Record_Type_Declaration', 'Typeof_Selection', 'Fixed_Point_Type_Selection', 'Vector_Type_Definition', 'Named_Address_Space_Qualified_Type_Definition', 'Managed_Handle_Type_Definition', 'Managed_Record_Type_Definition', 'Declaration_With_Assembler_Name', 'Declaration_With_Register', 'MS_For_Each_Loop', 'Try_Finally_Statement', 'Try_Except_Statement', 'Try_Catch_Finally_Statement', 'Computed_Goto_Statement', 'Leave_Statement', 'GNU_Asm_Statement', 'Super_Qualifier', 'Gnu_Attribute_Section', 'Ms_Attribute_Section', 'Declspec_Attribute', 'Pragma')
nonstandard_messages	Compiler messages to report as language extensions.	[102, 228, 230, 382, 398, 450, 451, 494, 504, 518, 605, 619, 620, 667, 668, 669, 731, 765, 799, 802, 837, 948, 949, 991, 993, 1023, 1102, 1103, 1156, 1211, 1235, 1285, 1297, 1342, 1372, 1395, 1399, 1427, 1553, 1565, 1569, 1570, 1572, 1584, 1604, 1605, 1606, 1607, 1608, 1610, 1611, 1612, 1613, 1614, 1615, 1616, 1617, 1618, 1630, 1717, 1727, 1766, 1802, 1816, 1817, 1818, 1901, 2019, 2068, 2220, 2252, 2329, 2368, 2372, 2380, 2409, 2410, 2480, 2613, 2620, 2626]

Possible Messages

Name	Message
compiler_reported_extension	Language extensions should not be used: {}
use_of_language_extension	Language extensions should not be used

MisraC2012-1.3

There shall be no occurrence of undefined or critical unspecified behaviour.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_extra_args	Whether to allow additional arguments that are not used by the format string.	True
allow_gnu_extensions	Whether to allow the GNU extensions to format specifications.	False
allow_unknown_specs	Whether to allow unknown format specifications. It may be necessary to set this option when using implementation-specific extensions. Arguments are not checked when the format string contains unknown format specifications.	False
compiler_messages_on_undefined	Compiler messages of these types are reported.	[1, 1665, 1816, 1817, 1818, 7, 172, 549, 70, 138, 120, 171, 8, 40, 968, 1037, 192, 1282, 137, 167, 157, 28, 59, 57]
functions	A dictionary mapping the names of the functions to check, to a triple {function_kind, fmt_param_index, arg_start_index} where function_kind is either 'printf' or 'scanf', fmt_param_index is the index of the format-string parameter, and arg_start_index is the index of the first variadic argument.	dict(...)

Possible Messages

Name	Message
arg_type_mismatch	{ } expects argument of type '{ }', but argument { } has type '{ }'
buffer_too_small	{ } may write up to { } characters to buffer of size { }.
empty_struct	Empty struct has undefined behavior
empty_union	Empty union has undefined behavior
invalid_conversion	Invalid or non-standard conversion specification
matching_arg_expected	{ } expects a matching '{ }' argument
missing_inline_def_in_multiple_units	Inline function is missing definition in { } compilation units (e.g. '{ }')
missing_inline_def_in_single_unit	Inline function is missing definition in compilation unit '{ }'
precision_for_conversion	Precision must not be used with %{} conversion specifier
struct_without_named	struct without named members has undefined behavior
too_many_args	Too many arguments for format.
undefined_behaviour	Undefined behaviour: { }
union_without_named	union without named members has undefined behavior
unknown_buffer_size	Potential buffer overflow: { } used with buffer of unknown size.
unlimited_read	Potential buffer overflow: { } has no limit on amount of characters read.
unsupported_assignment_suppression	%n does not support assignment suppression
unsupported_field_width	%n does not support field width
unsupported_flags	%n does not support flags
unsupported_flags_modifiers	Cannot use any flags or modifiers with '%%'
unsupported_hash	%{} does not support the '#' flag
unsupported_i_flag	%{} does not support the 'l' flag
unsupported_length_modifier	%{} does not support the '{ }' length modifier
unsupported_tick	%{} does not support the "" flag
unsupported_zero	%{} does not support the '0' flag

MisraC2012-2.1

A project shall not contain unreachable code.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
report_dead_if_branch_with_const_literal_condition	Whether bodies of if-branches with constant literal conditions should be reported.	True

Possible Messages

Name	Message
unreachable_code	Unreachable code

MisraC2012-2.2

There shall be no dead code.

Input: IR

Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
allow_void_var	Whether {void}var; should be allowed or reported.	True
assume_effect_for_all_calls	Whether all calls should be seen as having a side-effect.	False
assume_effect_when_undefined	Whether all calls to undefined (external) functions should be seen as having a side-effect.	True
list_side_effect_free_statements	Whether toplevel expressions from completely side-effect free statements should be listed as well.	False
report_dead_initializations	Whether initializations may be reported as dead.	False
report_do_while_false	Whether do ... while{0} should be reported.	True
report_more_removable_statements	Whether more (transitively) removable statements should be detected	False
report_while_true	Whether while{1} ... should be reported.	True
tolerate_void_cast	Whether a {void} cast is accepted.	True

Possible Messages

Name	Message
conditional_unused_def	Result of assignment is not used along some path(s) (disabled)
dead_false_branch	Redundant code, condition is always true
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Redundant code, parameter condition is always true
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Redundant code, parameter comparison to NULL is always true
dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Redundant code, parameter comparison to NULL is always false
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Redundant code, parameter condition is always false
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Redundant code, condition is always false
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Redundant code, variable condition is always true
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Redundant code, variable condition is always false
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead
init_used_in_other_isr	Initialization is only used in some interrupt handler
no_effect	Non-null statement without side-effect
removable_declaration	Declaration can be removed
removable_expression	Expression (but not necessarily all subexpressions) can be removed without affecting program behaviour
removable_statement	Statement can be removed
unused_def	Dead (redundant) code: result of assignment is not used
unused_init	Unused initialization
used_in_other_isr	Result of assignment is only used in some interrupt handler

MisraC2012-2.3

A project should not contain unused type declarations.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
unused_typedef	Unused type declaration (typedef'ed type not referenced)

MisraC2012-2.4

A project should not contain unused tag declarations.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
unused_tag	Unused tag declaration

MisraC2012-2.5

A project should not contain unused macro declarations.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
only_check_in_primary_file	Whether only macros defined in primary file should be checked.	False

Possible Messages

Name	Message
unused_macro	Unused macro declaration

MisraC2012-2.6

A function should not contain unused label declarations.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
unused_label	Unused label declaration

MisraC2012-2.7

There should be no unused parameters in functions.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
inspect_nonvirtual_functions	Whether parameters of non-virtual functions should be checked.	True
inspect_virtual_functions	Whether parameters of virtual functions should be checked. Violations will be reported in the base class when none of the derived classes make use of the parameter.	True
report_unnamed_parameters	Whether unnamed unused parameters should be reported as well.	True

Possible Messages

Name	Message
unused_parameter	Unused parameter

MisraC2012-3.1

The character sequences /* and // shall not be used within a comment.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
check_cpp_comments	Whether C++-style comments should be checked.	True
files_to_check	Files to apply this check to (Primary_File / User_Include_File / System_Include_File)	{'Primary_File', 'User_Include_File'}

Possible Messages

Name	Message
nested_c_comment	Comment containing /* sequence.
nested_cpp_comment	Comment containing // sequence.

MisraC2012-3.2

Line-splicing shall not be used in // comments.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
line_splicing_in_cpp_comment	Line-splicing shall not be used in // comments.

MisraC2012-4.1

Octal and hexadecimal escape sequences shall be terminated.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
unterminated_hex_escape_sequence	Hexadecimal escape sequence not terminated.
unterminated_octal_escape_sequence	Octal escape sequence not terminated.

MisraC2012-4.2

Trigraphs should not be used.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allow_in_comments	Allow the use of trigraphs in comments	True

Possible Messages

Name	Message
trigraph_use	Use of trigraph.

MisraC2012-5.1

External identifiers shall be distinct.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
maxlen	Number of significant characters in external identifiers	31
report_external_identifiers	Whether external identifiers should be compared to each other.	True
report_internal_identifiers	Whether internal identifiers should be compared to each other (including macros).	False
report_short_identifiers	If True, identifiers shorter than maxlen are considered as well.	True

Possible Messages

Name	Message
external_identifiers_not_distinct	External identifiers not distinct.
external_identifiers_sharing	Identifiers sharing first {} characters.
internal_identifiers_not_distinct	Internal identifiers not distinct.
internal_identifiers_sharing	Internal identifiers sharing first {} characters.

MisraC2012-5.2

Identifiers declared in the same scope and name space shall be distinct.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
maxlen	Number of significant characters in identifiers	63

Possible Messages

Name	Message
name_reused_in_same_c_name_space	Identifiers sharing first {} characters.

MisraC2012-5.3

An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allow_in_classes	Whether identifiers in local classes can hide surrounding ones	False
allow_in_lambdas	Whether identifiers in lambdas can hide surrounding ones	False
allow_in_local_classes	Whether identifiers in local classes can hide surrounding ones	False
allow_in_namespaces	Whether identifiers in namespaces can hide surrounding ones	False
allow_parameter_in_function_declaration	Whether identifiers of parameters in function declarations may hide surrounding ones, given the function declaration uses a different identifier for the same parameter that does not hide a surrounding one.	False
check_inheritance_hiding	Whether to check for hiding by inheritance.	False
check_nested_classes	Whether symbols from inner classes should be reported as hiding symbols of outer classes.	False
distinguish_name_spaces	Whether C name spaces should be distinguished.	True
hiding_entities	LIR node types to consider symbols that hide other symbols.	{'Variable', 'Typedef_Type', 'Field', 'Parameter'}
maxlen	Number of significant characters in identifiers	63
tolerate_constructor_parameters_hiding_fields	Allow constructor parameters to hide fields	True
tolerate_function_prototype_argument_hiding	Whether to tolerate arguments of a function prototype, that itself is a function argument, that have the same identifier as a argument of the outer function. If `True` is given, the rule will still report cases, where argument identifiers match up and the function prototype refers to arguments of the outer function (e.g. by `decltype()`).	False
tolerate_macro_local_identifier	Whether an identifier being local to a macro is allowed to hide identifiers from outside the macro	False
unchecked_types	Deprecated legacy option, kept to avoid breaking old configurations.	{'Routine_Call', 'Stack_Object_Definition'}

Possible Messages

Name	Message
hiding	{ } hides { }

MisraC2012-5.4

Macro identifiers shall be distinct.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
global_check	If True, the names are compared system-wide, otherwise the MisraC2012 version is used.	True
ignore_command_line_macros	If global_check is False: whether conflicts with command-line macros should be reported	False
maxlen	Number of significant characters in identifiers	63

Possible Messages

Name	Message
duplicate_macro_parameter	Macro identifiers shall be distinct.
macro_identifiers_not_distinct	Macro identifiers shall be distinct.
macro_name_conflict_with_parameter	Macro name and name of parameter of other currently-defined macro shall be distinct.

MisraC2012-5.5

Identifiers shall be distinct from macro names.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
global_check	If True, the names are compared system-wide, otherwise only against macros in the same unit.	True
ignore_command_line_macros	If global_check is False: whether conflicts with command-line macros should be reported	False
maxlen	Number of significant characters in identifiers	63

Possible Messages

Name	Message
identifier_clashes_with_macro	Identifiers shall be distinct from macro names.

MisraC2012-5.6

A typedef name shall be a unique identifier.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
tolerate_macros	Whether #define and #undef using the typedef's name is allowed	True
tolerate_typedef_entity	Whether the entity forming the typedef's underlying type can have the same name.	True

Possible Messages

Name	Message
reused_typedef	A typedef name shall be a unique identifier.

MisraC2012-5.7

A tag name shall be a unique identifier.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
check_types_only_with_external_linkage	Whether the types should only be checked when they have external linkage	False
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	If provided, only messages of these types are reported.	[469]
reported_severities	List of severities to display.	{'error', 'warning', 'remark'}
tolerate_typedef	Whether a typedef to the tag may have the same name.	True
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

Possible Messages

Name	Message
composite_defined_multiple_times	Composite type has multiple definitions.
enum_defined_multiple_times	Enumeration type has multiple definitions.
reused_tag	A tag name shall be a unique identifier.

MisraC2012-5.8

Identifiers that define objects or functions with external linkage shall be unique.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_static_no_linkage	Whether to allow same identifiers for static objects without linkage (e.g. function local static objects)	False
func_filter	Restricts which functions are considered.	has_external_linkage(['node'])
use_unqualified_names	If set to True, ns1::Type is considered equal to ns2::Type.	True
var_filter	Restricts which variables are considered.	has_external_linkage(['node'])

Possible Messages

Name	Message
reused_global_function_name	Identifiers that define objects or functions with external linkage shall be unique.
reused_global_variable_name	Identifiers that define objects or functions with external linkage shall be unique.

MisraC2012-5.9

Identifiers that define objects or functions with internal linkage should be unique.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_static_no_linkage	Whether to allow same identifiers for static objects without linkage (e.g. function local static objects)	False
func_filter	Restricts which functions are considered.	has_internal_linkage(['node'])
use_unqualified_names	If set to True, ns1::Type is considered equal to ns2::Type.	True
var_filter	Restricts which variables are considered.	has_internal_linkage(['node'])

Possible Messages

Name	Message
reused_global_function_name	Identifiers that define objects or functions with internal linkage should be unique.
reused_global_variable_name	Identifiers that define objects or functions with internal linkage should be unique.

MisraC2012-6.1

Bit-fields shall only be declared with an appropriate type.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
accept_system_typedef	Whether standard typedefs like int32_t are accepted if from a system header even when they do not use an explicit sign for the underlying type	True

Possible Messages

Name	Message
missing_bitfield_sign	Bit-field type should be an explicitly unsigned or signed integral type.

MisraC2012-6.2

Single-bit named bit fields shall not be of a signed type.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
signed_single_bitfield	Signed bit field shall be at least 2 bits long.

MisraC2012-7.1

Octal constants shall not be used.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
octal_literal	Use of octal literal.

MisraC2012-7.2

A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
check_only_hex_and_octal	Whether only hex and octal literals should be checked.	False
consider_conversion	Selects whether literals with an implicit conversion from signed to unsigned are reported.	False
consider_pp_expressions	Whether literals in #if conditions are checked.	True
report_all_unsuffixed_hex_and_octal	Whether hex and octal literals should always be reported when they are unsuffixed, independent of the environment	False

Possible Messages

Name	Message
missing_hex_suffix	Hexadecimal and octal constants require "U" suffix
missing_u_suffix	Constant of unsigned type requires "U" suffix

MisraC2012-7.3

The lowercase character "l" shall not be used in a literal suffix.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
lowercase_l_suffix	Lowercase "l" should not be used in a literal suffix

MisraC2012-7.4

A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
nonconst_string_literal	String literal should only be used as 'const char*'

MisraC2012-8.1

Types shall be explicitly specified.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
implicit_int	Type shall be explicitly specified

MisraC2012-8.2

Function types shall be in prototype form with named parameters

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
implicit_function_return_type	Function shall have explicit return type
missing_parameter_type	Function should not use old-style parameter declaration list
parameterless_func_without_void_param	Function with no parameters shall be declared with (void)
unnamed_parameter	Function parameter shall have a name

MisraC2012-8.3

All declarations of an object or function shall use the same names and type qualifiers.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_parameter_type_qualifier_differences	If enabled, do not report a violation for different top-level type qualifiers in parameters: int f(int p); int f(const int p) {} Top-level qualifiers are only relevant for the implementation of the function and have no effect on callers, so it is not necessary to replicate them in the header file.	False
check_undefined	Whether only-declared routines should also be checked.	False
ignore_casing	Whether to ignore casing for the check.	False
include_redefinitions	Whether declarations of redefining methods should be considered together with the redefined ones.	False
inspect_only_defined_routines	If False, re-declarations of undefined functions are compared to earlier declarations as well.	True
require_exact_match	Whether to check for identical or compatible types.	True

Possible Messages

Name	Message
decl_return_type_mismatch	Return type at declarations differs
parameter_name_mismatch	Parameter name at definition differs from name at declaration
parameter_type_mismatch	Parameter type at definition differs from type at declaration
return_type_mismatch	Return type at definition differs from type at declaration

MisraC2012-8.4

A compatible declaration shall be visible when an object or function with external linkage is defined.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
exclude_main	If True, no message for main() will be reported.	True

Possible Messages

Name	Message
missing_defined_variable_declaration	Missing declaration for object definition
missing_visible_compatible_declaration_for_defined_variable	A compatible declaration shall be visible when an object with external linkage is defined
missing_visible_compatible_declaration_for_funcdef	A compatible declaration shall be visible when a function with external linkage is defined
undeclared_defined_function	Missing declaration for function definition

MisraC2012-8.5

An external object or function shall be declared once in one and only one file.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_only_one_decl	Enables message if there are multiple declarations but all in same file.	True

Possible Messages

Name	Message
function_declared_in_multiple_files	Function declared in more than one file.
function_declared_multiple_times	Function declared more than once.
type_declared_in_multiple_files	Type declared in more than one file.
type_declared_multiple_times	Type declared more than once.
variable_declared_in_multiple_files	Variable declared in more than one file.
variable_declared_multiple_times	Variable declared more than once.

MisraC2012-8.6

An identifier with external linkage shall have exactly one external definition.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_undefined_specialized_template	Whether undefined specialized templates are tolerated when applied to C++ code.	True
allow_unselected_declarations	Select whether to report declarations with no definition if they are unused.	False
allowed_undefined_functions	Regular expressions for functions which are tolerated without a definition, e.g. compiler intrinsics.	[_.^*]
allowed_undefined_variables	Regular expressions for variables which are tolerated without a definition.	[_.^*]

Possible Messages

Name	Message
function_defined_multiple_times	Type, object or function has multiple definitions.
undefined_function	Type, object or function without definition
undefined_variable	Type, object or function without definition
variable_defined_multiple_times	Type, object or function has multiple definitions.

MisraC2012-8.7

Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allow_moving_to_other_primary_file	When a variable is only used in a single file X, but currently implemented in a different file Y, this controls whether the check suggests to move it into X and make it static there.	False
exclude_dllexport	If True, no suggestions will be produced to make functions marked as dllexport or dllimport static in a primary file.	True
exclude_function_locals	If True, variables that could even be function-local are not reported (this avoids overlapping messages with rule 8.9).	True
exclude_undefined	Whether only-declared symbols should be reported as well	True
template_args_can_be_static	Whether functions whose address is used as template argument can be made static (some compilers, like Microsoft's, don't allow it then).	False

Possible Messages

Name	Message
function_file_static	{ } can be declared static in primary file.
var_file_static	{ } can be declared static in primary file.

MisraC2012-8.8

The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
function_decl_missing_static	Declaration should use static storage class specifier
function_def_missing_static	Definition should use static storage class specifier
variable_decl_missing_static	Declaration should use static storage class specifier
variable_def_missing_static	Definition should use static storage class specifier

MisraC2012-8.9

An object should be defined at block scope if its identifier only appears in a single function.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
move_global_const_into_function	Controls suggestions for global constants that could be declared locally in a function.	True
only_check_unit_locals	Whether only global static variables should be checked.	False

Possible Messages

Name	Message
locality_function	Global {} can be declared inside function.

MisraC2012-8.10

An inline function shall be declared with the static storage class.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
nonstatic_inline_function	An inline function shall be declared with the static storage class

MisraC2012-8.11

When an array with external linkage is declared, its size should be explicitly specified.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
only_extern	Whether to consider only extern declared arrays	False
report_definitions	Whether definitions of array variables should also be reported	False

Possible Messages

Name	Message
unbounded_array	Array declaration with unknown size.

MisraC2012-8.12

Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
duplicate_enum_value	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique

MisraC2012-8.13

A pointer should point to a const-qualified type whenever possible.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
check_all_variables	Whether non-parameter variables should be checked as well.	True
ignore_forwarding_references	Whether to allow C++11 forwarding references (universal references) even if all template instances would also work with a const lvalue reference.	True
ignore_move_constructors	Whether to allow non-const rvalue references on constructors.	True
report_overrides	Whether to report violations on overrides. For virtual methods, this rule will report a violation only if the parameter can be marked const in all related methods in the inheritance hierarchy. Thus, the same violation is reported for each method in the inheritance hierarchy. You can set this option to False to report such violations only for the base methods.	True

Possible Messages

Name	Message
parameter_can_point_to_const	{} can be declared as pointer/reference to const.

MisraC2012-8.14

The restrict type qualifier shall not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
restrict_qualifier	The restrict type qualifier shall not be used

MisraC2012-9.1

The value of an object with automatic storage duration shall not be read before it has been set.

Input: IR

Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
perform_global_analysis	Whether global analysis (using iranalysis) should be done, or just a local analysis.	True

Possible Messages

Name	Message
pass_as_pointer_to_const_param	Passing uninitialized variable by pointer as function parameter with pointer-to-const type
possible_uninit	Use of possibly uninitialized variable
uninit	Use of uninitialized variable

MisraC2012-9.2

The initializer for an aggregate or union shall be enclosed in braces.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allowed_classes_without_braces	For the listed class types, don't require braces. This is useful for classes that transparently wrap a single contained value. For example, with this option "std::array<int, 3> a = {1,2,3};" is OK even though there could be two sets of braces (one for "std::array", the other and for its inner "int[3]" array).	['std::array']

Possible Messages

Name	Message
missing_aggregate_initialization_braces	Use braces to indicate structure of initialized entity

MisraC2012-9.3

Arrays shall not be partially initialized.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_any_zero_initialization	Whether {0} is allowed in nested aggregates as well.	True
allow_toplevel_zero_initialization	Whether {0} is allowed as top-level initialization for the complete aggregate	True

Possible Messages

Name	Message
incomplete_aggregate_initialization	Provide explicit initializer for each part of initialized entity

MisraC2012-9.4

An element of an object shall not be initialized more than once.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
array_element_initialized_multiple_times	Array element initialized more than once.
field_initialized_multiple_times	Field initialized more than once.

MisraC2012-9.5

Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
unbounded_array_with_designator	Size of array shall be specified explicitly when using designated initializers

MisraC2012-10.1

Operands shall not be of an inappropriate essential type.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
report_bitwise_ops	Controls whether to report 'Operand of bitwise operator should be essentially unsigned.'	True

Possible Messages

Name	Message
bool_used_numerically	{} is of essentially Boolean type, but is being used as a numeric value.
character_used_numerically	{} is of essentially character type, but is being used as a numeric value.
enum_used_in_arithmetic	{} is of essentially enum type, but is being used in an arithmetic operation.
float_used	{} must not have essentially floating type.
should_be_bool	{} should be essentially Boolean.
should_be_unsigned	{} should be essentially unsigned.
unary_minus_on_unsigned	Operand of unary '-' should not be of essentially unsigned type.

MisraC2012-10.2

Expressions of essentially character type shall not be used inappropriately in addition or subtraction operations

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
char_used_in_addition	Inappropriate use of essentially character type.
char_used_in_subtraction	Inappropriate use of essentially character type.

MisraC2012-10.3

The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
assignment_to_other_type	Inappropriate implicit conversion from {} to {}
case_type_mismatch	Case label of essential type {} does not match switch controlling expression of type {}

MisraC2012-10.4

Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allow_signed_constants_in_unsigned_context	If this option is enabled, the rule is relaxed to allow using signed constant integer expressions where an unsigned integer is expected.	False

Possible Messages

Name	Message
different_operand_type_categories	Different essential type categories used in operator: {} and {}

MisraC2012-10.5

The value of an expression should not be cast to an inappropriate essential type

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes], [EnumTypes]], [[SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes], [BoolTypes]], [[BoolTypes], [SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes]], [[CharacterTypes], [FloatingTypes]], [[FloatingTypes], [CharacterTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
type_category	Selection of type categories to consider for type changes inside same category.	[EnumTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	EssentialTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Explicit conversion to inappropriate essential type
cast_changes_type_inside_category	Inappropriate explicit conversion to other essential enum type

MisraC2012-10.6

The value of a composite expression shall not be assigned to an object with wider essential type

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
composite_assigned_to_wider	Composite expression of type {} should not be assigned to wider type {}

MisraC2012-10.7

If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
composite_operand_and_wider_operand	Result of composite expression of type {} is implicitly converted to type {}

MisraC2012-10.8

The value of a composite expression shall not be cast to a different essential type category or a wider essential type

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes], [SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	True
report_narrowing	Whether narrowing casts should be reported.	False
report_widening	Whether widening casts should be reported.	True
show_operand_in_entity	Whether entity should be from->to or from->to on operand	True
size_source_category	Selection of type categories of the operand being cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
size_target_category	Selection of type categories appearing as target types of the cast.	[SignedTypes, UnsignedTypes, FloatingTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	EssentialTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Composite expression is cast to different essential type category
narrowing_cast	Conversion to smaller type
widening_cast	Composite expression is cast to wider essential type

MisraC2012-11.1

Conversions shall not be performed between a pointer to a function and any other type.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[FunctionPointerTypes], [SignedTypes, UnsignedTypes, FloatingTypes, CharacterTypes, BoolTypes, EnumTypes, VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes, NullPointerTypes, OtherTypes]], [[SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes, VoidTypes, OtherTypes, VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes], [FunctionPointerTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	True
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
type_category	Selection of type categories to consider for type changes inside same category.	[FunctionPointerTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion between function pointer and other type
cast_changes_type_inside_category	Conversion between incompatible function pointer types

MisraC2012-11.2

Conversions shall not be performed between a pointer to an incomplete type and any other type.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[IncompletePointerTypes], [SignedTypes, UnsignedTypes, FloatingTypes, CharacterTypes, BoolTypes, EnumTypes, VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes, NullPointerTypes, OtherTypes]], [[VoidPointerTypes, IncompletePointerTypes, FunctionPointerTypes, ObjectPointerTypes, SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes, VoidTypes, OtherTypes], [[IncompletePointerTypes]]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	True
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
type_category	Selection of type categories to consider for type changes inside same category.	[IncompletePointerTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion between pointer to incomplete type and other type
cast_changes_type_inside_category	Conversion between pointer to incomplete type and other pointer to incomplete type

MisraC2012-11.3

A cast shall not be performed between a pointer to object type and a pointer to a different object type.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
type_category	Selection of type categories to consider.	[ObjectPointerTypes]
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_inside_category	Conversion between pointers to object type

MisraC2012-11.4

A conversion should not be performed between a pointer to object and an integer type.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[SignedTypes, UnsignedTypes, BoolTypes, EnumTypes, CharacterTypes], [ObjectPointerTypes]], [[ObjectPointerTypes], [SignedTypes, UnsignedTypes, BoolTypes, EnumTypes, CharacterTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	False
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion between pointer type and integer type

MisraC2012-11.5

A conversion should not be performed from pointer to void into pointer to object.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[VoidPointerTypes], [ObjectPointerTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	True
look_through_casts	If True, $[T^*](void^*)x$ will not be reported.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion between object pointer and void pointer

MisraC2012-11.6

A cast shall not be performed between pointer to void and an arithmetic type.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[VoidPointerTypes], [SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes]], [[SignedTypes, UnsignedTypes, FloatingTypes, BoolTypes, EnumTypes, CharacterTypes], [VoidPointerTypes]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	True
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "[from->to]operand"	False
type_system	Which type system to use: compiler types, underlying types, essential types	CompilerTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion between void pointer and arithmetic type

MisraC2012-11.7

A cast shall not be performed between pointer to object and a non-integer arithmetic type.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
category_changes	List of {from, to} type category pairs to check for. Both from and to are lists of categories.	[[[ObjectPointerTypes], [FloatingTypes, BoolTypes, CharacterTypes, EnumTypes]], [[FloatingTypes, BoolTypes, CharacterTypes, EnumTypes], [ObjectPointerTypes]]]]
check_explicit_casts	Whether explicit casts should be checked and reported.	True
check_implicit_casts	Whether implicit casts should be checked and reported.	True
look_through_casts	If True, operand after stripping casts is used.	False
only_complex_expressions	Whether all operands or only those deemed complex should be inspected.	False
show_operand_in_entity	Whether entity should be "from->to" or "(from->to)operand"	False
type_system	Which type system to use: compiler types, underlying types, essential types	EssentialTypeSystem

Possible Messages

Name	Message
cast_changes_type_category	Conversion between pointer type and non-integer arithmetic type

MisraC2012-11.8

A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
check_const	Whether casts affecting const-qualifiers should be reported	True
check_volatile	Whether casts affecting volatile-qualifiers should be reported	True
only_top_level	Whether the check only applies to the top-level pointee or recursively to deeper pointer levels as well	False

Possible Messages

Name	Message
cast_removes_const	Cast removes const qualification
cast_removes_volatile	Cast removes volatile qualification
implicit_cast_removes_const	Implicit cast removes const qualification
implicit_cast_removes_volatile	Implicit cast removes volatile qualification

MisraC2012-11.9

The macro NULL shall be the only permitted form of integer null pointer constant.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
require_nullptr	Require C++11 nullptr instead of macro NULL or literal zero (0).	False

Possible Messages

Name	Message
zero_as_null	Use of literal zero (0) as null-pointer-constant, use {} instead

MisraC2012-12.1

The precedence of operators within expressions should be made explicit.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
exception_levels	List of operator precedence levels for which no parentheses are required.	[]

Possible Messages

Name	Message
missing_parens_depends_on_precedence_level	Parentheses should be used to avoid dependence on precedence rules
sizeof_missing_parens	Operand of sizeof operator should be enclosed in parentheses

MisraC2012-12.2

The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
report_nonconst_cases	Whether a message should be printed for non-constant cases missing a range-checking if statement	False
use_essential_type	Whether essential type should be computed for allowed value range.	True

Possible Messages

Name	Message
excessive_shift	Right operand of shift operator has value {}, but should be in range 0 to {}
possible_excessive_shift	Right operand of shift operator should be checked to be in range 0 to {}

MisraC2012-12.3

The comma operator should not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
use_of_comma	Use of comma sequence operator.

MisraC2012-12.4

Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
const_expression_overflow	Overflow during evaluation of constant expression.

MisraC2012-12.5

The sizeof operator shall not have an operand which is a function parameter declared as "array of type".

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
sizeof_on_array_param	Operand of "sizeof" shall not be an array parameter

MisraC2012-13.1

Initializer lists shall not contain persistent side effects.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
assume_effect_when_undefined	Whether all calls to undefined [external] functions should be seen as having a side-effect.	True

Possible Messages

Name	Message
side_effect_call_in_initializer_list	Call in initializer list potentially causes side-effect
side_effect_in_initializer_list	Side effect in initializer list

MisraC2012-13.2

The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
ignore_exceptions	Ignore possible exceptions in the computation of side effects. (this option only has an effect if report_calls is enabled)	False
report_calls	If True, reports unsequenced function calls.	False

Possible Messages

Name	Message
unsequenced_call_volatile	Function calls unsequenced with volatile access
unsequenced_calls	Unsequenced function calls
unsequenced_read_write	Unsequenced read and write accesses
unsequenced_volatiles	Unsequenced volatile accesses
unsequenced_writes	Unsequenced writes

MisraC2012-13.3

A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_volatile	Whether incrementing/decrementing a volatile object should be allowed.	False

Possible Messages

Name	Message
decrement_mixed_with_side_effect	Decrement mixed with other side-effect
decrement_volatile	Decrement of volatile object
increment_mixed_with_side_effect	Increment mixed with other side-effect
increment_volatile	Increment of volatile object

MisraC2012-13.4

The result of an assignment operator should not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
assignment_result_used	The result of an assignment operator should not be used.

MisraC2012-13.5

The right hand operand of a logical && or || operator shall not contain persistent side effects.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_volatile_access	Whether access to a volatile variable should be reported as side effect.	False
assume_effect_when_undefined	Whether all calls to undefined [external] functions should be seen as having a side-effect.	True

Possible Messages

Name	Message
call_with_side_effect	Call in right-hand operand of && or may have side-effect
modifies_local_var	Right-hand operand of && or modifies '{}'
side_effect	Right-hand operand of && or has side-effect

MisraC2012-13.6

The operand of the sizeof operator shall not contain any expression which has potential side effects.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
side_effect_in_sizeof	Operand of "sizeof" shall not contain side effects

MisraC2012-14.1

A loop counter shall not have essentially floating type.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
loop_counter_model		MisraC2012LoopCounters

Possible Messages

Name	Message
float_loop_counter	Use of floating-point loop counter.

MisraC2012-14.2

A for loop shall be well-formed.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
loop_counter_model		MisraC2012LoopCountersFromContinue

Possible Messages

Name	Message
loop_condition_missing_counter	Loop condition shall use the loop counter
loop_condition_uses_modified_object	Loop condition shall not use objects modified in the loop body (other than loop control flags)
loop_continue_uses_modified_object	Third clause of for loop shall not use objects modified in the loop body
loop_counter_missing_in_continue	Loop counter must be modified in the third clause of for loop
loop_counter_modified	Loop counter modified within loop
loop_has_potential_side_effect_in_condition	Call in loop condition potentially causes side-effect
loop_has_potential_side_effect_in_continue	Call in third clause of for loop potentially causes side-effect
loop_has_side_effect_in_condition	Side effect in loop condition
loop_has_side_effect_in_continue	Side effect in third clause of for loop
loop_init_missing_counter	For loop initializer shall initialize the loop counter
malformed_continue_expression	Third clause of for loop shall be an expression that modifies the loop counter, and has no other effects.
malformed_endless_loop	Infinite loop must be of form 'for (; ;)'

MisraC2012-14.3

Controlling expressions shall not be invariant.

Input: IR

Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
report_do_while_false	Whether do ... while[0] should be reported.	False
report_while_true	Whether while[1] ... should be reported.	False

Possible Messages

Name	Message
dead_false_branch	Condition is always true: Else branch is dead
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Parameter condition is always true: Else branch is dead
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Parameter comparison to NULL is always true: Else branch is dead
dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Parameter comparison to NULL is always false: Then branch is dead
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Parameter condition is always false: Then branch is dead
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Condition is always false: Then branch is dead
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Variable condition is always true: Else branch is dead
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Variable condition is always false: Then branch is dead
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead

MisraC2012-14.4

The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
report_do_while_false	Whether do ... while[0] should be reported.	True
report_while_true	Whether while[1] ... should be reported.	True

Possible Messages

Name	Message
nonbool_if_condition	Condition should have essentially Boolean type
nonbool_loop_condition	Condition should have essentially Boolean type

MisraC2012-15.1

The goto statement should not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
goto	Do not use goto.

MisraC2012-15.2

The goto statement shall jump to a label declared later in the same function.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
backwards_goto	Label referenced by a goto statement shall be declared later in same function.

MisraC2012-15.3

Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
goto_label_block	Label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

MisraC2012-15.4

There should be no more than one break or goto statement used to terminate any iteration statement.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
inspect_gotos	Whether gotos exiting the loop should be counted.	True

Possible Messages

Name	Message
multiple_loop_breaks	More than one break or goto terminates loop.

MisraC2012-15.5

A function should have a single point of exit at the end.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
multiple_returns	Function shall have a single point of exit at the end of the function.

MisraC2012-15.6

The body of an iteration-statement or a selection-statement shall be a compound-statement.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
dangerous_else	Block or if statement required as else branch.
dangerous_then	Block required as then branch.
loop_body_not_compound	Loop body shall be a statement sequence.
switch_body_not_compound	switch body shall be a statement sequence.

MisraC2012-15.7

All if ... else if constructs shall be terminated with an else statement.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
defensive_else	Controls if the else branch of if-cascade's should contain a suitable comment or not be empty.	True

Possible Messages

Name	Message
non_defensive_else	Last else in the if-cascade must not be empty or contain a comment.
unterminated_if_cascade	Last if in the if-cascade has no else.

MisraC2012-16.1

All switch statements shall be well-formed.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
acceptable_end_items	This set characterizes allowed last items at the end of a switch case.	Exit_Switch
acceptable_middle_items	This set characterizes allowed last items in the middle of a switch case.	('Exit_Switch', 'Switch_Case_Label')
allow_complete_enum	Whether a switch having a case for all enumerators is tolerated without a default.	False
allow_empty_statements	Whether empty statements at the end of switch-case should be ignored.	False
allow_fallthrough_comment	Misra allows a comment with text "fall" and "through".	True
allow_fallthrough_macro	Misra allows __fallthrough as marker for allowed fallthrough.	True
allow_if	Whether an if containing acceptable items at the end of both branches is accepted.	False
allow_noreturn_function_calls	Whether a function call to a [[noreturn]] function is considered a valid termination.	False
comment	The comment that indicates an intentional fallthrough. Can be a list of comments.	[]
disallow_multiple_compound_statements	Whether there can be more than one toplevel compound statement.	True
disallowed_nested_statements	Statement types not allowed inside compound statements within case-clause body.	None
disallowed_toplevel_statements	Statement types not allowed in case-clause body on toplevel.	('Named_Label', 'Unconditional_Branch', 'Declaration_Statement')
minimum_switch_cases	The number of cases a switch statement should at least have.	1

Possible Messages

Name	Message
case_label_nested	A switch label shall only be used in a switch.
default_neither_first_nor_last	"default" clause must be first or last in switch.
empty_default	Default clause should contain a statement or comment.
empty_switch	Switch has no "case" clause.
forbidden_compound_statement_in_case	At most one compound statement is allowed on toplevel in case-clauses.
forbidden_nested_statement_in_case	Statement is not allowed inside compound statement in case-clauses.
forbidden_toplevel_statement_in_case	Statement is not allowed outside a compound statement in case-clauses.
minimum_switch_cases	Switch has too little "case" clauses.
missing_default	Switch has no "default" clause.
switch_body_not_compound	A switch statement shall be a well-formed switch statement.
switch_not_well_formed	Switch has a statement before the first case label.
switch_over_bool	Switch expression is effectively Boolean.
unterminated_case	This switch-case is not terminated with {}, last statement is {}.
unterminated_empty_case	This switch-case is empty, not terminated with {}.

MisraC2012-16.2

A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
case_label_nested	A switch label shall only be used in a switch.

MisraC2012-16.3

An unconditional break statement shall terminate every switch-clause.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
acceptable_end_items	This set characterizes allowed last items at the end of a switch case	Exit_Switch
acceptable_middle_items	This set characterizes allowed last items in the middle of a switch case	('Exit_Switch', 'Switch_Case_Label')
allow_empty_statements	Whether empty statements at the end of switch-case should be ignored.	False
allow_fallthrough_comment	Misra allows a comment with text "fall" and "through".	True
allow_fallthrough_macro	Misra allows __fallthrough as marker for allowed fallthrough.	True
allow_if	Whether an if containing acceptable items at the end of both branches is accepted.	False
allow_noreturn_function_calls	Whether a function call to a [[noreturn]] function is considered a valid termination.	False
comment	The comment that indicates an intentional fallthrough. Can be a list of comments.	[]

Possible Messages

Name	Message
unterminated_case	This switch-case is not terminated with {}, last statement is {}.
unterminated_empty_case	This switch-case is empty, not terminated with {}.

MisraC2012-16.4

Every switch statement shall have a default label.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allow_complete_enum	Whether a switch having a case for all enumerators is tolerated without a default.	False

Possible Messages

Name	Message
empty_default	Default clause should contain a statement or comment.
missing_default	Switch has no "default" clause.

MisraC2012-16.5

A default label shall appear as either the first or the last switch label of a switch statement.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
default_neither_first_nor_last	"default" clause must be first or last in switch.

MisraC2012-16.6

Every switch statement shall have at least two switch-clauses.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
minimum_switch_cases	The number of cases a switch statement should at least have.	1

Possible Messages

Name	Message
empty_switch	Switch has no "case" clause.
minimum_switch_cases	Switch has too few "case" clauses.

MisraC2012-16.7

A switch-expression shall not have essentially Boolean type.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
switch_over_bool	Switch expression is effectively Boolean.

MisraC2012-17.1

The features of <stdarg.h> shall not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header [used for symbols like NULL and size_t which are defined in multiple headers]	['NULL', 'size_t']
header	Name of the header file which should not be used.	stdarg
macros	List of names of forbidden predefined macros.	['va_arg', 'va_start', 'va_end', 'va_copy', 'va_list']

Possible Messages

Name	Message
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.
predefined_macro_invocation	Usage of forbidden entity from <stdarg.h>.

MisraC2012-17.2

Functions shall not call themselves, either directly or indirectly.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
function_cycle	Cycle of recursive functions
recursive_function	Function is directly recursive

MisraC2012-17.3

A function shall not be declared implicitly.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
reference_to_missing_function_prototype	Referenced function needs prototype declaration
reference_to_undeclared_function	Referenced function needs a declaration

MisraC2012-17.4

All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
lambda_return_missing_value	Return without value in non-void lambda expression
missing_return	Non-void function needs return with value at end.
missing_return_in_lambda	Non-void lambda expression needs return with value at end.
return_missing_value	Return without value in non-void function

MisraC2012-17.5

The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
report_argument_too_big	Whether arguments with more elements than necessary should be reported	False
report_pointer_arguments	Whether pointers passed as arguments for array parameters should be reported.	False

Possible Messages

Name	Message
array_argument_size_mismatch	Mismatch in array size between parameter and argument
pointer_argument_for_array_parameter	Passing pointer for array parameter might not have correct array size

MisraC2012-17.6

The declaration of an array parameter shall not contain the static keyword between the [].

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
c99_static_array_param	The declaration of an array parameter shall not contain the static keyword between the []

MisraC2012-17.7

The value returned by a function having non-void return type shall be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allowed_functions	Calls to these functions are ignored.	frozenset(['strncat', 'memmove', 'memset', 'strcat', 'strcpy', 'memcpy', 'strncpy'])
inspect_template_instances	Whether calls in template instances should be reported.	False

Possible Messages

Name	Message
discarded_return_with_entity	Return value of function discarded.
discarded_return_without_entity	Return value of function discarded.

MisraC2012-17.8

A function parameter should not be modified.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
modified_parameter	A function parameter should not be modified.

MisraC2012-18.1

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.

Input: IR
Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
arithmetic_out_of_bounds	Pointer arithmetic on {1} creates pointer outside array bounds of {0}
out_of_bounds	Access into array is out of bounds
possible_indirect_out_of_bounds	Pointer-indirect access through {1} might be out of bounds accessing {0}
possible_out_of_bounds	Access into array might be out of bounds

MisraC2012-18.2

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Input: IR
Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
possible_unrelated_ptr_subtraction	Subtracting possibly unrelated pointers
unrelated_ptr_subtraction	Subtracting unrelated pointers

MisraC2012-18.3

>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.

Input: IR
Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
possible_unrelated_ptr_comparison	Comparing possibly unrelated pointers
unrelated_ptr_comparison	Comparing unrelated pointers

MisraC2012-18.4

The +, -, += and -= operators should not be applied to an expression of pointer type.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
pointer_arithmetic	Pointer arithmetic not allowed

MisraC2012-18.5

Declarations should contain no more than two levels of pointer nesting.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
max_levels	Maximum number of allowed pointer-indirection levels.	2

Possible Messages

Name	Message
too_many_pointer_levels	More than {} level{} of pointer indirection

MisraC2012-18.6

The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.

Input: IR
Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
escaping_address	Escaping address of local variable
possibly_escaping_address	Possibly escaping address of local variable (as target of {1})

MisraC2012-18.7

Flexible array members shall not be declared.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
flexible_array_member	Flexible array members shall not be declared.

MisraC2012-18.8

Variable-length array types shall not be used.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
vla	Variable-length array types shall not be used.

MisraC2012-19.1

An object shall not be assigned or copied to an overlapping object.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
assignment_to_overlapping	An object shall not be assigned to an overlapping object

MisraC2012-19.2

The union keyword should not be used.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allow_tagged_unions	Whether tagged unions (nested in a struct that has an enum discriminator) should be allowed	False

Possible Messages

Name	Message
union	The union keyword should not be used.

MisraC2012-20.1

#include directives should only be preceded by preprocessor directives or comments.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_extern_blocks	Whether to allow includes in `extern` blocks and not report a violation.	False

Possible Messages

Name	Message
include_after_code	#include should only be preceded by comments or preprocessor directives

MisraC2012-20.2

The ', " or \ characters and the /* or // character sequences shall not occur in a header file name.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
check_quotation_mark	Whether system includes should be checked for containing a ".	True
forbidden	The substrings to check for.	set(['//', "'", '\\', """", '/*'])

Possible Messages

Name	Message
nonstandard_include_character	The ', " or \ characters and the /* or // character sequences shall not occur in a header file name.

MisraC2012-20.3

The #include directive shall be followed by either a <filename> or "filename" sequence.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
nonstandard_include_form	The #include directive shall be followed by either a <filename> or "filename" sequence.

MisraC2012-20.4

A macro shall not be defined with the same name as a keyword.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
language_extensions	Additional keywords that must not be redefined by macros.	()

Possible Messages

Name	Message
macro_having_keyword_name	A macro shall not be defined with the same name as a keyword.

MisraC2012-20.5

#undef should not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
undef	#undef should not be used

MisraC2012-20.6

Tokens that look like a preprocessing directive shall not occur within a macro argument.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
directives	Preprocessing directives to check for in macro arguments.	('#if', '#ifdef', '#ifndef', '#elif', '#else', '#endif', '#pragma', '#warning', '#error', '#line', '#include', '#include_next', '#ident', '#region', '#endregion', '#asm', '#endasm', '#define', '#undef')

Possible Messages

Name	Message
pp_directive_as_macro_arg	Macro invocation argument looks like preprocessing directive

MisraC2012-20.7

Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
unsafe_macro_param	Macro parameter not enclosed in parentheses

MisraC2012-20.8

The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
nonbool_pp_if_condition	The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1

MisraC2012-20.9

All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
undefined_macro_used	Zero used for undefined preprocessing identifier

MisraC2012-20.10

The # and ## preprocessor operators should not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
hash_in_macro	The # and ## preprocessor operators should not be used.

MisraC2012-20.11

A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
report_only_concatenated_stringized	If True, only macros where a parameter is stringized and the result used for token concatenation are reported.	True

Possible Messages

Name	Message
concatenated_stringized	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator
more_than_one_macro_hash	Only one # or ## allowed in macro definition

MisraC2012-20.12

A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
report_only_mixed_uses	Whether to report macros that only use the parameter with # or ##	True

Possible Messages

Name	Message
mixed_macro_arg_substitution	Macro parameter {} is used both normally and with # or ## (where macro expansion is suppressed for the macro passed as argument for it at the positions listed below)
unexpanded_macro_as_argument	Macro parameter {} is used with # or ##, so macro expansion is suppressed for the macro passed as argument for it at the positions listed below

MisraC2012-20.13

A line whose first token is # shall be a valid preprocessing directive.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
malformed_pp_directive	Malformed or unknown preprocessing directive

MisraC2012-20.14

All #else, #elif and #endif preprocessing directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
missing_endif	The #endif for this directive is missing
missing_if	The #if for this directive is missing

MisraC2012-21.1

#define and #undef shall not be used on a reserved identifier or reserved macro name.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
additional_reserved_identifiers	Names listed here are seen as violations as well.	set(['assert', 'errno', 'defined'])
allow_reserved_identifier_as_include_guard	If True, macros used for include guards are not checked for being a reserved identifier.	False
check_for_keyword	Whether macro names being keywords should be reported as well.	False

Possible Messages

Name	Message
macro_having_reserved_name	#define of reserved identifier or reserved macro name
undef_of_reserved_name	#undef of reserved identifier or reserved macro name

MisraC2012-21.2

A reserved identifier or macro name shall not be declared.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_function_declarations	Whether nondefining function declarations with library names are allowed	False
check_locals	Whether parameters and local variables should also be checked	True
check_reserved_enum_identifier	Whether enumerator names should be checked to use a reserved identifier	True
check_reserved_function_identifier	Whether function names should be checked to use a reserved identifier	True
check_reserved_type_identifier	Whether type names should be checked to use a reserved identifier	True
check_reserved_variable_identifier	Whether variable names should be checked to use a reserved identifier	True
report_fields	Whether fields using a library name should be reported.	True

Possible Messages

Name	Message
enumerator_having_libname	A reserved identifier or macro name shall not be declared.
field_having_libname	A reserved identifier or macro name shall not be declared.
routine_having_libname	A reserved identifier or macro name shall not be declared.
type_having_libname	A reserved identifier or macro name shall not be declared.
variable_having_libname	A reserved identifier or macro name shall not be declared.

MisraC2012-21.3

The memory allocation and deallocation functions of <stdlib.h> shall not be used

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_base_classes_from	List of path globbing patterns to identify the location of base classes which (together with classes derived from them) should not be reported. For example, a globbing pattern like '/usr/include/*/qt*/Qt*' would allow classes derived from Qt classes if your Qt installation resides there.	[]

Possible Messages

Name	Message
cpp_new_delete	Builtin dynamic memory management operator used.
stdlib_memory_function_call	Dynamic memory management function from <stdlib.h> is called.

MisraC2012-21.4

The standard header file <setjmp.h> shall not be used.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header (used for symbols like NULL and size_t which are defined in multiple headers)	['NULL', 'size_t']
header	Name of the header file which should not be used.	setjmp

Possible Messages

Name	Message
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.

MisraC2012-21.5

The standard header file <signal.h> shall not be used.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header (used for symbols like NULL and size_t which are defined in multiple headers)	['NULL', 'size_t']
header	Name of the header file which should not be used.	signal
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
symbol_header	Name of the header file of which the symbols should not be used.	signal
symbols	Names of symbols which are forbidden.	['sig_atomic_t', 'SIG_DFL', 'SIG_ERR', 'SIG_IGN', 'SIGABRT', 'SIGFPE', 'SIGILL', 'SIGINT', 'SIGSEGV', 'SIGTERM', 'signal', 'raise']
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.

MisraC2012-21.6

The Standard Library input/output functions shall not be used.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
symbol_header	Name of the header file which should not be used.	['stdio', 'wchar']
symbols	Names of symbols which are forbidden.	['fgetc', 'fgets', 'fprintf', 'fputc', 'fputs', 'fread', 'fscanf', 'fwrite', 'getc', 'getchar', 'gets', 'perror', 'printf', 'putc', 'putchar', 'scanf', 'ungetc', 'vfprintf', 'vfscanf', 'vprintf', 'vscanf', 'fgetwc', 'fgetws', 'getwc', 'getwchar', 'fwscanf', 'wscanf', 'fwscanf', 'vwscanf', 'fputwc', 'fputws', 'putwc', 'putwchar', 'fwprintf', 'wpprintf', 'vwprintf', 'ungetwc', 'swprintf', 'swscanf', 'vwscanf', 'vwpprintf', 'vswprintf', 'vswscanf', 'vwscanf', 'fwide', 'remove', 'rename', 'tmpfile', 'tmpnam', 'fclose', 'fflush', 'fopen', 'freopen', 'setbuf', 'setvbuf', 'snprintf', 'sprintf', 'sscanf', 'vsprintf', 'vsprintf', 'vscanf', 'puts', 'fgetpos', 'fseek', 'fsetpos', 'ftell', 'rewind', 'clearerr', 'feof', 'ferror']
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

MisraC2012-21.7

The atof, atoi, atol and atoll functions of <stdlib.h> shall not be used.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
symbol_header	Name of the header file of which the symbols should not be used.	stdlib
symbols	Names of symbols which are forbidden.	['atof', 'atoi', 'atol', 'atoll']
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

MisraC2012-21.8

The library functions abort, exit and system of <stdlib.h> shall not be used.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
symbol_header	Name of the header file of which the symbols should not be used.	stdlib
symbols	Names of symbols which are forbidden.	['abort', 'exit', 'system']
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

MisraC2012-21.9

The library functions bsearch and qsort of <stdlib.h> shall not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
symbol_header	Name of the header file of which the symbols should not be used.	stdlib
symbols	Names of symbols which are forbidden.	['bsearch', 'qsort']
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

MisraC2012-21.10

The Standard Library time and date functions shall not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header (used for symbols like NULL and size_t which are defined in multiple headers)	['NULL', 'size_t']
header	Name of the header file which should not be used.	time
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
symbol_header	Name of the header file of which the symbols should not be used.	('wchar', 'time')
symbols	Names of symbols which are forbidden.	['difftime', 'time', 'clock', 'timespec_get', 'asctime', 'asctime_s', 'ctime', 'ctime_s', 'strftime', 'wcsftime', 'gmtime', 'gmtimes_s', 'localtime', 'localtime_s', 'mktime', 'CLOCKS_PER_SEC', 'tm', 'time_t', 'clock_t', 'timespec']
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h} -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.

MisraC2012-21.11

The standard header file <tgmath.h> shall not be used.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allowed_symbols	Symbols which should be allowed despite being defined in the header (used for symbols like NULL and size_t which are defined in multiple headers)	['NULL', 'size_t']
header	Name of the header file which should not be used.	tgmath

Possible Messages

Name	Message
lib_header_entity_use	Usage of forbidden entity from <{}>.
lib_header_include	Unit includes <{}>.

MisraC2012-21.12

The exception handling features of <fenv.h> should not be used.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
included_headers	Whether the rule should also look in headers included by the configured symbol_header.	True
symbol_header	Name of the header file of which the symbols should not be used.	fenv
symbols	Names of symbols which are forbidden.	[`feclearexcept`, `fegetexceptflag`, `feraiseexcept`, `fesetexceptflag`, `fetestexcept`, `FE_INEXACT`, `FE_DIVBYZERO`, `FE_UNDERFLOW`, `FE_OVERFLOW`, `FE_INVALID`, `FE_ALL_EXCEPT`]
translate_header_name	Whether to auto-translate the `symbol_header` (e.g. stdlib.h) -> cstdlib).	True

Possible Messages

Name	Message
forbidden_libheader_symbol_use	Usage of forbidden entity from <{}>.

MisraC2012-21.14

The Standard Library function memcmp shall not be used to compare null terminated strings.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
memcmp_on_strings	memcmp shall not be used to compare strings, use strncmp instead.

MisraC2012-21.15

The pointer arguments to the Standard Library functions memcpy, memmove and memcmp shall be pointers to qualified or unqualified versions of compatible types.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
incompatible_memcpy_args	Arguments shall be pointers to compatible types.

MisraC2012-21.16

The pointer arguments to the Standard Library function memcmp shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_char	Whether to allow memcmp on 'char' type.	False
allow_composites_without_padding	Whether to allow using memcmp on structs and unions that have no padding bytes.	False
allow_float	Whether to allow memcmp on floating point types.	False

Possible Messages

Name	Message
disallowed_memcmp_pointer_arg	Disallow type of pointer argument.
memcmp_char_pointer_arg	memcmp shall not be used with char pointer argument, use strncmp instead.
memcmp_float	memcmp shall not be used to compare floats as the same value may be stored using different representations.
memcmp_padding	memcmp shall not be used to compare structs with padding.
memcmp_struct_pointer_arg	memcmp shall not be used with struct pointer argument as it would compare padding as well.
memcmp_union_pointer_arg	memcmp shall not be used with union pointer argument as it would compare padding and different kinds of representation.

MisraC2012-21.19

The pointers returned by the Standard Library functions localeconv, getenv, setlocale or, strerror shall only be used as if they have pointer to const-qualified type.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
funcs		{'localeconv', 'getenv', 'setlocale', 'strerror'}

Possible Messages

Name	Message
nonconst_system_pointer_retrievals	Return value should be assigned to a pointer to const-qualified type.
string_of_system_pointer_modified	Return value of call to system function should be considered const, including strings referenced by it

MisraC2012-21.20

The pointer returned by the Standard Library functions asctime, ctime, gmtime, localtime, localeconv, getenv, setlocale or strerror shall not be used following a subsequent call to the same function.

Input: IR

Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
resources	Configuration of system pointer types as resources and allocate/release functions for them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict[...]

Possible Messages

Name	Message
possible_use_after_free	System pointer possibly used after call to (modifying) system function.
use_after_free	System pointer used after call to (modifying) system function.

MisraC2012-22.1

All resources obtained dynamically by means of Standard Library functions shall be explicitly released.

Input: IR

Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)

Possible Messages

Name	Message
memory_leak	Call allocates leaking memory
possible_memory_leak	Call allocates possibly leaking memory

MisraC2012-22.2

A block of memory shall only be freed if it was allocated by means of a Standard Library function.

Input: IR

Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)

Possible Messages

Name	Message
double_free	Dynamic memory released here was already released earlier
possible_double_free	Dynamic memory released here possibly already released earlier
possible_stack_free	{ } possibly released by call to { } is a stack object
possible_use_after_free	Dynamic memory possibly used after it was previously released
possible_wrong_release	Resource possibly released using wrong function (allocation used {0})
stack_free	{ } released by call to { } is a stack object
use_after_free	Dynamic memory used after it was previously released
wrong_release	Resource released using wrong function (allocation used {0})

MisraC2012-22.3

The same file shall not be open for read and write access at the same time on different streams.

Input: IR

Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
modes	Describes when an allocating call opens a file in read or write mode. This is a dict resource -> dict with keys "Resource_Parameter", "Check_Write_Write", and "Modes" (being a dict with key "Read" and "Write")	dict(...)
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)

Possible Messages

Name	Message
mode_conflict	Same file used for both reading and writing
possible_mode_conflict	Same file possibly used for both reading and writing

MisraC2012-22.4

There shall be no attempt to write to a stream which has been opened as read-only.

Input: IR

Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
forbidden	Dict which lists forbidden operations per resource. The mapping gives each case a description which maps to a dict for key "Forbidden_Functions", "Mode_Parameter", "Mode".	dict(...)
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)

Possible Messages

Name	Message
forbidden_operation	Attempt to write to a stream which has been opened as read-only
possibly_forbidden_operation	Possible attempt to write to a stream which has been opened as read-only

MisraC2012-22.5

A pointer to a FILE object shall not be dereferenced.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
file_pointer_dereference	A pointer to a FILE object shall not be dereferenced

MisraC2012-22.6

The value of a pointer to a FILE shall not be used after the associated stream has been closed.

Input: IR

Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)

Possible Messages

Name	Message
possible_use_after_free	FILE pointer possibly used after stream has been closed.
use_after_free	FILE pointer used after stream has been closed.

MisraC2012-22.7

The macro EOF shall only be compared with the unmodified return value from any Standard Library function capable of returning EOF.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
disallowed_eof_comparison	EOF should only be compared to the unmodified return value of a library function.

MisraC2012-22.8

The value of errno shall be set to zero prior to a call to an errno-setting-function.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
errno_clearers		[]
errno_readers		[' perror']
standard_posix		False
variant_cert		False

Possible Messages

Name	Message
uncleared_errno	errno should be set to zero before calling an errno-setting function.

MisraC2012-22.9

The value of errno shall be tested against zero after calling an errno-setting-function.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
errno_readers		[' perror']
standard_posix		False

Possible Messages

Name	Message
missing_errno_check	errno should be tested against zero after this call to an errno-setting function.

MisraC2012-22.10

The value of errno shall only be tested when the last function to be called was an errno-setting-function.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
errno_clearers		[]
errno_readers		[' perror ']
standard_posix		False

Possible Messages

Name	Message
misplaced_errno_check	errno should only be tested after a call to an errno-setting function.

Rules in Group MisraC2012Directive

MisraC2012Directive-1.1

Any implementation-defined behaviour on which the output of the program depends shall be documented and understood.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
files_to_check	Files to be checked for scanner-based part of the rule, e.g. Primary_File or File.	Primary_File
show_include_path	If True, violations regarding nested includes print the #include path.	False

Possible Messages

Name	Message
access_control	Number of access control declarations is higher than translation limit (which is {})
at_quick_exit_functions	Number of functions registered with at_quick_exit() is higher than translation limit (which is {})
atexit_functions	Number of functions registered with atexit() is higher than translation limit (which is {})
base_classes	Number of direct and indirect base classes is higher than translation limit (which is {})
call_arguments	Number of arguments is higher than translation limit (which is {})
catch_handlers	Number of catch handlers is higher than translation limit (which is {})
chars_in_line	Number of characters in logical line is higher than translation limit (which is {})
chars_in_string	Number of characters in string is higher than translation limit (which is {})
cpp_members	Number of members is higher than translation limit (which is {})
data_members	Number of data members is higher than translation limit (which is {})
direct_bases	Number of direct base classes is higher than translation limit (which is {})
enumerators	Number of enumerators is higher than translation limit (which is {})

expr_in_core_constant	Number of full expressions evaluated within a core constant expression is higher than translation limit (which is {})
external_identifiers	Number of external identifiers in this unit is higher than translation limit (which is {})
final_virtual_functions	Number of final overriding virtual functions is higher than translation limit (which is {})
friends	Number of friend declarations is higher than translation limit (which is {})
function_parameters	Number of parameters is higher than translation limit (which is {})
macro_arguments	Number of macro arguments is higher than translation limit (which is {})
macro_parameters	Number of macro parameters is higher than translation limit (which is {})
macros	Number of macro definitions in this unit is higher than translation limit (which is {})
member_initializers	Number of member initializers is higher than translation limit (which is {})
multi_character_char_literal	More than one character in character literal
nested_blocks	Number of nested blocks is higher than translation limit (which is {})
nested_composites	Nesting level of structs and unions is higher than translation limit (which is {})
nested_declarators	Number of pointer/array/function declarators is higher than translation limit (which is {})
nested_externals	Number of nested external specifications is higher than translation limit (which is {})
nested_includes	Nesting level of #includes is higher than translation limit (which is {})
nested_preprocessor_ifs	Nesting level of #if is higher than translation limit (which is {})
nested_template_instantiations	Number of nested template instantiations is higher than translation limit (which is {})
nonbasic_char_in_comment	Using characters which are not from the basic source character set relies on implementation-defined behaviour
nonbasic_char_in_filename	Using characters which are not from the basic source character set relies on implementation-defined behaviour
nonbasic_char_in_pragma	Using characters which are not from the basic source character set relies on implementation-defined behaviour
nonbasic_char_in_string	Using characters which are not from the basic source character set relies on implementation-defined behaviour
object_size	Number of bytes in object is higher than translation limit (which is {})
parenthesized_declarators	Nesting level of parenthesized declarators is higher than translation limit (which is {})
parenthesized_expressions	Nesting level of parenthesized subexpressions is higher than translation limit (which is {})
placeholders	Number of placeholders is higher than translation limit (which is {})
pragma	Using #pragma relies on implementation-defined behaviour
recursive_constexpr	Number of recursive constexpr function invocations is higher than translation limit (which is {})
scope_qualifications	Number of scope qualifications is higher than translation limit (which is {})
shift_right_negative	If first operand is negative, using this bitwise operator relies on implementation-defined behaviour
significant_chars_external	Number of significant initial characters for external identifier is higher than translation limit (which is {})
significant_chars_internal	Number of significant initial characters for internal identifier is higher than translation limit (which is {})
static_members	Number of static members is higher than translation limit (which is {})
switch_cases	Number of cases is higher than translation limit (which is {})
template_params	Number of template parameters is higher than translation limit (which is {})
throw_spec	Number of throw specifications is higher than translation limit (which is {})
variables_in_block	Number of variables in block is higher than translation limit (which is {})
virtual_bases	Number of virtual base classes is higher than translation limit (which is {})

MisraC2012Directive-2.1

All source files shall compile without any compilation errors.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
message_predicate	If provided, a custom predicate to filter relevant messages. Receives the message node and should return True for messages to report	None
reported_messages	If provided, only messages of these types are reported.	None
reported_severities	List of severities to display.	('error',)
use_error_number	Whether the error number from the frontend should be used.	False
use_rule_severity	Whether the rule's severity or the compiler's severity should be used.	True

MisraC2012Directive-4.1

Run-time failures shall be minimized.

Input: IR
Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
abstract_interpretation_div_by_zero	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
abstract_interpretation_overflow	Use abstract-interpretation-based "symbolic expression analysis" as additional postprocessing step.	False
argument_checks	Configuration of [dis]allowed values in addition to what iranalysis.config provides. This is a dict key -> dict with keys 'Functions', 'Parameter', and 'Allowed'/'Disallowed'. The parameter number counting starts at 0. Arguments passed in for this parameters are checked against the value specification given with '[Dis]allowed'.	dict(...)
execute_checks_covered_in_other_rules	Whether iranalysis-based checks covered by other Misra-C:2012 rules should be executed here as well.	False
forbidden	Dict which lists forbidden operations per resource. The mapping gives each case a description which maps to a dict for key "Forbidden_Functions", "Mode_Parameter", "Mode".	dict(...)
modes	Describes when an allocating call opens a file in read or write mode. This is a dict resource -> dict with keys "Resource_Parameter", "Check_Write_Write", and "Modes" (being a dict with key "Read" and "Write")	dict(...)
perform_global_analysis	Whether global analysis (using iranalysis) should be done, or just a local analysis.	True
report_dead_initializations	Whether initialisations may be reported as dead.	True
report_do_while_false	Whether do ... while[0] should be reported.	True
report_while_true	Whether while[1] ... should be reported.	True
resources	Configuration of memory types and allocate/release functions for them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)

Possible Messages

Name	Message
argumentViolation	Argument not within allowed values
arithmetic_out_of_bounds	Pointer arithmetic on {1} creates pointer outside array bounds of {0}
asgn_not_an_enumerator	Assignment of value which does not correspond to an enumerator
asgn_out_of_enum_range	Assignment of value outside the range of the enumerators

cast_overflow	Cast on result of arithmetic computation may cause overflow
cast_truncate	Cast may truncate value
cast_underflow	Cast on result of arithmetic computation may cause underflow
conditional_unused_def	Result of assignment is not used along some path(s)
dead_false_branch	Condition is always true: Else branch is dead
dead_false_branch_type_limits	Condition is always true due to limited range of data type: Else branch is dead
dead_false_branch_type_limits_in_context	Condition is true in context due to limited range of data type: Else branch is dead in the context
dead_param_false_branch	Parameter condition is always true: Else branch is dead
dead_param_false_branch_type_limits	Parameter condition is always true due to limited range of data type: Else branch is dead
dead_param_null_false_branch	Parameter comparison to NULL is always true: Else branch is dead
dead_param_null_false_branch_type_limits	Parameter comparison to NULL is always true due to limited range of data type: Else branch is dead
dead_param_null_true_branch	Parameter comparison to NULL is always false: Then branch is dead
dead_param_null_true_branch_type_limits	Parameter comparison to NULL is always false due to limited range of data type: Then branch is dead
dead_param_true_branch	Parameter condition is always false: Then branch is dead
dead_param_true_branch_type_limits	Parameter condition is always false due to limited range of data type: Then branch is dead
dead_true_branch	Condition is always false: Then branch is dead
dead_true_branch_type_limits	Condition is always false due to limited range of data type: Then branch is dead
dead_true_branch_type_limits_in_context	Condition is false in context due to limited range of data type: Then branch is dead in the context
dead_var_false_branch	Variable condition is always true: Else branch is dead
dead_var_false_branch_type_limits	Variable condition is always true due to limited range of data type: Else branch is dead
dead_var_true_branch	Variable condition is always false: Then branch is dead
dead_var_true_branch_type_limits	Variable condition is always false due to limited range of data type: Then branch is dead
division_by_zero	Division by zero
double_free	Dynamic memory released here was already released earlier
escaping_address	Escaping address of local variable
forbidden_operation	The way in which this resource was allocated forbids this operation
init_used_in_other_isr	Initialization is only used in some interrupt handler
memory_leak	Call allocates leaking memory
mode_conflict	Same file used for both reading and writing
modulo_by_zero	Modulo by zero
null_deref	Pointer is NULL at dereference
out_of_bounds	Access into array is out of bounds
overflow	Arithmetic computation may cause overflow
pass_as_pointer_to_const_param	Passing uninitialized variable by pointer as function parameter with pointer-to-const type
possible_argumentViolation	Argument possibly not within allowed values
possible_division_by_zero	Possible division by zero
possible_double_free	Dynamic memory released here possibly already released earlier
possible_indirect_out_of_bounds	Pointer-indirect access through {1} might be out of bounds accessing {0}

possible_memory_leak	Call allocates possibly leaking memory
possible_mode_conflict	Same file possibly used for both reading and writing
possible_modulo_by_zero	Possible modulo by zero
possible_null_deref	Pointer may be NULL at dereference
possible_out_of_bounds	Access into array might be out of bounds
possible_stack_free	{} possibly released by call to {} is a stack object
possible_uninit	Use of possibly uninitialized variable
possible_unrelated_ptr_comparison	Comparing possibly unrelated pointers
possible_unrelated_ptr_subtraction	Subtracting possibly unrelated pointers
possible_use_after_free	Dynamic memory possibly used after it was previously released
possible_wrong_release	Resource possibly released using wrong function [allocation used {}]
possiblyEscapingAddress	Possibly escaping address of local variable [as target of {}]
possiblyForbiddenOperation	The way in which this resource was allocated possibly forbids this operation
stack_free	{} released by call to {} is a stack object
underflow	Arithmetic computation may cause underflow
uninit	Use of uninitialized variable
unrelated_ptr_comparison	Comparing unrelated pointers
unrelated_ptr_subtraction	Subtracting unrelated pointers
unused_def	Result of assignment is not used
unused_init	Unused initialization
use_after_free	Dynamic memory used after it was previously released
usedInOtherISR	Result of assignment is only used in some interrupt handler
wrong_release	Resource released using wrong function [allocation used {}]

MisraC2012Directive-4.2

All usage of assembly language should be documented.

Input: IR

Source languages: Assembler, C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
useOfAssembler	All usage of assembly language should be documented.

MisraC2012Directive-4.3

Assembly language shall be encapsulated and isolated.

Input: IR

Source languages: Assembler, C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
asm_not_encapsulated	Assembly language shall be encapsulated and isolated.

MisraC2012Directive-4.4

Sections of code should not be "commented out".

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
files_to_check	Files to be checked, e.g. Primary_File or File.	Primary_File
normal_text	Tokens that indicate non-code comments. This is the first step of checking a comment. If any of these words exist in the comment text, it is regarded as not containing code.	{' and ', ' or ', ' but ', ' now ', ' to ', ' is ', ' are ', ' only ', ' be ', ' has ', ' the ', ' with ', ' because ', ' when ', ' oder ', ' und ', '://', '###', 'AXIVION', '++++', '----', '===='}
normal_text_regex	Regular expression to match normal non-code text. This is the second step of checking a comment. If this regex matches the comment text, the comment is regarded as not containing code.	\b[\w\d_]+\s+[\w\d_]+\s+[\w\d_]+\b
suspicious	Tokens that indicate code in comments. This is the final step of checking a comment. If the first two steps did not exclude the comment from further inspection, the comment text is searched for any of these suspicious tokens. The comment is regarded as containing code, if any suspicious token is found.	{' = ', ' == ', ' >= ', ' <= ', '[', ']', '::', '->', '->*', ' ::*', 'if', 'while', 'for', 'if', 'while', 'for', '#pragma', '#else', '#endif', '#if', '#include', '+', '--'}

Possible Messages

Name	Message
code_in_c_comment	Sections of code shall not be commented out using C-style comments.
code_in_cpp_comment	Sections of code should not be commented out using C++ comments.

MisraC2012Directive-4.5

Identifiers in the same name space with overlapping visibility should be typographically unambiguous.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
distinguish_c_name_spaces	Whether to allow similar identifiers in different C name spaces (type tag/label/member/ordinary identifier)	True
distinguish_macro_names	Whether to allow macros to have similar names as other identifiers.	False
distinguish_type_names	Whether to allow types to have similar names as other identifiers. For example, a variable declaration "MyClass myClass;" is allowed by this option. Note: unlike the option 'distinguish_c_name_spaces', this option also distinguishes typedef names, not just struct/class tag names.	False
exclude_system_headers	Whether to exclude identifiers appearing in system headers	False
normalizations	Which pairs of characters should be seen as ambiguous	[('0', 'O'), ('1', 'I'), ('l', 'L'), ('i', 'I'), ('5', 'S'), ('2', 'Z'), ('8', 'B'), ('rn', 'm'), ('n', 'h'), ('_', '')]

Possible Messages

Name	Message
ambiguous	Identifiers are typographically ambiguous
casing	Identifiers only differ in casing

MisraC2012Directive-4.6

typedefs that indicate size and signedness should be used in place of the basic numerical types.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
allow_floating_point	Whether floating point types without should not be reported.	False
ignore_inherited	If true, missing typedefs in inherited methods are not reported.	False
treat_plain_char_as_numerical	Whether the char type without any signedness should be reported.	False

Possible Messages

Name	Message
missing_integer_typedef	Use of base type outside typedef.
wrong_integer_typedef	Use of badly named typedef for base type.

MisraC2012Directive-4.7

If a function returns error information, then that error information shall be tested.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value
inspect_template_instances	Whether calls in template instances should be reported.	False
relevant_functions	If provided, only calls to these functions are inspected.	[]

Possible Messages

Name	Message
discarded_error_with_entity	Integral return value (possibly error code) discarded.
discarded_error_without_entity	Integral return value (possibly error code) discarded.

MisraC2012Directive-4.8

If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden.

Input: IR
Source languages: C

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
composite_can_be_opaque	Implementation of composite type should be hidden in unit {}

MisraC2012Directive-4.9

A function should be used in preference to a function-like macro where they are interchangeable.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
------	-------------	-------

Possible Messages

Name	Message
function_macro_definition	Prefer function over function-like macro

MisraC2012Directive-4.10

Precautions shall be taken in order to prevent the contents of a header file being included more than once.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
macro_name_restrictions	Python iterable of functions with parameters (file, define, macro) to perform additional checks on the macro used as include guard in the given file. Should return None if macro is accepted, else a message to print for the violation. The parameters are: PIR File node for the header, token for the macro #define or #pragma once, and the name of the macro or the string 'pragma once'.	None

Possible Messages

Name	Message
duplicate_include_guard	Duplicate include guard.
include_guard_missing	Missing include guard.
include_guard_partial	Include guard does not cover complete file.
include_guard_with_wrong_define	Bad include guard: #define uses wrong macro, should be {}.
include_guard_without_define	Incomplete include guard: missing #define {}.

MisraC2012Directive-4.11

The validity of values passed to library functions shall be checked.

Input: IR

Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
argument_checks	Configuration of (dis)allowed values in addition to what iranalysis.config provides. This is a dict key -> dict with keys 'Functions', 'Parameter', and 'Allowed'/'Disallowed'. The parameter number counting starts at 0. Arguments passed in for this parameters are checked against the value specification given with '(Dis)allowed'.	dict{...}

Possible Messages

Name	Message
argumentViolation	Argument not within allowed values
possibleArgumentViolation	Argument possibly not within allowed values

MisraC2012Directive-4.12

Dynamic memory allocation shall not be used.

Input: IR

Source languages: C

Configuration

Name	Explanation	Value
allow_base_classes_from	List of path globbing patterns to identify the location of base classes which (together with classes derived from them) should not be reported. For example, a globbing pattern like '/usr/include/*/qt*/Qt/*' would allow classes derived from Qt classes if your Qt installation resides there.	[]

Possible Messages

Name	Message
cpp_new_delete	Builtin dynamic memory management operator used.
stdlib_memory_function_call	Dynamic memory management function from <stdlib.h> is called.

MisraC2012Directive-4.13

Functions which are designed to provide operations on a resource should be called in an appropriate sequence.

Input: IR

Source languages: C

Note: Rule requires CONFIG.Run_IRAnalysis_Checks = True.

Configuration

Name	Explanation	Value
resources	Configuration of resources and operations on them in addition to what iranalysis.config provides. This is a dict resource -> dict with keys 'Allocate', 'Release'.	dict(...)

Possible Messages

Name	Message
missing_second_operation	Unmatched resource operation, counterpart is missing
possibly_missing_second_operation	Possibly unmatched resource operation, counterpart might be missing

Rules in Group Parallelism

Parallelism-IncorrectCriticalRegion

Critical regions must obey certain layout rules.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
check_all_statement_sequences	If set to True, all statement sequence inside routine definitions are checked for correct pairing individually, otherwise only the top-level statement sequence is checked.	False
enter_critical_functions	List of function names to enter a critical region.	[]
enter_critical_macros	List of macro names to enter a critical region (macros must expand to asm() statement).	[]
exit_critical_functions	List of function names to exit a critical region.	[]
exit_critical_macros	List of macro names to exit a critical region (macros must expand to asm() statement).	[]
nested_critical_regions	If set to True, critical regions nest; if set to False, a single exit-critical-region terminates all open critical regions.	True

Parallelism-UnsafeVarAccess

Do not access global variables or static members outside critical region.

Input: IR

Source languages: C, C++

Configuration

Name	Explanation	Value
access_kinds	Access kinds (e.g. Reading_Operand_Interface, Writing_Operand_Interface, Address_Operand_Interface).	('Reading_Operand_Interface', 'Writing_Operand_Interface')
allow_c11_atomics	If set, don't report races on C11 atomic variables.	True
allow_volatile_sig_atomic_t	If set, don't report races on variables of type "volatile sig_atomic_t".	False
debug_output	Option to provide diagnostic output.	False
enter_critical_functions	List of function names to enter a critical region.	[]
enter_critical_macros	List of macro names to enter a critical region (macros must expand to asm() statement).	[]
excluded_routines	List of functions that should be excluded from check.	[]
excluded_subgraphs	List of entry functions to subgraphs that should be excluded as subgraph from check.	[]
exit_critical_functions	List of function names to exit a critical region.	[]
exit_critical_macros	List of macro names to exit a critical region (macros must expand to asm() statement).	[]
inspect_pointers	Whether pointer targets should be inspected to detect more global variable uses.	False
nested_critical_regions	If set to True, critical regions nest; if set to False, a single exit-critical-region terminates all open critical regions.	True
output_safe_accesses	When enabled, outputs not only unsafe variable accesses, but also the safe ones.	False
partitions	Dict with partition name as key and dict as value which may contain keys 'entries' and/or 'vectors' with lists of entry points or vector table variables respectively. If special partition '*IRQ*' to configure interrupt handlers is missing, all functions not reached by any of the other options are treated as interrupt handlers.	dict(...)
report_cfg_based_critical_region_issues	Report unbalanced lock/unlock pairs within a routine. This has the same intention, but is slightly less strict than the purely syntactic check performed by the rule Parallelism-IncorrectCriticalSection.	False
show_identical_access	When enabled, outputs variable accesses of same kind (i.e., R/R and W/W).	False
show_object_number	Option for debugging (shows internal node numbers).	False
treat_types_as_atomic	List of type-patterns. A type-pattern is either a regular expression of a type name, or a triple of {min. alignment, max. size, type name-regex}. Each of the triple's components may be None. None is interpreted as general wildcard.	[]

Possible Messages

Name	Message
multiple_lock_add	Lock is acquired while it is already locked.
removed_nonexisting_lock	Lock is released, although it is not currently locked.
unbalanced_locks_path	Different control flow paths have different sets of locks.
unbalanced_locks_routine	Routine may return with different lock set than it is entered with {{in_set} vs {out_set}}.

Rules in Group Style.Indentation

Style.Indentation-AllmanBraces

Use Allman bracing style.

Input: IR

Source languages: C, C++, C++/CLI

Details

This rule enforces the usage of Allman bracing style. Opening braces should be on a separate line and at the same column as the parent statement/expression. Closing braces should be on the same column as opening brace. Children should be indented and must not further to the left than braces.

Configuration

Name	Explanation	Value
allow_single_line_empty_seq	Whether empty statement sequences on the same line are accepted: {} This is always assumed if allow_single_line_seq is True.	False
allow_single_line_seq	Whether all statement sequences on the same line are accepted: {...}	False
empty_macro_invocations	Macro invocations which expand to empty strings "" can be considered as taking up a location. This is intended to support constructs like QT's "emit signal_func()" where "emit" expands to "". In this case, the column of "emit" should start the line of source code. By default (value None): empty macro invocations will be supported as taking up a location. If configured as set of strings, only the macros whose name is in the set will be considered, other macros which expand to "" will be ignored. If the set is empty, macros which expand to "" will always be ignored.	None
expected_indent	Dictionary mapping PIR class names to expected number of indent columns for statements of this type. A value of None means that the node types are not checked at all, and a number string means that the node types have to be in exactly this column (not relative to surrounding sequence). Relative indents can be negative as well (clipped at 1). 'Statement' as key is used for the default value. If the value is a set/list/tuple of the above, then the element is accepted if any of the indents matches.	dict(...)

Possible Messages

Name	Message
all_children_misindented	Use Allman bracing style. All statements in this sequence should be corrected to start in column {}.
child_should_beIndented	Use Allman bracing style. Statement should be indented to start in column {}.
child_should_beOutdented	Use Allman bracing style. Statement should be outdented to start in column {}.
close_wrong_column	Use Allman bracing style. Closing brace not in same column as opening one (should be in column {}, but is in column {}).
open_wrong_column	Use Allman bracing style. Opening brace for body of {} in wrong column (should be in column {}, but is in column {}).
open_wrong_line	Use Allman bracing style. Opening brace for body of {} in wrong line (should be in line {}).

Style.Indentation-WhitesmithBraces

Use Whitesmith bracing style.

Input: IR

Source languages: C, C++, C++/CLI

Details

This rule enforces the usage of Whitesmith bracing style. Opening braces should be on a separate line with a specific column offset to the parent statement/expression. Closing braces should be on the same column as opening brace. Children should be on the same column as the braces.

Configuration

Name	Explanation	Value
allow_single_line_empty_seq	Whether empty statement sequences on the same line are accepted: {} This is always assumed if allow_single_line_seq is True.	False
allow_single_line_seq	Whether all statement sequences on the same line are accepted: {...}	False
control_structure_indentation_mapping	Dictionary mapping PIR class names to expected number of column offset for their braces. This varies between control structures. Statements inside this structure must start at the same column as the braces.	dict(...)
expected_indent	Dictionary mapping PIR class names to expected number of indent columns for statements of this type. A value of None means that the node types are not checked at all, and a number string means that the node types have to be in exactly this column (not relative to surrounding sequence). Relative indents can be negative as well (clipped at 1). 'Statement' as key is used for the default value. If the value is a set/list/tuple of the above, then the element is accepted if any of the indents matches.	dict(...)

Possible Messages

Name	Message
all_children_misindented	Use Whitesmith bracing style. All statements in this sequence should be corrected to start in column {}.
child_should_beIndented	Use Whitesmith bracing style. Statement should be indented to start in column {}.
child_should_beOutdented	Use Whitesmith bracing style. Statement should be outdented to start in column {}.
close_wrong_column	Use Whitesmith bracing style. Closing brace not in same column as opening one (should be in column {}, but is in column {}).
open_wrong_column	Use Whitesmith bracing style. Opening brace for body of {} in wrong column (should be in column {}, but is in column {}).
open_wrong_line	Use Whitesmith bracing style. Opening brace for body of {} in wrong line (should be in line {}).

Rules in Group Style.Metrics

Style.Metrics-MaxComplexity

Functions must not exceed cyclomatic complexity limit.

Input: IR

Source languages: C, C++, C++/CLI, C#

Details

Complex control flow can make code hard to understand. Try splitting up the code into multiple functions that can be understood individually.

Note

Cyclomatic complexity is also available as a metric, which can be configured to produce a metric violation when limits are exceeded. This stylecheck rule is available as an alternative to the metric.

Configuration

Name	Explanation	Value
maxcomplexity	Maximum acceptable cyclomatic complexity.	15
show_value	Whether metric value should be displayed.	False

Possible Messages

Name	Message
cyclomatic_complexity_with_value	Cyclomatic complexity of {} exceeds limit of {}.
excessive_cyclomatic_complexity	Cyclomatic complexity exceeds limit of {}.

Style.Metrics-MaxConditions

An expression must not have more than N logical operators.

Input: IR

Source languages: C, C++, C++/CLI, C#

Details

This rule warns when more than the configured number of logical operators (`&&` or `||`) are used within a single expression.

As an exception, this rule does not warn about complex conditions within custom operator == or operator != definitions. Those operators are often implemented by compare all the class members in a long chain of logical operators.

Rationale

Complex conditions can be hard to understand, and should be split into several statements or extracted into a function.

Configuration

Name	Explanation	Value
ignore_in	Names of routines to ignore for this check.	{'operator==', 'operator!=')}
maxconditions	Maximum acceptable number of conditions in an expression.	7

Possible Messages

Name	Message
excessive_condition_complexity	An expression must not have more than {} logical operators.

Style.Metrics-MaxNesting

A function must have a nesting level less than or equal N.

Input: IR

Source languages: C, C++, C++/CLI, C#

Details

Avoid exceeding the configured maximum nesting level. Use guard clauses to reduce nesting, or extract code into separate functions.

Configuration

Name	Explanation	Value
count_elseif	Whether nesting should be increased for else if	False
maxnesting	Maximum acceptable nesting level. Statements on top-level have a nesting level of 1.	10

Possible Messages

Name	Message
excessive_nesting	Nesting level {} greater than {}.

Style.Metrics-MaxOneStmtPerLine

Do not put more than one statement in a line.

Input: IR

Source languages: C, C++, C++/CLI, C#

Details

To keep the code readable, put each statement on its own line.

Example

```
// BAD:  
a = 1; b = 2;  
  
// GOOD:  
a = 1;  
b = 2;
```

Configuration

Name	Explanation	Value
ignore_stmts	Statements to be ignored when counting statements.	['Statement_Sequence']

Possible Messages

Name	Message
multiple_statements_per_line	Multiple statements per line.

Style.Metrics-MaxParams

A function must not have more than N parameters.

Input: IR

Source languages: C, C++, C++/CLI, C#

Configuration

Name	Explanation	Value
ignore_inherited	Do not report functions inheriting a unacceptable number of parameters.	False
maxparams	Maximum acceptable number of parameters.	7

Possible Messages

Name	Message
excessive_parameter_number	Function with {} parameters more than {}.

Style.Metrics-MaximumLineLength

The length of a line of code must not exceed the character limit.

Input: IR

Source languages: C, C++, C++/CLI, C#

Details

The length of a line of code must not exceed the character limit.

Configuration

Name	Explanation	Value
maximum_length	The maximum number of characters in a line.	160
show_value	Whether metric value should be displayed.	False

Possible Messages

Name	Message
excessive_line_length	Limit the length of a line of code to {} characters.
excessive_line_length_with_value	Length of a line with {} characters exceeds limit of {} characters.

Style.Metrics-TooManyIncludes

Report units with high number of included files

Input: IR

Source languages: C, C++

Details

This rule will report compilation units that include more than the configured number of files.

Configuration

Name	Explanation	Value
allowed	Maximum permissible number of included files	400
count_indirect_system_includes	Whether system-includes from inside system headers should be counted.	True
show_value	Select whether message should indicate the current value and limit.	False

Possible Messages

Name	Message
too_many_includes_in_unit	unit includes more than {} files.
too_many_includes_with_value	unit includes {} files which is more than the limit of {} files.

Rules in Group Style.Naming

Style.Naming-CSharpNamingConvention

Naming conventions for C#, inspired partly by <http://msdn.microsoft.com/de-de/library/vstudio/ms229043%28v=vs.100%29.aspx>

Input: IR

Source languages: C#

Details

This rule checks naming conventions for C#, inspired partly by the guidelines specified [here](#).

Configuration

Name	Explanation	Value
naming	Dictionary to map entity -> naming convention.	dict(...)

Style.Naming-CapitalizeFunctions

Start functions and methods with an upper-case letter.

Input: IR

Source languages: C, C++, C++/CLI

Details

Functions and methods must start with an upper-case letter.

See Also

[Rule NamingConvention](#) supercedes this rule, as it provides more flexible support for naming conventions.

Configuration

Name	Explanation	Value
allow_underscores	Allow underscores in identifier.	True

Possible Messages

Name	Message
funcname_starts_lowercase	Function name does not start with upper-case letter.
funcname_starts_lowercase_or_has_underscore	Function name does not start with upper-case letter or contains underscores.

Style.Naming-FileExtensionNaming

Use only certain extensions for file names.

Input: IR

Source languages: C++, C++/CLI

Configuration

Name	Explanation	Value
allowed_extensions	Which file extensions are allowed.	set(['.h', '..hxx', '.cpp', '.hpp'])
file_type	The files to check the extensions of.	{'Primary_File', 'User_Include_File'}

Possible Messages

Name	Message
illegal_file_extension	Use only certain extensions for file names.

Style.Naming-FilenameNaming

Use only certain characters for file names.

Input: IR

Source languages: C, C++, C++/CLI

Details

This rule checks the use of filename names.

Configuration

Name	Explanation	Value
allowed_charset	Regular expression of valid characters, ".", "/", and "" are added by the rule.	A-Za-z0-9!"#%&\'\\()*\\+\\,-\\.:/;=>\\?\\[]\\^_\\{\\}\\~-\\t\\f\\v\\n\\r\\0\\a\\b
show_charset	Wether charset should appear in message.	False

Possible Messages

Name	Message
illegal_character	Use only certain characters for file names.
illegal_character_with_pattern	Use only characters in "[{}]*" for file names.

Style.Naming-NamingConvention

Check named entities against naming conventions.

Input: IR

Source languages: C, C++, C++/CLI

Details

This is a highly flexible rule that checks names against a configurable list of conditions.

In the current configuration, this rule will check:

- Const data member must be all upper-case
- Const global variable must be all upper-case
- Data member must be all lower-case
- Data member of a class must start with "m_"
- Enumeration type must use camel-case, starting with upper-case letter
- Enumerator must be all upper-case
- Enumerators must start with common prefix
- Function must be all lower-case
- Global variable must be all lowercase and start with g_
- Label must be all lower-case
- Local variable must be all lower-case
- Macro must be all upper-case
- Namespace must be all lower-case
- Parameter must be all lower-case
- Private or protected function member must be prefixed by underscore, use camel-case, starting with lower-case character
- Public function member must use camel-case, starting with lower-case character
- Static data member must be all lower-case
- Static function member must be all lower-case
- Typedef type must use camel-case, starting with upper-case letter

- User-defined type must use camel-case, starting with upper-case letter

Configuration

Name	Explanation	Value
excluded_global_functions	Names of global functions to which the check should not be applied.	('main', 'WinMain', 'wmain', 'wWinMain', 'DllMain')
naming	Each partial naming rule for the identifier of a certain node type has the following format: node_type_name : [(checker, message), (checker, message), ...] "checker" can be either a regular expression (in which case it must match the identifier) or a complex checker consisting of a guarding predicate (guard) and an regular expression (in which case the guard governs if the regexp must match). If a checker fails, its message is issued. All checkers in the list must match in order to get no message, i.e., if you want to state different regexps for different combinations of aspects (visibility etc), make sure the guards create the needed disjointness of your regexp checks. If multiple checkers fail for the same identifier, you will end up with multiple messages for that identifier. Some aspects are expressed by attributes (constness, visibility), some by the hierarchy (staticness).	dict(...)

Style.Naming-NoDoubleUnderscoreInMacro

Do not use more than one consecutive underscore in a macro.

Input: IR

Source languages: C, C++

Details

This rule prevents the use of multiple consecutive underscores within macro names.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
double_underscore_in_macro	Macro definition with consecutive underscores.
macro_starting_with_double_underscore	Macro name starting with consecutive underscores.

Style.Naming-NoSingleCharIdentifier

Do not use identifiers consisting of just a single character.

Input: IR

Source languages: C, C++, C++/CLI

Configuration

Name	Explanation	Value
exclude_catch	If True, identifiers to name the exception in catch handlers are tolerated.	False
exclude_local_nonstatic_variables	If True, local non-static variables are tolerated.	False
exclude_local_static_variables	If True, local static variables are tolerated.	False
exclude_loop_counter	If True, for-loop counters are tolerated.	False
whitelist	Allowed single-character names.	[]

Possible Messages

Name	Message
single_char_identifier	Identifier is single-character identifier.

Style.Naming-SourceFileName

Name a source_file for a type defined in it.

Input: IR

Source languages: C, C++, C++/CLI

Configuration

Name	Explanation	Value
file_type	Type of source_file file to examine.	File
junk_characters	Characters that should not be considered in the filename to type similarity.	_-::
named_node_types	Physical IR node types to inspect for logical names to compare against the file name	Named_Type_Interface
similarity_threshold	Ratio of how similar the names need to be.	0.9

Possible Messages

Name	Message
source_file_name_not_type	The file should be named as a type it declares.

Rules in Group Style.Parens

Style.Parens-MissingLogicalOperandParens

The operands of a logical && or || shall be primary-expressions.

Input: IR

Source languages: C, C++, C++/CLI

Configuration

Name	Explanation	Value
allowed	Can be used to name additional allowed IR node types for the operands.	None
require_postfix_eprission	Whether postfix or primary expressions are required as operands.	False

Possible Messages

Name	Message
missing_parens_for_logical_operator_operand	Operand of && or must be parenthesized.

Style.Parens-MissingParens

Limited dependence should be placed on C's operator precedence rules in expressions.

Input: IR

Source languages: C, C++, C++/CLI

Configuration

Name	Explanation	Value
allow_algebraic_order	Whether parens can be omitted for multiplicative operators as operands in additive operators	False

Possible Messages

Name	Message
missing_parens_depends_on_precedence	Parentheses should be used to avoid dependence on precedence rules

Style.Parens-MissingParensForPrecedenceLevel

The precedence of operators within expressions should be made explicit.

Input: IR

Source languages: C, C++, C++/CLI

Configuration

Name	Explanation	Value
exception_levels	List of operator precedence levels for which no parentheses are required.	[]

Possible Messages

Name	Message
missing_parens_depends_on_precedence_level	Parentheses should be used to avoid dependence on precedence rules

Style.Parens-ParensDuplicatingAlgebraicOrder

Mathematical expressions that follow algebraic order do not require parentheses.

Input: IR

Source languages: C, C++, C++/CLI

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
parens_duplicating_algebraic_order	Mathematical expressions that follow algebraic order do not require parentheses

Style.Parens-SizeofMissingParens

The precedence of operators within expressions should be made explicit.

Input: IR

Source languages: C, C++, C++/CLI

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
sizeof_missing_parens	Operand of sizeof operator should be enclosed in parentheses

Style.Parens-SuperfluousRHSParens

Limited dependence should be placed on C's operator precedence rules in expressions.

Input: IR

Source languages: C, C++, C++/CLI

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
assignment_rhs_in_parens	No parentheses required for right-hand side of assignment

Style.Parens-SuperfluousUnaryOperatorParens

Limited dependence should be placed on C's operator precedence rules in expressions.

Input: IR

Source languages: C, C++, C++/CLI

Configuration

Name	Explanation	Value
allow_redundant_parentheses_in_logical_operator	Allow redundant parentheses around unary operators when they appear as operand of '&&' or ' '. This option can be used to avoid a conflict with MisraC++-5.2.1, which requires parentheses in these instances.	False

Possible Messages

Name	Message
unary_op_in_parens	No parentheses required for unary operator

Rules in Group Style.Whitespace

Style.Whitespace-LineBreaks

You have to use the specified sort of line breaks.

Input: IR

Source languages: C, C++, C++/CLI

Configuration

Name	Explanation	Value
allowed_line_break_form	Selects the form of line break that is allowed ('\n', '\r' or '\r\n').	
excluded_files	Globbing patterns for files to exclude from this check in addition to -exclude.	[]
file_types	Selects file types to be checked: Primary_File, User_Include_File, System_Include_File.	('Primary_File', 'User_Include_File')

Possible Messages

Name	Message
bad_line_break	Line break is not of the allowed form {!r}
bad_line_break_block	Line breaks from here on up to line {} (inclusive) are not of the allowed form {!r}
file_using_bad_line_breaks	All line breaks in this file are not of the allowed form {!r}

Style.Whitespace-NoBlankLinesAtBraces

Avoid blank lines after { and before }.

Input: IR

Source languages: C, C++, C++/CLI

Details

There should be no blank line after an opening brace or before a closing brace.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
blank_after_brace	Avoid blank lines after {.
blank_before_brace	Avoid blank lines before }.

Style.Whitespace-NoTabs

Do not use tabs.

Input: IR

Source languages: C, C++, C++/CLI

Configuration

Name	Explanation	Value
per_file	Flag which combines all violations in a file to one violation where individual findings are handled as additional entities.	False

Possible Messages

Name	Message
tab_character	Do not use tabs.

Style.Whitespace-NoTrailingWhitespace

Do not use trailing whitespace.

Input: IR

Source languages: C, C++, C++/CLI

Configuration

Name	Explanation	Value
allow_in_comment	Whether trailing whitespace in comments should be allowed.	False

Possible Messages

Name	Message
trailing_whitespace	Do not use trailing whitespace.

Style.Whitespace-NoWhitespaceMemberSelection

No whitespace before or after a member selection.

Input: IR

Source languages: C, C++, C++/CLI

Details

The should be no whitespace around the following operators: ., ->, .* and ->*

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
whitespace_member_selection	Whitespace around member selection.

Style.Whitespace-NoWhitespacePointerReference

No whitespace rule in declarations between type and * or &.

Input: IR

Source languages: C, C++, C++/CLI

Details

There should never be a whitespace between the type and * or & in declarations.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
whitespace_between	Avoid whitespace between type and * or &.

Style.WhiteSpace-NoWhitespaceUnaryOperator

No whitespace after an unary operator.

Input: IR

Source languages: C, C++, C++/CLI

Details

There should be no whitespace after the unary operators ~, !, + and -.

Configuration

Name	Explanation	Value

Possible Messages

Name	Message
whitespace_after_unary	Whitespace after unary operator.

Style.WhiteSpace-WhitespaceNextToOperator

There must be exactly one whitespace on both sides of an operator.

Input: IR

Source languages: C, C++, C++/CLI

Details

There must be exactly one whitespace on both sides of an operator: Instead of `a=b+c`, write `a = b + c`.

Configuration

Name	Explanation	Value
at_least_one_whitespace_for_assignments	Flag which weakens the rules default to expect just one space next to assignment tokens. Multiple spaces will be allowed.	False
at_least_one_whitespace_for_binary_operators	Flag which weakens the rules default to expect just one space next to binary operator tokens. Multiple spaces will be allowed.	False
check_assignment	Flag if assignment tokens should be checked for missing spaces.	True
check_binary_operator	Flag if binary operator tokens should be checked for missing spaces.	True

Possible Messages

Name	Message
whitespaceViolation	No or incorrect whitespaces next to operator.