

Laslo Kraus

PROGRAMSKI JEZIK

C++

sa rešenim zadacima

AKADEMSKA MISAO
Beograd, 2007

Laslo Kraus

PROGRAMSKI JEZIK C++
SA REŠENIM ZADACIMA
Sedmo izdanje

Recenzenti

Dr Igor Tartalja
Dr Jelica Protić

Izdavač

AKADEMSKA MISAO
Bul. kralja Aleksandra 73, Beograd

Štampa

Planeta print, Beograd

Tiraž

500 primeraka

ISBN 978-86-7466-288-5

NAPOMENA: Fotokopiranje ili umnožavanje na bilo koji način ili ponovno objavljivanje ove knjige - u celini ili u delovima - nije dozvoljeno bez prethodne izričite saglasnosti i pismenog odobrenja izdavača.

Dušanki i Katarini

Predgovor

Ova knjiga predstavlja udžbenik za programski jezik *C++* za široki krug čitalaca. Knjigu mogu da koriste i ljudi s relativno malo programerskog iskustva, ali je vrlo korisna i za profesionalne programere kojima jezik *C++* nije osnovni programski jezik u profesionalnoj delatnosti.

Jezik *C++* je izrastao iz jezika *C*. Ova knjiga je nastavak knjige *Programski jezik C sa rešenim zadacima*. Podrazumeva se da čitalac zna osnovne elemente jezika *C* i da ima izvjesno iskustvo u programiranju na jeziku *C*.

Programski jezik *C++* je izložen u potpunosti, neki delovi s više neki s manje detalja. Od sadržaja prateće standardne biblioteke klasa i funkcija prikazani su samo važniji delovi.

Jedini način da se nauči neki programski jezik je da se pišu programi na njemu. Ova knjiga je u potpunosti podređena tom osnovnom načelu.

Pre svega, u uvodnom poglavlju 1 ukratko su izložene komande za obradu programa na jeziku *C++* na računarima pod operativnim sistemima *UNIX* i *MS-DOS*. Pored toga, dat je kratak pregled jezika *C* kao podsetnik na ono što bi trebalo da je poznato od ranije.

U poglavlju 2 prikazana su manja proširenja jezika *C* koja čine da jezik postane bolji jezik *C*. Deo tih izmena je uveden radi otklanjanja nekih nekonzistentnosti jezika *C*, drugi deo uvodi i nove mogućnosti. U novine spadaju upućivači koji omogućavaju prenos podataka u potprograme pomoću adrese na način koji se razlikuje od korišćenja pokazivača, uvođenje operatora za dinamičko dodeljivanje memorije i operatora za ulaz i izlaz podataka uz primenu konverzija, mogućnost umetanja manjih funkcija u kôd, kao i mogućnost definisanja više funkcija s istim imenima pod uslovom da imaju parametre različitih tipova. Za razvoj velikih programske sistema uvedena je i mogućnost razbijanja celokupnog programa na više prostora imena koji čine odvojene dosege identifikatora.

Ono što posebno obeležava jezik *C++* je uvođenje klasa sa svim pratećim konceptima. Klase su tipovi podataka u pravom smislu reči, jer pored definisanja mogućih vrednosti podataka klasnih tipova, mogu da definišu i operacije koje i samo koje mogu da se primenjuju na te podatke. U poglavlju 3 izloženi su osnovni pojmovi vezani za klase.

U jeziku *C++* posebna pažnja je posvećena stvaranju i uništavanju objekata klasnih tipova. Klase mogu da se definišu tako da nijedan novi objekat ne može da ostane neinicijalizovan i da svaki objekat bude propisno uništen kada više nije potreban. Kako se to postiže objašnjeno je u poglavlju 4.

Jezik C++ omogućava da se standardnim operatorima dodeljuju posebna tumačenja za slučajeve kada su operandi objekti klasnih tipova. To se postiže definisanjem odgovarajućih operatorskih funkcija, kojima je posvećeno poglavljje 5.

Grupe objekata u svakodnevnom životu mogu da se dele na podgrupe. Na primer, vozači mogu da se podele na automobile, avione i brodove. Ako zajedničke osobine grupe objekata mogu da se opišu nekom klasom, klase za opis osobina podgrupa mogu da se obrazuju izvođenjem iz klase zajedničkih osobina. Izvedene klase obrađene su u poglavljiju 6.

Jezik C++ omogućava da rukovanje izuzecima (greškama) vremenski ne opterećuje program sve dok se izuzetak ne pojavi. Kada se izuzetak pojavi, kontrola se automatski predaje jednom od rukovalaca izuzecima koje je definisao programer. Način rukovanja izuzecima prikazan je u poglavljiju 7.

Generičke funkcije i klase omogućavaju automatsko generisanje funkcija ili klasa kada istu obradu treba primeniti na podatke različitih tipova. Na primer, algoritam za uređivanje niza objekata po nekom kriterijumu ne zavisi od tipa objekata koji se uređuju. Opisivanje šablonu generičkih funkcija i klasa je predmet poglavljja 8.

Kao dopuna jeziku C++ postoji velika biblioteka standardnih klasa i funkcija za rad sa zbirkama podataka (nizovi, liste, skupovi, ...), tekstovima, kompleksnim brojevima itd. U poglavljiju 9 je prikazan deo te biblioteke.

Ulaz i izlaz podataka nije deo jezika C++ već se realizuje odgovarajućim pratećim bibliičkim klasama. U poglavljju 10 prikazan je deo tih klasa koji je dovoljan i za relativno složene ulazne i izlazne operacije, uključujući i rad s datotekama različitih struktura.

Jezik C++ je vrlo složen. Nisu svi detalji neophodni svakome, a naročito ne početnicima. Odjeljci u knjizi koji mogu da se preskoče u prvom čitanju, bilo zbog složenosti, bilo zbog manjeg značaja, obeleženi su sa Δ .

Rukovodeći se gore istaknutim načelom o učenju jezika, ova knjiga, pored manjih primera u toku izlaganja, na kraju svakog poglavlja sadrži nekoliko rešena zadatka. Više zadataka mogu da se nađu u autorovoj zbirici *Rešeni zadaci iz programskog jezika C++* (videti [7]). Kroz zadatke u ovoj knjizi i u zbirci skreće se pažnja na neke specifičnosti jezika C++ i daju ideje za elegantno i efikasno rešavanje nekih problema.

Ova knjiga je više nego udžbenik za programski jezik C++. Kroz rešene zadatke prikazane su i primene osnovnih elemenata objektno orijentisanog programiranja, a na primery koji se uopšteno sreću u računarskoj tehnici: obradi redova, stekova, listi, tekstova, datoteka itd. Posebna pažnja posvećena je i inženjerskim aspektima programiranja. Nije dovoljno da program rešava zadati problem, već treba da bude i „lep”, razumljiv, da troši što manje memorije i da bude što efikasniji.

Svi programi koji se nalaze u ovoj knjizi potpuni su u smislu da mogu da se izvršavaju na računaru. Provereni su na nekoliko računara korišćenjem nekoliko različitih prevodilaca za jezik C++.

Izvorni tekstovi svih programa iz ove knjige mogu da se preuzmu sa Interneta, sa adrese galeb.etf.bg.ac.yu/~kraus/knjige/. Na istoj adresi će se objaviti ispravke eventualnih, naknadno otkrivenih grešaka u knjizi. Svoja zapažanja čitaoci mogu da upute autoru elektronskom poštom na adresu kraus@etf.bg.ac.yu.

Beograd, april 2007

Laslo Kraus

Sadržaj

Predgovor	v
Sadržaj	vii
1 Uvod	1
1.1 O programskom jeziku C++.....	1
1.2 Obrada programa na jeziku C++	2
1.2.1 Rad pod operativnim sistemom <i>UNIX</i>	2
1.2.2 Rad pod operativnim sistemom <i>MS-DOS</i> i prevodiocem <i>Borland C++ 4</i>	
1.3 Programski jezik C	6
1.3.1 Elementi jezika	6
1.3.2 Tipovi podataka	6
1.3.3 Operatori i izrazi	9
1.3.4 Naredbe	12
1.3.5 Funkcije	14
1.3.6 Struktura programa	15
1.3.7 Preprocessor	17
1.3.8 Bibliotečke funkcije	18
2 Proširenja jezika C	19
2.1 Elementi jezika	19
2.1.1 Komentari	19
2.1.2 Službene reči	19
2.2 Tipovi podataka	20
2.2.1 Logički podaci	20
2.2.2 Znakovne konstante	20
2.2.3 Simboličke konstante	20
2.2.4 Definisanje podataka	20
2.2.5 Definisanje tipova	22
2.2.6 Nabranjanja	22
2.2.7 Uvek promenljiva polja u strukturama	23

2.2.8 Bezimene unije Δ	23
2.2.9 Pokazivači.....	24
2.2.10 Upućivači.....	24
2.2.10.1 Definisanje upućivača	24
2.2.10.2 Upućivači i funkcije	26
2.3 Operatori	27
2.3.1 Vrednost izraza kao <i>vrednost</i>	28
2.3.2 Konverzija tipa	28
2.3.3 Ulaz i izlaz podataka.....	30
2.3.3.1 Operatori za čitanje i pisanje s konverzijom.....	31
2.3.3.2 Manipulatori za podešavanje parametara konverzije	32
2.3.3.3 Čitanje i pisanje znakova bez konverzije	33
2.3.4 Dinamička dodela memorije.....	34
2.4 Funkcije	36
2.4.1 Ugrađene funkcije.....	36
2.4.2 Podrazumevane vrednosti parametara funkcija	37
2.4.3 Preklapanje imena funkcija	38
2.4.4 Dohvatanje delova složenih podataka	39
2.4.5 Povezivanje s drugim jezicima Δ	40
2.4.6 Glavna funkcija	41
2.5 Prostori imena	41
2.5.1 Definisanje prostora imena	41
2.5.2 Upotreba identifikatora u prostoru imena.....	42
2.5.3 Bezimeni prostor imena.....	44
2.5.4 Uklapanje prostora imena	45
2.5.5 Prostor imena std.....	46
2.6 Zadaci	48
2.6.1 Metoda podele za uređivanje	48
3 Klase	53
3.1 Definisanje i deklarisanje klasa	54
3.2 Objekti klasnih tipova	55
3.3 Metode klase	56
3.4 Konstruktori.....	58
3.4.1 Definisanje klasa s konstruktorima.....	59
3.4.2 Pozivanje konstruktora	59
3.4.3 Definisanje konstruktora	60
3.4.4 Konstruktori posebne namene	61
3.4.4.1 Podrazumevani konstruktor	61
3.4.4.2 Konstruktor kopije	62
3.4.4.3 Konstruktor konverzije	63
3.4.5 Tok izvršavanja konstruktora	64
3.4.6 Inicijalizacija nizova objekata	65
3.4.7 Konstruktori za proste tipove podataka	65
3.5 Destruktori	66
3.6 Konstante klasnih tipova	69

Mario Jancic ix

3.7 Zajednički članovi klase	69
3.8 Prijateljske funkcije klase	72
3.9 Pokazivači na polja klase Δ	74
3.10 Unutrašnje klase.....	75
3.11 Lokalne klase	76
3.12 Strukture i unije	76
3.13 Dijagrami klasa	78
3.14 Zadaci 81	
3.14.1 Obrada tačaka u ravni	81
3.14.2 Obrada uređenih skupova	84
3.14.3 Operacije nad jedinstvenim stekom	90
3.14.4 Obrada krugova u ravni	93
4 Preklapanje operatora	101
4.1 Operatorske funkcije.....	101
4.2 Preklapanje operatora $++$ i $--$	104
4.3 Operatori za ulaz i izlaz podataka ($>>$, $<<$)	105
4.4 Preklapanje operatora (<i>tip</i>)	106
4.4.1 Automatska konverzija tipa	107
4.5 Preklapanje operatora =	109
4.5.1 Inicijalizacija i dodela vrednosti	110
4.6 Preklapanje operatora []	112
4.7 Preklapanje operatora ()	114
4.8 Preklapanje operatora $->$ Δ	116
4.9 Preklapanje operatora new i delete Δ	117
4.10 Nabranjanja i preklapanje operatora Δ	118
4.11 Zadaci 120	
4.11.1 Obrada vremenskih intervala	120
4.11.2 Obrada polinoma	125
4.11.3 Obrada redova	133
5 Izvedene klase	141
5.1 Definisanje izvedenih klasa	142
5.2 Dijagrami klasa za izvedene klase	144
5.3 Upotreba članova izvedenih klasa	145
5.4 Virtuelne osnovne klase Δ	147
5.5 Stvaranje i uništavanje primeraka izvedenih klasa	149
5.6 Konverzija tipa između osnovnih i izvedenih klasa	150
5.7 Virtuelne metode	152
5.8 Apstraktne klase	156
5.9 Dinamička konverzija tipa podataka* Δ	157
5.10 Dinamičko određivanje tipa podataka Δ	158
5.11 Zadaci 160	
5.11.1 Obrada geometrijskih figura u ravni	160
5.11.2 Obrada zbirki geometrijskih figura u ravni Δ	170

6 Izuzeci.	193
6.1 Rukovanje izuzecima	193
6.2 Prijavljivanje izuzetaka	194
6.3 Prihvatanje izuzetaka	195
6.4 Neprihvaćeni i neočekivani izuzeci Δ	197
6.5 Standardni izuzeci Δ	198
6.6 Zadaci	199
6.6.1 Obrada vektora zadatih opsega indeksa	199
6.6.2 Izračunavanje određenog integrala	204
7 Generičke funkcije i klase	217
7.1 Definisanje šablonu	217
7.2 Generisanje funkcija i klasa	219
7.3 Podrazumevane vrednosti parametara šablonu Δ	221
7.4 Specijalizacija Δ	222
7.4.1 Specijalizacija generičkih klasa Δ	222
7.4.2 Specijalizacija generičkih funkcija Δ	223
7.5 Generičke metode i unutrašnje generičke klase Δ	224
7.6 Zadaci	226
7.6.1 Fuzija nizova objekata	226
7.6.2 Obrada stekova objekata	229
7.6.3 Uređivanje nizova podataka Δ	232
8 Standardna biblioteka Δ	237
8.1 Usluge Δ	237
8.1.1 Relacioni operatori Δ	237
8.1.2 Parovi podataka Δ	238
8.1.3 Funkcijske klase i objekti Δ	239
8.1.4 Kompleksni brojevi (complex) Δ	241
8.2 Zbirke podataka Δ	241
8.2.1 Iteratori Δ	242
8.2.2 Vektori (vector) Δ	245
8.2.3 Redovi s dva kraja (deque) Δ	249
8.2.4 Liste (list) Δ	249
8.2.5 Specijalne zbirke (queue, priority_queue i stack) Δ	250
8.2.6 Preslikavanja (map i multimap) Δ	252
8.2.7 Skupovi (set i multiset) Δ	254
8.2.8 Tekstovi (string) Δ	254
8.2.9 Nizovi bitova (vector<bool> i bitset) Δ	258
8.3 Algoritmi Δ	260
8.3.1 Algoritmi nad pojedinačnim podacima Δ	260
8.3.2 Obrada pojedinačnih elemenata zbirki Δ	260
8.3.3 Pretraživanje zbirki Δ	263
8.3.4 Obrada zbirki Δ	265
8.3.5 Obrada uređenih sekvenčijalnih zbirki Δ	267

8.4 Zadaci Δ	269
8.4.1 Obrada obojenih geometrijskih figura u ravni Δ	269
9 Ulaz i izlaz Δ	285
9.1 Tokovi za datoteke Δ	286
9.1.1 Stvaranje tokova za datoteke Δ	286
9.1.2 Otvaranje i zatvaranje datoteka Δ	287
9.2 Tokovi u memoriji Δ	288
9.2.1 Stvaranje tokova u memoriji Δ	288
9.2.2 Pristup sadržaju tokova u memoriji Δ	288
9.3 Rad s tekstualnim tokovima Δ	289
9.3.1 Prenos bez konverzije Δ	289
9.3.2 Prenos s konverzijom Δ	290
9.4 Rad s binarnim tokovima Δ	292
9.5 Pozicioniranje unutar toka (direktni pristup) Δ	293
9.6 Signalizacija grešaka Δ	293
9.7 Klase za ulaz i izlaz Δ	294
9.8 Zadaci Δ	296
9.8.1 Ostvarenje relativnih datoteka Δ	296
Literatura.....	305
Indeks.....	307

1 Uvod

1.

1.1 O programskom jeziku C++

Programski jezik *C* je jezik opšte namene, srednjeg nivoa, koji omogućava dosta intiman kontakt s hardverom računara. Poseduje strukturirane tipove podataka i upravljačke strukture što je karakteristika viših programskega jezika. S druge strane podržava manipulaciju bitovima, korišćenje procesorskih registara, pristup podacima pomoću adrese i operatore orijentisane ka hardveru računara. Ovo su karakteristike nižih programskega jezika, kao što su simbolički mašinski jezici. Sintaksa jezika omogućava koncizno izražavanje i pisanje strukturiranih programa.

Programski jezik *C* projektovao je *Dennis Ritchie* 1972. godine u *Bell*-ovim laboratorijama. Osnovni cilj je bio sastavljanje jezika nezavisnog od računara, sa karakteristikama viših programskega jezika, koji će moći da zameni simboličke mašinske (*assembler-ske*) jezike koji su, i te kako, zavisni od računara.

Dugi niz godina osnovna definicija jezika *C* bio je referentni priručnik (*Reference Manual*) u sastavu prvog izdanja knjige *The C Programming Language* čiji su autori *Brian W. Kernighan* i *Dennis M. Ritchie*. Varijanta jezika *C* koja je opisana u toj knjizi danas se naziva *Klasičan C*.

Zvanični standard za jezik *C*, takozvani *ANSI C*, izdao je Američki nacionalni institut za standarde (*American National Standards Institute*) 1989. godine pod brojem X3.159-1989.

Pojavom novih tehnika u programiranju, prvenstveno pojegovom objektno orijentisanog programiranja, osetila se potreba za novim mogućnostima jezika *C*. Tako su 1980. godine dodate klase, provera i konverzije tipova argumenata prilikom pozivanja funkcija i još neke druge novine. Tako dobijeni jezik nazivao se *C sa klasama*.

Glavnu novinu predstavlja mogućnost definisanja novih tipova podataka u pravom smislu te reči. Dok strukture (*struct*) u jeziku *C* definišu samo moguće vrednosti za definisane podatke, novouvedene klase (*class*) definišu i moguće operacije nad tim podacima. Štaviše, jedine operacije nad podacima date klase mogu da budu samo one koje su predviđene definicijom te klase.

Posle daljeg proširivanja, na prvom mestu dodavanjem virtuelnih funkcija i preklapanja operatora, jezik je 1983/84. godine konačno dobio današnje ime *C++*. Ime treba da sug-

riše da se ne radi o novom jeziku, već o proširivanju jezika *C* (*++* je u jeziku *C* operator potvećavanja!). Preko 95% jezika *C* usvojeno je bez izmena i u jeziku *C++*. Promjenjeni su samo detalji koji su morali da budu promenjeni radi obezbeđivanja konzistentnosti novih koncepcija u jeziku *C++*. Te izmene se odnose na vrlo suptilne detalje koji dolaze do izražaja samo kod krajnje profesionalnog programiranja.

U nedostatku zvaničnog standarda, kao osnovna definicija jezika u početku se koristila knjiga *The C++ Programming Language*, čiji je autor *Bjarne Stroustrup*. On se smatra i autorom samog jezika *C++*.

Kasnije su dodate nove mogućnosti kao što su višestruko nasleđivanje, apstraktne klase, mehanizmi za sastavljanje generičkih klasa i za rukovanje izuzecima (obradu grešaka). Osnovnu definiciju jezika *C++* s početka 1991. godine predstavlja knjiga *The Annotated C++ Reference Manual* čiji su autori *Margaret A. Ellis* i *Bjarne Stroustrup*. To je istovremeno bio i jedan od osnovnih dokumenata grupe za izradu *ANSI* standarda za jezik *C++*.

ANSI standard za jezik *C++* usvojen je 14. 11. 1997. godine. U nastavku ove knjige standard za jezik *C++* naziva se skraćeno *Standard*.

Standardom je jezik *C++* znatno proširen. Dodata je mogućnost provere tipa objekata za vreme izvršavanja programa, detaljnije su razradene generičke funkcije i klase i definisana je bogata biblioteka gotovih klasa i funkcija za često korišćene obrade.

Jezik *C++* danas je jedan od najmoćnijih jezika za objektno orientisano programiranje.

Operativni sistem *UNIX* predstavlja prirodno okruženje za jezik *C++*, kao što je to slučaj i sa jezikom *C*. Ali, s obzirom da se radi o jeziku opšte namene, nudi se i pod drugim operativnim sistemima, kao što je *MS-DOS* i *MS-Windows* na danas vrlo popularnim ličnim računarima tipa *IBM-PC*.

1.2 Obrada programa na jeziku *C++*

Obrada programa sastoji se od sledeća četiri koraka:

- unošenje izvornog teksta programa u datoteku na disku,
- prevodenje izvornog teksta programa,
- povezivanje prevedenog oblika programa s potrebnim korisničkim i sistemskim potprogramima u izvodljivi oblik, i
- izvršavanje programa.

U narednim odeljcima prikazano je kako se prethodni koraci izvode u dva najčešća okruženja u kojima se koristi programski jezik *C++*. Prvo je operativni sistem *UNIX* koji predstavlja prirodno okruženje za jezik *C++*. Drugo često okruženje u kome se koristi jezik *C++* je operativni sistem *MS-DOS* na ličnim računarima tipa *IBM-PC* i njima sličnim računarima.

1.2.1 Rad pod operativnim sistemom *UNIX*

Za sastavljanje izvornog teksta programa može da se koristi bilo koji urednik teksta. Standardni urednik teksta koji je sigurno raspoloživ pod operativnim sistemom *UNIX* je ekranSKI urednik *vi*. Treba da se napomene da postoje i urednici teksta koji su znatno efikasniji i lakši za korišćenje od urednika *vi*, ali oni se ne nalaze na svakom računaru.

1.2.1 Rad pod operativnim sistemom *UNIX*

U slučaju rada na jeziku *C++* ime datoteke mora da se završi sufiksom *.C*. Komanda upravljačkog jezika operativnog sistema *UNIX* za unošenje i obradu izvornog teksta programa u datoteci *imeprog.C* je:

```
vi imeprog.C
```

Po završetku sastavljanja izvornog teksta programa, komandom:

```
cc imeprog.C
```

prevodi se izvorni tekst programa na jeziku *C++* iz datoteke *imeprog.C*. Rezultat prevodenja se smešta u datoteku *imeprog.o*, i odmah se stvara i povezani oblik programa koji se smešta u datoteku standardnog imena *a.out*. Ukoliko ovo standardno i bezlično ime ne odgovara, proizvoljno ime za datoteku sa povezanim oblikom programa može da se odabere opcijom *-o*. To proizvoljno ime obično se poklapa s imenom datoteke koja sadrži izvorni oblik programa, naravno bez sufiksa *.C*:

```
cc imeprog.C -o imeprog
```

U slučaju sastavljanja velikog programskega sistema vrlo često se izvorni tekst glavnog programa (glavne funkcije) i svih potrebnih potprograma (funkcija) smešta u više datoteka. U tom slučaju potrebno je da se preskoči povezivanje prilikom prevodenja sadržaja datoteka u kojima se nalaze samo funkcije. To se postiže dodavanjem opcije *-c* u komandi *CC*:

```
CC -c imeprog.C
```

U komandi *CC* može da bude naveden i čitav niz imena datoteka od kojih neke mogu da sadrže izvorne tekstove, a neke prevedene oblike programskih modula. Pomoću komande:

```
CC ime1.C ime2.o ... imeN.C -o paket
```

prevodiće se sadržaji datoteka čija se imena završavaju na *.C* i rezultati prevodenja smeštaju se u odgovarajuće datoteke sa sufiksom imena *.o*. Posle toga, povezuju se programski moduli iz svih novostvorenih datoteka kao i iz datoteka čija se imena u prethodnoj komandi završavaju sa *.o*. Povezani oblik celokupnog programskega paketa smešta se u datoteku imena *paket*.

Na kraju, izvršavanje programa postiže se komandom koja se sastoji samo od imena datoteke koja sadrži povezani oblik programa. To ime može da bude standardno ime *a.out* ili ime koje je odabrao programer:

```
a.out
```

```
imeprog
```

Posle jedne od ove dve komande počinje izvršavanje programa. U toku rada programa korisnik treba da preko tastature unosi podatke po redosledu kako ih očekuje program s „glavnog ulaza“ računara. Na ekranu će se pojaviti podaci koji se unose preko tastature kao i podaci koje program šalje na „glavni izlaz“ računara.

Pod operativnim sistemom *UNIX* postoji vrlo elegantan način za „skretanje“ glavnog ulaza i/ili glavnog izlaza. Time se postiže da se podaci čitaju iz neke datoteke umesto s tastature, odnosno da se upisuju u neku datoteku umesto ispisivanja na ekranu. To se postiže izvršavanjem programa pomoću komande oblika:

```
imeprog <podaci> >rezult
```

Znak *<* ispred imena označava da se čitanje s glavnog ulaza vrši iz te datoteke (strelica koja „izlazi“ iz imena datoteke), a znak *>* ispred imena da se pisanje na glavnom izlazu vrši u tu datoteku (strelica koja „ulazi“ u ime datoteke). Datoteke *podaci* i *rezult* su tekstualne

datoteke, što znači da mogu da se obrađuju urednikom teksta ili drugim programima za obradu teksta.

Treba da se napomene da operativni sistem *UNIX* vodi računa o velikim i malim slovima. Zbog toga, sve navedene komande moraju da se pišu tačno na gore navedeni način. Velika i mala slova su značajna i u imenima datoteka. Tako *alfa.C* i *ALFA.C* predstavljaju dve različite datoteke, a *alfa.c* se ne prihvata kao datoteka koja sadrži izvorni tekst programa na jeziku *C++*, već kao datoteka koja sadrži izvorni tekst programa na jeziku *C*.

1.2.2 Rad pod operativnim sistemom *MS-DOS* i prevodiocem *Borland C++*

Prilikom rada na ličnim računarima tipa *IBM-PC* pod operativnim sistemom *MS-DOS* postoji veći izbor mogućnosti nego pod operativnim sistemom *UNIX*. Pored raznih urednika teksta na raspolaganju su i nekoliko prevodilaca za jezik *C++*. Svakog od tih prevodilaca prati i odgovarajuće okruženje u kome radi.

U današnje vreme, kada se uglavnom koristi operativni sistem *MS-Windows*, ovde opisani način rada može da se sprovodi u komandnom prozoru.

Za prikazivanje odabran je rad s prevodiocem *Borland C++* iz razloga što taj prevodilac radi u otvorenom okruženju operativnog sistema *MS-DOS*, što se s njim radi vrlo slično kao i pod operativnim sistemom *UNIX* i što se s njim lako obrađuju veliki programski sistemi čiji se izvorni tekst nalazi u više datoteka. Potrebe većeg dela ove knjige zadovoljava bilo koja verzija prevodioca *Borland C++* počev od verzije 3.0. Za rukovanje izuzecima neophodna je verzija 4.0 ili neka novija. Za sve *Standard* predvidene mogućnosti neophodna je verzija 5.5 ili neka novija.

Za sastavljanje izvornog teksta programa može da se koristi bilo koji urednik teksta. To može da bude standardni urednik teksta *EDIT* koji se nudi pod operativnim sistemom *MS-DOS*, ali može da se koristi i *NOTEPAD* operativnog sistema *Windows*.

U slučaju rada na jeziku *C++* ime datoteke mora da ima proširenje *.CPP*. Komanda upravljačkog jezika operativnog sistema *MS-DOS* za unošenje i obradu izvornog teksta programa u datoteku *imeprog.CPP* je:

EDIT *imeprog.CPP*

Po završetku sastavljanja izvornog teksta programa, jednom od komandi, zavisno od verzije prevodioca:

BCC *imeprog*
BCC32 *imeprog*

prevodi se izvorni tekst programa na jeziku *C++* iz datoteke *imeprog.CPP*. Rezultat prevodenja smešta se u datoteku *imeprog.OBJ*, i odmah se stvara i povezani oblik programa koji se smešta u datoteku s imenom *imeprog.EXE*.

U slučaju sastavljanja velikog programskega sistema vrlo često se izvorni tekst glavnog programa (glavne funkcije) i svih potrebnih potprograma (funkcija) smešta u više datoteka. U tom slučaju potrebno je da se preskoči povezivanje prilikom prevodenja sadržaja datoteke u kojima se nalaze samo potprogrami. To se postiže dodavanjem opcije */C* u komandi *BCC*, odnosno *BCC32*:

BCC */C* *imeprog*
BCC32 */C* *imeprog*

1.2.2 Rad pod operativnim sistemom *MS-DOS* i prevodiocem *Borland C++*

U komandi *BCC*, odnosno *BCC32* može da bude naveden i čitav niz imena datoteka, od kojih neke mogu da sadrže izvorne tekstove a neke prevedene oblike programskega modula. Pomoću komande:

BCC *ime1* *ime2.OBJ* ... *imeN.CPP*
BCC32 *ime1* *ime2.OBJ* ... *imeN.CPP*

prevodiće se sadržaji datoteka čija imena nemaju proširenje ili imaju proširenje *.CPP*, i rezultati prevodenja smeštaće se u odgovarajuće datoteke s proširenjem imena *.OBJ*. Posle toga povezaće se programske moduli iz svih novostvorenih datoteka kao i iz datoteka čija imena u prethodnim komandama imaju proširenja *.OBJ*. Povezani oblik celokupnog programskega paketa smešta se u datoteku s imenom *ime1.EXE*.

Na kraju, izvršavanje programa postiže se komandom koja se sastoji samo od imena datoteke, bez proširenja *.EXE*, koja sadrži povezani oblik programa. Komanda u slučaju datoteke *imeprog.EXE* je:

imeprog

Posle ove komande počinje izvršavanje programa. U toku rada programa korisnik treba da preko tastature unosi podatke po redosledu kako ih očekuje program s „glavnog ulaza” računara. Na ekranu će se pojaviti podaci koji se unose preko tastature kao i podaci koje program piše na „glavni izlaz” računara.

Pod operativnim sistemom *MS-DOS* postoji vrlo elegantan način za „skretanje” glavnog ulaza i/ili glavnog izlaza. Time se postiže da se podaci čitaju iz neke datoteke umesto s tastature, odnosno da se upisuju u neku datoteku umesto ispisivanja na ekranu. To se postiže izvršavanjem programa pomoću komande oblike:

imeprog <podaci >rezult

Znak *<* ispred imena označava da se čitanje s glavnog ulaza vrši iz te datoteke (strelica koja „izlazi“ iz imena datoteke), a znak *>* ispred imena da se pišanje na glavnom izlazu vrši u tu datoteku (strelica koja „ulazi“ u ime datoteke). Datoteke *podaci* i *rezult* su tekstualne datoteke, što znači da mogu da se obrađuju urednikom teksta ili drugim programima za obradu teksta.

Operativni sistem *MS-DOS* ne pravi razliku između malih i velikih slova. Zbog toga, mala i velika slova u navedenim komandama mogu da se koriste na potpuno proizvoljan način. To važi i za imena datoteka, pa *ALFA.CPP* i *aLFA.cPP* predstavljaju istu datoteku koja sadrži tekst izvornog programa na jeziku *C++*.

Na kraju ovog izlaganja neophodno je da se napomene da firma *Borland* pored gore-pomenutog prevodioca za jezik *C++* nudi i takozvano „integrisano okruženje“ za razvoj programa na jeziku *C++* koje se pokreće komandom *BC*, odnosno kod novijih verzija komandom *BCB*. To okruženje u sebi sadrži urednik teksta, prevodilaca, alate za otkrivanje grešaka, alat za rukovanje razvojem velikih programskega sistema i još neke druge alate koji čine rad udobnijim od gore opisanog rada u otvorenom okruženju.

Integrisano okruženje firme *Borland* počev od verzije 4.0 je u obliku aplikacije za operativni sistem *Windows*. Komunikacija s korisnikom se obavlja pomoću grafičke površi. Podržava razvoj programa kako za *MS-DOS*, tako i za *Windows*. Naravno, što je neki alati moćniji, to je on glomazniji i složeniji za korišćenje.

Počev od verzije prevodioca 5.0, integrirano okruženje se naziva *C++ Builder*. Razvoj aplikacija za *Windows* u ovom okruženju mnogo je udobniji nego u ranijim okruženjima.

1.3 Programski jezik C

1.3.1 Elementi jezika

Skup znakova koji se koriste u jeziku C čine mala i velika slova engleskog alfabeta, deset decimalnih cifara i veći broj znakova interpunkcije. Pravi se razlika između malih i velikih slova.

Leksički simboli su nedeljni nizovi znakova. U jeziku C dele se na identifikatore, konstante, službene reči, operatore i separatore.

Identifikatori označavaju sve vrste elemenata programa: promenljivih, simboličkih konstanti, tipova podataka, oznaka i funkcija. Mogu da se sastoje od slova, cifara i znaka podvučeno (_), s tim da prvi znak ne sme da bude cifra. Mogu da imaju proizvoljnu dužinu s tim da je najmanje 31 znak značajan. Izuzetak su „spoljašnja” imena kod kojih su najmanje 6 znakova značajni i to bez razlikovanja malih i velikih slova. Današnji prevodioci uglavnom zanemaruju ovo ograničenje.

Službene reči jezika C su rezervisane reči i ne mogu da se koriste kao identifikatori. Potpun spisak službenih reči, uključujući i one uvedene u jeziku C++, dat je u odeljku 2.1.2.

Leksički simboli mogu da se pišu, uz nekoliko izuzetaka, spojeno ili međusobno razdvojeno proizvoljnim brojem „belih” znakova. U bele znakove spadaju znak za razmak, vertikalna tabulacija, prelazak u novi red i prelazak na novi list.

U širem smislu, u bele znakove se ubrajaju i komentari. Komentari su proizvoljni tekstovi koji stavljeni između /* i */ predstavljaju objašnjenja čitaocu programa. Mogu da se stave između bilo koja dva leksička simbola (kao i beli znakovi) i mogu da se protežu i kroz više redova.

1.3.2 Tipovi podataka

Tipovi podataka određuju moguće vrednosti koje podaci mogu da imaju i moguće operacije koje mogu da se izvode nad tim podacima.

Podaci mogu da budu *prosti* (skalarni, nestrukturirani) ili *složeni* (strukturirani). Prosti podaci ne mogu da se dele na manje delove koji bi mogli nezavisno da se obrađuju. Složeni podaci se sastoje od nekoliko elemenata koji i sami mogu da budu prosti ili složeni.

Od prostih tipova jezik C poznaje samo numeričke tipove. Među njima razlikuju se celi brojevi i realni brojevi.

Osnovne celobrojne tipove čine tipovi **char** i **int**. Varijante tih tipova označavaju se dodavanjem modifikatora ispred ovih oznaka: **unsigned char**, **signed char**, **short int**, **long int**, **unsigned int**, **unsigned short int** i **unsigned long int**. Ako se koristi modifikator reč **int** može da se izostavi (na primer: **short** znači **short int**).

Logički podaci (tip **logical**) ne postoje kao zaseban tip podataka. Kada je to potrebno celobrojni podaci (tip **int**) vrednosti nula tumače se kao logička neistina (**false**), bilo koja nenulta vrednost tumači se kao logička istina (**true**).

Osnovne realne tipove čine tipovi **float** (jednostruka tačnost) i **double** (dvostruka tačnost), a jedina varijanta tih tipova označava se sa **long double** (višestruka tačnost).

Za sve osnovne tipove, uključujući i njihove varijante, postoje odgovarajuće konstante.

Celobrojne konstante mogu da budu decimalne (ako prva cifra nije 0), oktalne (ako počinju sa 0) ili heksadecimalne (ako počinju sa 0x ili 0X). Zavisno od vrednosti, one su tipa

1.3.2 Tipovi podataka

int ili **long**. Za neoznačene varijante treba da se koristi sufiks **u** ili **U**, a za dugačku varijantu, bez obzira na vrednost, sufiks **1** ili **L**.

Znakovne konstante su celobrojne konstante (tip **int**) čije su vrednosti kodovi naveđenih znakova. Pišu se u jednom od oblika 'z', '\ooo', '\xhn', '\Xhn' ili '\u', gde su *z* – bilo koji štampanjući znak osim znakova \, ' i ", *ooo* – niz oktalnih cifara, *hn* – niz heksadecimalnih cifara, a *u* – jedan od znakova a, b, f, n, r, t, v, \, ' ili ".

Nabrojane konstante su celobrojne (tip **int**) konstante koje se definišu nabranjem u naredbama oblika:

```
enum ime_nabranja { ime_konstante = vrednost_konstante , ... } ;
```

Ime nabranja je identifikator pomoću kojeg je moguće kasnije pozivati se na posmatrano nabranje. *Imena konstanti* su identifikatori simboličkih konstanti kojima se dodeljuju vrednosti konstantnih celobrojnih izraza označenih sa *vrednost konstante*. Ako iza imena neke konstante ne стоји vrednost dodeliće joj se vrednost koja je za jedan veća od vrednosti prethodne konstante u nizu, odnosno koja je nula ako se radi o prvoj konstanti u nizu.

Za realne konstante koristi se isključivo decimalni brojevni sistem i podrazumeva se dvostruka tačnost (**double**). Za jednostruku tačnost treba da se koristi sufiks **f** ili **F**, a za višestruku tačnost **l** ili **L**. Najopštiji oblik za sve tačnosti je eksponencijalni oblik *mEk*, gde se *m* naziva mantisa, a *k* eksponent. Vrednost predstavljenog broja je *m* × 10^{*k*}. Delovi najopštijeg oblika mogu da nedostaju, ali decimalna tačka u mantisi ili eksponent moraju da budu prisutni.

Podaci se definišu naredbama oblika:

```
modifikator opis_tipa naziv_podatka = početna_vrednost , ... ,  
naziv_podatka = početna_vrednost ;
```

Opis tipa može da bude ime bilo koje od ranije pomenutih osnovnih tipova sa svim mogućim varijantama, oznaka nekog složenog tipa ili detaljan opis složenog tipa. Za slučaj nabranja treba pisati **enum** *ime_nabranja*, ili celokupan opis nabranja.

Naziv podatka za proste tipove je jednostavno identifikator podatka. Za dole opisane izvedene tipove uz identifikator mogu da stoje još i odgovarajući *modifikatori*.

Početna vrednost može da bude proizvoljan izraz odgovarajućeg tipa pod uslovom da svi operandi u izrazu imaju definisane vrednosti u momentu izvršavanja posmatrane deklarativne naredbe. U nedostatku =*početna_vrednost* odgovarajući podatak ima slučajnu početnu vrednost. Dodata početne vrednosti prilikom definisanja podataka naziva se *inicijalizacija*, a sama početna vrednost *inicijalizator*.

Modifikator na početku naredbe, ako postoji, može da bude **const** ili **volatile**. Ako nema modifikatora, definisanim podacima se može menjati vrednost u toku izvršavanja programa (*promenljivi podaci*). Modifikator **const** označava da vrednosti definisanih podataka ne mogu da budu promenjene u toku izvršavanja programa (*nepromenljivi podaci*). U tom slučaju moraju da budu navedene početne vrednosti za sve podatke, tj. moraju da se navedu inicijalizatori. Bez obzira na nepromenljivost takvi podaci ne mogu da se koriste na mestima gde se izričito zahtevaju konstante. Modifikator **volatile** označava da se vrednosti definisanih podataka mogu promeniti mimo kontrolu programa (*nepostojani podaci*).

Svakom osnovnom tipu, a i dole opisanim izvedenim tipovima, može da se dodeli neko ime naredbom oblika:

```
typedef opis_tipa Naziv_tipa ;
```

Opis tipa podleže istim pravilima kao i u slučaju naredbe za definisanje podataka. *Naziv tipa* za slučaj prostih tipova je identifikator koji se dodeljuje odabranom tipu. Za neke izvedene tipove uz taj identifikator treba da stoe i odgovarajući modifikatori. Identifikatori koji se uvode naredbama **typedef** kasnije mogu da se koriste kao oznaka tipa ravnopravno s imenima osnovnih tipova.

Polazeći od gore opisanih osnovnih tipova podataka može da se sastavi neograničeni broj *izvedenih tipova*. U izvedene tipove u jeziku C spadaju pokazivači, nizovi, strukture, unije i strukture od bitova.

Pokazivači su prosti podaci u koje mogu da se smeste adrese podataka ili potprograma u operativnoj memoriji. Broj bajtova koje zauzima neki pokazivač zavisi od mogućeg opsega adresa na datom računaru, a ne od broja bajtova koje zauzima pokazivani podatak.

Pokazivači se deklarišu naredbama za definisanje podataka dodavanjem modifikatora ***** (promenljiv pokazivač), ***const** (nepromenljiv pokazivač) ili ***volatile** (nepostojan pokazivač) ispred identifikatora pokazivača u nazivima podataka. Oznaka tipa na početku naredbi predstavlja tip pokazivanih podataka. Eventualni modifikator ispred oznake tipa označava nepromenljivost ili nepostojanost pokazivanih podataka, a ne pokazivača samih.

Dva pokazivača su istog tipa ako pokazuju na podatke istih tipova. Specijalan tip za pokazivače je generički pokazivač (tip **void***) kod kojeg se ne zna tip pokazivanih podataka.

Postoji jedna jedinica konstanta pokazivačkog tipa, 0 koja označava da dati pokazivač ne pokazuje ni na jedan podatak. Za simboličko označavanje te vrednosti može da se koristi simbolička konstanta **NULL**.

Nizovi su složeni podaci koji se sastoje od više elemenata međusobno jednakih tipova. Ako su elementi nekog niza prostog tipa, govori se o jednodimenzionalnom nizu ili vektoru. Niz je dvodimenzionalan (matrica) ako su mu elementi jednodimenzionalni nizovi. Nizovi mogu da budu s proizvoljnim brojem dimenzija.

Podaci tipa niza deklarišu se naredbama za definisanje podataka dodavanjem modifikatora [**dužina**] iza identifikatora niza. Oznaka tipa u naredbi označava tip elemenata niza.

Dužina predstavlja broj elemenata niza i treba da je celobrojan konstantan izraz. Elementi niza se obeležavaju rednim brojevima 0, 1, ..., **dužina**-1 koji se nazivaju *indeksi elemenata*. Za višedimenzionalne nizove potrebno je navesti po jedan modifikator navedenog oblika za svaku dimenziju.

Niske znakove nizovi (tip **string**) ne postoje kao poseban tip podataka. Za njih se koriste nizovi znakova (tip **char[]**) u kojima se iza poslednjeg korisnog znaka nalazi još jedan element sa sadržajem '\0'.

Konstante nizovnog tipa postoje samo za niske znakove i pišu se u obliku "tekst", gde je **tekst** proizvoljan niz znakova koji čini vrednost konstantne niske. Između znakova navoda mogu da se koriste svi ranije pomenuti oblici predstavljanja znakovnih konstanti.

Početne vrednosti podacima tipa niza mogu da se dodeljuju u obliku niza vrednosti stavljениh između para vitičastih zagrada:

```
{ vrednost , vrednost , ... , vrednost }
```

1.3.2 Tipovi podataka

s tim da *vrednosti* moraju da budu isključivo konstantni izrazi. U slučaju niza znakova (tip **char[]**) može da se koristi i notacija za konstantne niske. Za višedimenzionalne nizove svaka vrednost i sama treba da bude niz vrednosti navedenog oblika.

Strukture su složeni tipovi podataka koji se sastoje od uređenih nizova elemenata koji mogu da budu međusobno različitih tipova. Ti elementi nazivaju se *polja strukture*. Polja strukture se obeležavaju identifikatorima. Opšti oblik opisa strukture u naredbama za definisanje podataka je:

```
struct ime_strukture { niz_deklaracija }
```

Ime strukture je identifikator definisane strukture. Ono nema status identifikatora tipa, tako da u naredbama za definisanje podataka i u naredbama **typedef** kao opis tipa mora da se piše **struct** *ime strukture*. Niz deklaracija se sastoji od niza naredbi za definisanje podataka, pri čemu nije dozvoljeno korišćenje modifikatora **const** i **volatile**, niti navođenje početnih vrednosti. Identifikatori koji se uvode na taj način predstavljaju identifikatore polja strukture.

Bez obzira na svoju složenost, strukture se smatraju pojedinačnim podaćima (ne nizovima) i često mogu da se koriste na isti način kao i prosti (skalarni) podaci.

Notacija za navođenje početnih vrednosti struktura je slična kao i u slučaju nizova, s tim što navedene vrednosti moraju da se slažu po tipu s odgovarajućim članovima strukture.

Unije su složeni tipovi podataka koje omogućavaju da se u isti memorijski prostor, u raznim trenucima, smeštaju podaci različitih tipova. Opšti oblik opisa unija je istovetan opisu struktura, osim što umesto reči **struct** treba da se koristi reč **union**. Za razliku od struktura, gde svi članovi istovremeno imaju definisane vrednosti, kod unija svakog momenta samo jedan član ima definisanu vrednost. To je poslednji od članova kome je dodeljena vrednost. Početna vrednost može da bude dodeljena samo prvom članu date unije.

Strukture od bitova su strukture čija su polja dužine nekoliko bitova koji se, na mašinski zavisan način, pakuju u mašinske reči računara. Definišu se opisima **struct** kod kojih su sva polja nekog celobrojnog tipa (najčešće **unsigned int**). Iza identifikatora svakog od polja treba da se navede dužina u obliku :*a*, gde je *a* broj bitova koje dati član zauzima.

1.3.3 Operatori i izrazi

Operatori predstavljaju radnje koje se izvršavaju nad operandima dajući pri tome određene rezultate. Izrazi su proizvoljno složeni sastavi operanada i operatora.

Aritmetički operatori izvode osnovne aritmetičke operacije. U njih spadaju binarni operatori za sabiranje (+), oduzimanje (-), množenje (*), deljenje (/) i nalaženje ostatka deljenja celih brojeva (%). Unarni operator + nema nikakvog efekta, a - služi za izmenu algebarskog znaka broja. U jeziku C postoje još dva unarna operatora za povećavanje (++) i smanjivanje (--) vrednosti operanda za jedan. Ova dva operatora mogu da budu napisana ispred operanda (prefiksna notacija) kada je vrednost izraza nova vrednost operanda, ili iza operanda (postfiksna notacija) kada je vrednost izraza vrednost operanda pre promene.

Relacijski operatori upoređuju numeričke podatke i daju logičke vrednosti. U jeziku C to znači celobrojnu vrednost 0 za logičku neistinu, odnosno celobrojnu vrednost 1 za logičku istinu. Ti operatori se obeležavaju sa == (jednako), != (različito), < (manje), <= (manje ili jednako), > (veće) i >= (veće ili jednako).

Logički operatori izvode logičke operacije nad logičkim podacima dajući logički rezultat. U jeziku C nulta vrednost operanda smatra se logičkom neistinom, a bilo koja ne-nulta vrednost logičkom istinom. Rezultati su 0 za logičku neistinu i isključivo 1 za logičku istinu. Postoje unarni operator za logičku *ne* (!) operaciju i binarni operatori za logičku *i* (*&&*) i *ili* (*||*) operaciju. U slučaju operatora *&&* prvo se izračunava vrednost prvog operanda i ako je to =0, rezultat je 0 i vrednost drugog operanda uopšte se ne izračunava. U slučaju operatora *||* prvo se izračunava vrednost prvog operanda i ako je to ≠0, rezultat je 1 i vrednost drugog operanda se uopšte ne izračunava.

Operatori po bitovima od standardnih viših programskih jezika postoje samo u jeziku C i u jezicima koji su proizašli iz jezika C (C++, Java). Oni obavljaju manipulacije nad bitovima unutar celobrojnih podataka. Postoje unarni operator za komplementiranje bit po bit (~) i binarni operatori za logičke operacije *i* (*&*), *uključivo ili* (*||*) i *isključivo ili* (*^*), bit po bit. Pored toga postoje binarni operatori za pomeranje binarne vrednosti prvog operanda uлево (<<) i уdesno (>>) za broj mesta koji je jednak vrednosti drugog operanda.

Uslovni operator (*? :*) je specifičan ternarni operator jezika C. Ima tri operanda i piše se u obliku *a**? b:c***. Prvo se izračunava vrednost izraza *a* i ako ima vrednost logičke istine (*≠0*), izračunava se vrednost izraza *b* i to predstavlja vrednost celog izraza. Ako *a* ima vrednost logičke neistine (*=0*), izračunava se vrednost izraza *c* i to predstavlja rezultat celog izraza. Bitno je da se od izraza *b* i *c* uvek izračunava samo jedan.

Adresni operatori manipulišu adresama podataka. U užem smislu tu spadaju unarni operatori za nalaženje adrese datog podatka (*&*) i za posredan pristup podatku pomoću pokazivača (*), koji se naziva i operator indirektnog adresiranja. U širem smislu u adresne operatori spadaju i aditivni aritmetički operatori. Naime, dozvoljeno je na vrednost nekog pokazivača (adrese) dodati celobrojnu vrednost, odnosno od nje oduzeti celobrojnu vrednost. Jedinica mere promene adrese u tim slučajevima je veličina pokazivanih podataka. Drugim rečima ako pokazivač *p* pokazuje na neki element datog niza tada je *p+1* pokazivač na naredni, a *p-1* na prethodni element tog niza. Slično tome, dozvoljeno je oduzeti vrednost dva pokazivača koji pokazuju na elemente istog niza. Pošto je i sada jedinica mere veličina pokazivanih podataka, rezultat je razlika indeksa ta dva elementa niza. Na kraju, dozvoljeno je i upoređivati vrednosti dva pokazivača relacijskim operatorima, s tim da upotreba operatora *<*, *<=*, *>* i *>=* ima smisla samo ako obe operande pokazuju na elemente istog niza.

Operatori za dodelu vrednosti su, takođe, jedna od specifičnosti jezika C. Naime, dok je u drugim jezicima dodela vrednosti naredba, u jeziku C je operator (=) i zbog toga, može da bude i više dodeljivanja vrednosti u sastavu istog izraza (naredbe). Pored toga, postoje još deset operatora dodele vrednosti za slučajeve kada rezultat nekog binarnog operatora treba da se smešta u prvi operand tog operatora. To su operatori *+=*, *-=*, **=*, */=*, *%=*, *&=*, *|=*, *^=*, *<= i >=*. Izraz *a@=b* se tumači kao *a=a@b*, osim što se izraz *a* izračunava samo jednom. Pošto se strukture smatraju pojedinačnim podacima, mogu se međusobno dodeljivati operatorom za dodelu vrednosti (=).

Operator zarez (,) služi za lančanje izraza. U slučaju izraza *a, b* prvo se izračunava vrednost izraza *a*, posle toga vrednost izraza *b* i rezultat celokupnog izraza je *b*, nezavisno od vrednosti *a*. Ovo u slučaju lanca od više izraza (na primer: *a, b, c, d*) znači da se izrazi u lancu izračunavaju sleva udesno i vrednost celog izraza je vrednost poslednjeg izraza u lancu.

Operator za konverziju tipa pretvara tip vrednosti svog operanda u odgovarajuću vrednost naznačenog tipa. Izraz za konverziju tipa podatka *a* u tip *T* piše se u obliku (*T*) *a*. *T* može da bude oznaka tipa bilo kog osnovnog tipa, identifikator koji je nekom tipu dodeljen naredbom **typedef** ili oznaka pokazivačkog tipa na podatke proizvoljnih tipova.

Veličina podataka može da se dobije unarnim operatorom **sizeof**. Operand može da bude naziv tipa podataka unutar para oblih zagrada ili proizvoljan izraz. U slučaju izraza dobija se veličina rezultata bez izračunavanja vrednosti izraza. Po definiciji **sizeof** (**char**) je uvek 1. Za slučaj nizova rezultat je veličina celog niza (veličina elemenata pomnožena s brojem elemenata).

Indeksiranje, kako se naziva pristup elementima nizova, u jeziku C se smatra binarnim operatorom ([]) koji se piše oko svog drugog operanda: *a[b]*, gde *a* predstavlja niz (u obliku identifikatora niza ili adresnog izraza), a *b* indeks (redni broj) željenog elementa u obliku celobrojnog izraza. Izraz s indeksiranjem *a[b]* i izraz adresne aritmetike **(a+b)* su međusobno istovetni. Istovetnost važi i za izraze *&a[b]* i *a+b*. Identifikator niza u izrazima predstavlja pokazivač na prvi element niza.

Pristup poljima struktura i unija vrši se binarnim operatorima . ili \rightarrow . Prvi operand operatora . treba da je podatak, a operatora \rightarrow pokazivač na podatak tipa strukture ili unije. Drugi operand oba operatora treba da je identifikator polja strukture ili unije. Izraz *a->b* je istovetan izrazu (**a*).*b*.

Pozivanje funkcije u jeziku C smatra se binarnim operatorom () koji se piše oko svog drugog operanda: *f(a, b, c)*. Prvi operand je pozivana funkcija *f* predstavljena identifikatorom ili adresnim izrazom. Drugi operand je niz argumenata *a, b, c*. Argumenti *a, b* i *c* su proizvoljni izrazi potrebnih tipova. Zarez (,) ovde predstavlja separator među argumentima, a ne operator *zarez* za stvaranje niza izraza. Ako neki od argumenata treba da je niz izraza, isti mora da se stavlja unutar para oblih zagrada.

Lvrednosti je izraz koji označava neki podatak ili funkciju u memoriji. Imena podataka predstavljaju **lvrednosti**. Rezultat primene operatora indirektnog adresiranja (*) i indeksiranja ([]) za slučaj jednodimenzionalnog niza je **lvrednost**. Operandi unarnih operatora *&*, *++* i *--*, kao i operand na levoj strani svih operatora za dodelu vrednosti moraju da budu **lvrednosti**. Kovanica **lvrednosti** upravo potiče otuda: to je vrednost koja može da stoji na levoj strani operatora za dodelu vrednosti.

Ako su operandi nekog operatora različitih tipova, primenjuje se automatska konverzija tipa operanda „jednostavnijeg“ tipa u tip operanda „složenijeg“ tipa i onda se izračunava rezultat operatora. Najjednostavniji mogući tip rezultata je **int** (tipovi **char** i **short** uvek se pretvore u tip **int**). Za njim slede tipovi **unsigned int**, **long**, **unsigned long**, **float**, **double** i **long double**.

Kod dodele vrednosti tip numeričkog rezultata koji se dodeljuje pretvara se u tip numeričkog podatka kome se dodeljuje vrednost. U slučaju pokazivača dodela vrednosti je dozvoljena samo ako su obe operande pokazivači na podatke istih tipova, osim ako je jedan od operanada generički pokazivač (tip **void***). U ostalim slučajevima mora da bude eksplicitno naznačena konverzija tipa dodeljivanog pokazivača.

Redosled izvršavanja susednih operatora u složenim izrazima određuje se zagradama (()), prioritetom operatora i smerom grupisanja operatora. Pregled prioriteta i smerova grupisanja svih operatora, uključujući i nove operatore koji su uvedeni u jeziku C++, dat je u tablici 2.1 u odeljku 2.3.

Redosled izračunavanja operanada operatora i argumenata funkcija u jeziku C nije propisan (izuzev za operatore `&&`, `||`, `? :` i `zarez (,)`). Zbog toga redosled izvršavanja nesusednih operatora nije jednoznačno određen. Smatra se lošim stilom programiranja pisanje izraza čiji rezultati zavise od redosleda izračunavanja operanada operatora i argumenata funkcija. Potencijalni izvori zavisnosti od redosleda izračunavanja operanada su operatori i funkcije koji daju više od jednog rezultata, tj. koji pored glavnog rezultata daju i dodatne *bočne efekte*. Operatori s bočnim efektima su `++`, `--` i svi operatori dodele vrednosti.

1.3.4 Naredbe

Naredba je osnovna jedinica obrade u programima. U jeziku C naredba može da bude prosta, složena ili upravljačka.

Prosta naredba se sastoji od izraza iz prethodnog odeljka iza koje se nalazi terminotor tačka-zarez (`izraz;`). Specijalan slučaj proste naredbe je *prazna naredba* koja se sastoji samo od tačka-zareza (`;`).

Složene naredbe predstavljaju složene strukture naredbi kojima se određuje redosled izvršavanja naredbi u programu. Nazivaju se i *upravljačke strukture*. Upravljačke strukture mogu da se podele u tri grupe: sekvenca, selekcije i ciklusi ili petlje.

Upravljačke naredbe prenose tok upravljanja na neko mesto u programu. To su razne naredbe skokova.

Sekvenca je najjednostavnija upravljačka struktura i predstavlja niz naredbi koje se izvršavaju jedna za drugom. Niz naredbi koje čine sekvencu stavlja se između para vitičastih zagrada:

```
{ naredba naredba ... naredba }
```

Svaka naredba u nizu može da bude prosta ili složena. Sekvenca može da bude i prazna (`()`), tj. da ne sadrži nijednu naredbu.

Pošto na početku svake sekvence, pre izvršnih naredbi, mogu da budu i deklarativne naredbe za definisanje podataka i tipova, ova struktura naziva se i *blok*. Dejstvo tih deklarativnih naredbi je do kraja bloka u kome se nalaze.

Selekcije su složene naredbe koje omogućavaju uslovno izvršavanje jedne ili nijedne naredbe iz skupa od jedne ili više naredbi.

Osnovnom selekcijom se uslovno izvršava jedna od dve naredbe. U jeziku C ostvaruje se naredbom oblike:

```
if ( izraz ) naredba_1 else naredba_2
```

Ukoliko logički `izraz` ima vrednost logičke istine (`!=0`), izvršava se `naredba_1`. Ako je vrednost izraza logička neistina (`=0`), izvršava se `naredba_2`. Deo `else naredba_2` može da nedostaje. Tada se, u stvari, radi o uslovnom izvršavanju ili preskakanju `naredbe_1`. Takva upravljačka struktura naziva se i *uslovni preskok*.

Prilikom uklapanja osnovnih selekcija, tj. kada su `naredba_1` i/ili `naredba_2` i same osnovne selekcije, ponekad se javlja problem u odlučivanju koja reč `else` pripada kojoj reči `if`. Opšte pravilo glasi: službena reč `else` uvek pripada najbližoj reći `if` koja se nalazi ispred nje i za koju još nije pronađen njen `else` deo. Od ovog pravila može da se odstupi korišćenjem sekvenci.

Selekcija pomoću skretnice sastoji se od niza naredbi i celobrojnog izraza čija vrednost određuje prvu naredbu u nizu odakle počinje izvršavanje. U jeziku C ostvaruje se naredbom oblike:

1.3.4 Naredbe

```
switch ( izraz ) {
    case α : niz_naredbi_1
    case β : niz_naredbi_2
    ...
    default: niz_naredbi_I
    ...
    case ω : niz_naredbi_N
}
```

Oznake `case α:`, `case β:`, ..., `case ω:` označavaju mesta u nizu naredbi odakle počinje izvršavanje ako celobrojan `izraz` ima vrednost α , β , ..., ω , respektivno. Specijalna oznaka `default` označava mesto na koje se skače ukoliko vrednost izraza nije jednakoj od vrednosti u oznakama `case`. Posle izvršenog skoka naredbe se izvršavaju redom, do kraja celokupne upravljačke strukture, zanemarujući eventualne usputne oznake. U odsustvu oznake `default` može da se desi da nijedna naredba u nizu ne bude izvršena.

Ciklusi su upravljačke strukture koje omogućavaju ponovljeno izvršavanje neke naredbe (proste ili složene). U jeziku C postoje tri vrste ciklusa.

Osnovni ciklus s izlazom na vrhu ostvaruje se naredbom oblike:

```
while ( izraz ) naredba
```

Na početku svakog prolaska kroz ciklus izračunava se vrednost logičkog `izraza` i ako se dobija logička istina (`!=0`), izvršava se `naredba`. Ciklus se završava kada vrednost `izraza` postane logička neistina (`=0`). Može da se desi da `naredba` ne bude nijednom izvršena.

Generalizovani ciklus s izlazom na vrhu ostvaruje se naredbom oblike:

```
for ( izraz_1 ; izraz_2 ; izraz_3 ) naredba
```

Prvo se izračunava vrednost `izraza_1`, kao priprema za ulazak u ciklus. Posle toga, na početku svakog prolaska kroz ciklus izračunava se vrednost logičkog `izraza_2` i ako se dobija logička istina (`!=0`) izvršava se `naredba` i izračunava se `izraz_3` kao priprema za naredni prolazak kroz ciklus. Ciklus se završava kada vrednost `izraza_2` postane logička neistina (`=0`). I ovde može da se desi da `naredba` ne bude nijednom izvršena. Bilo koji od izraza može da nedostaje. U slučaju kada nedostaje `izraz_2`, smatra se da uvek ima vrednost logičke istine. To znači da tada ciklus nema prirođan završetak.

Ciklus s izlazom na dnu ostvaruje se naredbom oblike:

```
do naredba while (izraz) ;
```

Prvo se izvršava `naredba` koja čini sadržaj ciklusa. Posle toga izračunava se vrednost logičkog `izraza` i ako se dobija logička istina (`!=0`), skoci se na ponovno izvršavanje `naredbe`. Ciklus se završava kada vrednost `izraza` postane logička neistina (`=0`). U ovom slučaju `naredba` se uvek izvršava bar jednom.

Skokovi su upravljačke naredbe kojima se tok upravljanja prenosi na neko drugo mesto u programu. U jeziku C postoje tri vrste skokova.

Iskakanje iz upravljačke strukture izvršava se naredbom `break;`. Time se postiže preskakanje preostalih naredbi unutar selekcije pomoću skretnice (`switch`) skakanjem na prvu naredbu neposredno iza selekcije, ili prevremenim završetkom ciklusa (`while`, `for`, `do`) skakanjem na prvu naredbu neposredno iza ciklusa.

Skok na kraj ciklusa izvršava se naredbom `continue;`. Time se postiže preskakanje preostalih naredbi do kraja ciklusa i izvršavanje radnji vezanih za opsluživanje ciklusa. To

u slučaju ciklusa **while** i **do** znači ponovno izračunavanje *izraza*, odnosno u slučaju ciklusa **for** izračunavanje *izraza 3* pa *izraza 2*.

Skok s proizvoljnim odredištem izvršava se naredbom oblika:

```
goto oznaka ;
```

Oznaka predstavlja odredište skoka i po formi je identifikator. Mesto u programu na koje treba skočiti obeležava se umetanjem *oznake*: ispred naredbe na koju treba da se prenese upravljanje. Odredište skoka može da bude bilo gde u programu, uz jedino ograničenje da ne sme da se uskače u blok na čijem početku se nalaze neke definicije koje bi se time preskočile. Zbog teško sagledivog ponašanja programa pri uskakanju u blok koji je deo složene naredbe (selekcije ili ciklusa), kao praktično pravilo uzima se da uskakanje u blok nije dozvoljeno.

1.3.5 Funkcije

Programski jezik C, formalno, poznaje samo jednu vrstu potprograma koji se nazivaju *funkcije*. Funkcije, na većini programskih jezika, na osnovu izvesnog broja *parametara* daju jedan rezultat koji se naziva *vrednost funkcije*. Jezik C dozvoljava da funkcije daju i druge rezultate koji se nazivaju *bočni efekti*. Sama vrednost funkcije često se i ne koristi, a mogu da se prave i funkcije koje uopšte ne stvaraju vrednost funkcije, već samo bočne efekte.

Definisanje funkcija vrši se naredbama za definisanje funkcije čiji je opšti oblik:

```
oznaka_tipa naziv_funkcije ( niz_parametara ) telo_funkcije
```

Oznaka tipa predstavlja tip vrednosti funkcije. Vrednost funkcije može da bude jedan pojedinačan podatak bilo kog tipa, uključujući i strukture i unije, ali ne može da bude tipa niza. Vrednost funkcije može da bude i pokazivačkog tipa. Za funkcije koje ne daju vrednost funkcije, kao oznaka tipa koristi se službena reč **void**.

Naziv funkcije je identifikator s eventualnim modifikatorom * ukoliko je vrednost funkcije pokazivač na podatke tipa navedenog u oznaci tipa.

Niz parametara je niz deklaracija parametara funkcije koji se nazivaju i *formalni argumenti* funkcije. Svaka deklaracija treba da sadrži oznaku tipa, identifikator parametra i, eventualno, modifikatore * ili [duž] za pokazivačke i nizovne parametre (za prvu dimenziju niza ne mora da se navede dužina, samo [] je dovoljno). Pored toga, mogu da se koriste i modifikatori **const** za označavanje da funkcija ne menja vrednost na koji pokazuje dati pokazivački parametar, ili **volatile** za označavanje da je funkcija spremna na promene vrednosti pokazivanog parametra mimo kontrole programa. Niz parametara može biti prazan. Tada između oblih zagrada treba da se stavi službena reč **void**.

Telo funkcije je po formi blok. Na početku bloka mogu da se deklarišu podaci lokalni za funkciju koji se koriste u izvršnom delu funkcije. Parametri funkcije smatraju se lokalnim podacima funkcije koji se prilikom pozivanja funkcije inicijalizuju vrednostima odgovarajućih argumenata koji se nazivaju i *stvari argumenti*. Posledica toga je da vrednost argumenta ne može da se promeni. Ako je parametar pokazivač može da se promeni vrednost pokazivanog podatka. Na taj način se stvaraju bočni efekti funkcije.

Vrednost funkcije u telu funkcije zadaje se u naredbi za povratak u pozivajuću funkciju, čiji je opšti oblik:

```
return izraz ;
```

1.3.5 Funkcije

gde *izraz* mora po tipu da se slaže s naznačenim tipom vrednosti funkcije. U slučaju numeričkih tipova tip izraza se, po potrebi, automatski pretvara u predviđeni tip. Kod funkcija koje ne daju vrednost funkcije (tipa **void**), naredba za povratak u pozivajuću funkciju ne sme da sadrži izraz, već samo **return**; Štaviše, naredba **return**; uopšte ne mora da postoji. Dolazak do kraja bloka dovodi do povrata u pozivajuću funkciju.

Deklarisanje funkcija vrši se naredbama oblika:

```
oznaka_tipa naziv_funkcije ( niz_parametara ) ;
```

Za razliku od definisanja, deklaracijom se zadaju samo tip vrednosti funkcije i broj i tipovi parametara. Umesto tela funkcije navodi se samo prazna naredba (;). U nizu parametara identifikatori parametara mogu da se izostave. Rezultat deklarisanja funkcije naziva se *prototip funkcije*. Prototip funkcije treba da bude naveden u slučajevima kada se neka funkcija poziva na mestu na kome definicija funkcije nije dostupna prevodiocu.

Pozivanje funkcija vrši se izrazima oblika:

```
funkcija ( niz_izraza )
```

Funkcija označava funkciju čije se izvršavanje traži. Može da bude identifikator funkcije ili adresni izraz čija je vrednost adresa željene funkcije. Vrednosti izraza u nizu izraza predstavljaju argumente funkcije kojima se inicijalizuju odgovarajući parametri.

Mada funkcije nisu podaci ipak mogu da se definišu *pokazivači na funkcije* opisom oblika:

```
oznaka_tipa ( * identifikator_pokazivača ) ( niz_naziva_tipova )
```

Dodatni par oblih zagrada je neophodan kako prilikom definisanja pokazivača na funkciju, tako i prilikom pozivanja funkcije pomoću pokazivača na tu funkciju.

Glavna funkcija u jeziku C je funkcija koju poziva operativni sistem računara radi izvršavanja programa. Prototip glavne funkcije može biti:

```
void main ( void ) ;  
int main ( void ) ;  
void main ( int bpar , char * vpar [ ] ) ;  
int main ( int bpar , char * vpar [ ] ) ;
```

ili
ili
ili

Prvi parametar *bpar* predstavlja broj argumenata (reči) u komandi operativnog sistema, uključujući i samu komandu za izvršavanje programa. Drugi parametar *vpar* je niz (vektor) pokazivača od *bpar+1* elemenata na niske koji sadrže reči (nizove znakova između dva bela znaka) u komandi operativnog sistema. Vrednost poslednjeg pokazivača (*vpar[bpar]*) je NULL. Ako se za tip vrednosti funkcije navede **int**, vraćena vrednost se dostavlja operativnom sistemu kao završni status programa. Preporučuje se da tip funkcije **main()** bude **int** a ne **void**.

1.3.6 Struktura programa

Izvorni tekst programskog sistema na jeziku C može da se nalazi u proizvoljnem broju datoteka. Svaka datoteka se sastoji od niza definicija i deklaracija podataka i funkcija.

Podaci definisani izvan svih funkcija predstavljaju globalne podatke. Oni mogu da se deklarišu proizvoljan broj puta i da se definišu tačno jednom. Prilikom deklarisanja podatka samo se navede identifikator i tip podatka. Prilikom definisanja, pored deklarisanja

dodeljuje se i memoriski prostor i početna vrednost. Globalni podatak se deklariše dodavanjem modifikatora **extern** na početku naredbe za definisanje podataka.

Funkcije su uvek globalne. I one mogu da se deklarišu proizvoljan broj puta i da se definišu tačno jednom (videti i odeljak 1.3.5).

Doseg identifikatora predstavlja deo programa unutar kojeg identifikator može da se koristi. Identifikatori mogu da se koriste od mesta definisanja do kraja dosega. Postoje pet vrsta dosega.

Datotečki doseg imaju identifikatori definisani izvan funkcija. Ti identifikatori se nazivaju i *globalni identifikatori*. Mogu da predstavljaju funkcije, podatke, tipove podataka i simboličke konstante.

Blokovski doseg imaju identifikatori definisani unutar blokova. Ti identifikatori se nazivaju i *lokalni identifikatori*. Za parametre funkcija se smatra da su definisani na početku bloka koji čini telo funkcije.

Funkcijski doseg imaju samo oznake za skokove naredbama **goto**. Taj doseg se proteže na celo telo funkcije, bez obzira na mesto definisanja oznake unutar tela funkcije.

Prototipski doseg imaju parametri u prototipovima funkcija.

Strukturni doseg imaju identifikatori članova struktura i unija. Oni mogu da se koriste samo primenom operatora za pristup članovima (. ili ->).

Redeklarisanjem identifikatora unutar dosega tog identifikatora spoljašnja deklaracija postaje nevidljiva do kraja unutrašnjeg dosega.

Način povezivanja označava način na koji se povezuju isti globalni identifikatori, korišćeni u različitim datotekama. Isti identifikatori sa *spoljašnjim povezivanjem* označavaju isti podatak ili funkciju unutar svih datoteka programskega sistema. Identifikatori s *unutrašnjim povezivanjem* su lokalni za datoteke u kojima su definisani, pa u svakoj datoteci označavaju međusobno nezavisne podatke ili funkcije.

Globalni identifikatori funkcija i podataka imaju spoljašnje povezivanje, izuzev kada se prilikom njihovog definisanja koristi modifikator **static**. Globalni identifikatori uvedeni naredbama **typedef** ili **enum** uvek imaju unutrašnje povezivanje.

Kod lokalnih identifikatora ne postoji pojam povezivanja, izuzev kada se prilikom njihovog deklarisanja koristi modifikator **extern**. Ti identifikatori tada predstavljaju globalne podatke koji treba da su definisani drugde u programu (u istoj ili drugoj datoteci), jedino što imaju blokovski doseg.

Po **trajnosti** podaci mogu da se podele u četiri grupe.

Trajni podaci postoje od momenta stvaranja do kraja programa. Globalni podaci se stvaraju na početku izvršavanja programa, a lokalni prilikom prvog ulaska u blok unutar koga su definisani. Globalni podaci su uvek trajni, a lokalni mogu da se učine trajnim dodavanjem modifikatora **static**. Trajni podaci imaju nulte početne vrednosti.

Prolazni podaci se stvaraju u momentu definisanja i postoje do napuštanja dosega njihovih identifikatora. To se podrazumeva za sve lokalne podatke. Prolazni podaci imaju slučajne početne vrednosti. Smeštaju se u operativnu memoriju (može da se koristi modifikator **auto**, koji se i podrazumeva) ili u procesorske registre (dodavanjem modifikatora **register**).

Privremeni podaci se stvaraju u toku izvršavanja složenih izraza, kada se pokaže potreba za smeštanjem međurezultata u memoriju. Postoje dok su „neophodni”, obično do kraja izračunavanja tekućeg izraza.

Dinamičke podatke stvara i uništava programer sâm. Postoje od momenta izvršavanja zahteva za njihovim stvaranjem do izvršavanja zahteva za uništavanje, odnosno do kraja programa. Dinamičkim podacima može da se pristupa samo pomoću pokazivača i imaju slučajne početne vrednosti.

1.3.7 Preprocesor

Preprocesor jezika C obavlja određene obrade izvornog teksta programa pre početka prevodenja.

Zamena leksičkih simbola postiže se direktivama sledećih oblika:

```
#define makro niz_simbola
#define makro(parametar,...,parametar) niz_simbola
```

ili

Od mesta ove direkтиve do kraja datoteke zamenjivaće se svako pojavljivanje identifikatora *makro* navedenim *nizom simbola*. Elementi niza mogu da budu bilo koji leksički simboli: identifikatori, konstante, operatori, separatori. Ako se *niz simbola* sastoji samo od jedne konstante identifikator *makro* može da se shvatí i kao simbolička konstanta.

U slučaju oblika s parametrima svako pojavljivanje identifikatora *parametar* unutar *niza simbola* zameniće se odgovarajućim argumentima koji se u tekstu programa navedu uz identifikator *makro*.

Zamena leksičkih simbola može da se prekine pre kraja datoteke umetanjem direkтиве oblika:

```
#undef makro
```

Umetanje sadržaja datoteke postiže se direktivama oblika:

```
#include "imedat"
#include <imedat>
```

ili

U prvom slučaju navedena datoteka se prvo trazi u korisnikovom katalogu, pa ako se ne pronađe pretražuju se sistemske kataloge. U drugom slučaju pretražuju se samo sistemske kataloge.

Umetanje sadržaja datoteke koristi se za efikasno umetanje u tekst programa dužih sekvenci direkтиve **#define**, deklaracija globalnih podataka i funkcija.

Uslovno prevodenje delova izvornog teksta programa postiže se korišćenjem nekih od sledećih direkтиva:

#if izraz	/* Ako je konstantni izraz #0.	*/
#ifdef ident	/* Ako je identifikator definisan.	*/
#ifndef ident	/* Ako identifikator nije definisan.	*/
#elif izraz	/* Inače, ako je konstantni izraz #0.	*/
#else	/* Inače.	*/
#endif	/* Kraj oblasti uslovnog prevodenja.	*/

U *izrazima* može da se koristi specijalan izraz **defined ident** ili **defined (ident)**, koji ima vrednost 1L ako je *ident* definisan, odnosno 0L ako nije.

1.3.8 Bibliotečke funkcije

U sastavu jezika C definisan je veći broj bibliotečkih funkcija za često korišćene elementarne obrade. Sve te funkcije su podeljene u izvestan broj grupa prema srodnosti. Prototipovi funkcija iz svake grupe i definicije pratećih simboličkih konstanti nalaze se u odgovarajućim zaglavljima. Ta zaglavja su tekstualne datoteke koje treba staviti na raspolaganje prevodiocu odgovarajućim direktivama `#include`.

Najvažnija od svih je grupa za obradu datoteka (`<stdio.h>`). U njoj se nalaze funkcije za otvaranje i zatvaranje datoteka, za obradu tekstualnih datoteka sa ili bez ulazno/izlazne konverzije i za obradu binarnih datoteka. Tu se nalaze i funkcije za interne binarno-tekstualne konverzije podataka.

Druga po važnosti je grupa uslužnih funkcija (`<stdlib.h>`) u kojoj se nalaze, pored ostalog, funkcije za stvaranje i uništavanje dinamičkih podataka.

Od ostalih grupa interesantno je napomenuti matematičke funkcije (`<math.h>`), obradu niski (`<string.h>`) i ispitivanje znakova (`<ctype.h>`).

2 Proširenja jezika C

Programski jezik C++ sadrži veliki broj novih elemenata. Ono što posebno obeležava jezik C++ je uvodenje klasa sa svim pratećim konceptima. Postoji, međutim, i izvestan broj manjih novina u odnosu na jezik C. Deo od njih predstavlja neophodne izmene radi otklanjanja nekih nekonzistentnosti u jeziku C, dok druge uvode neke nove mogućnosti.

Ovo poglavlje je posvećeno upravo tim „manjim“ proširenjima. Nijedno od njih još ne čini jezik C++ objektno orijentisanim jezikom.

2.1 Elementi jezika

2.1.1 Komentari

Pored komentara stavljenih između simbola `/*` i `*/`, u jeziku C++ mogu da se koriste i komentari koji počinju simbolom `//`. Takvi komentari se završavaju na kraju reda bez posebnog označavanja završetka. Unutar komentara započetih simbolom `//` simboli `/*` i `*/` nemaju nikakav službeni značaj, već se uzimaju kao obični parovi znakova. Slično tome, unutar komentara započetih simbolom `/*` simbol `//` se uzima kao običan par znakova, bez službenog značaja.

Pošto se u praksi najčešće koriste kratki komentari i to na krajevima redova, novi način označavanja komentara udobniji je za korišćenje.

2.1.2 Službene reči

Uvođenje novih pojmljivačkih reči u jezik C++ iziskivalo je korišćenje novih službenih reči za njihovo obeležavanje. Tako je spisak službenih reči, koje su rezervisane reči i kao takve ne mogu da se koriste kao identifikatori u programima, produžen čak za 28 novih reči. Sve službene reči jezika C++ čine:

asm	auto	bool	break	case	catch
char	class	const	const_cast	continue	default
delete	do	double	dynamic_cast	else	enum
explicit	export	extern	false	float	for
friend	goto	if	inline	int	long
mutable	namespace	new	operator	or	private
protected	public	register	reinterpret_cast	return	short
signed	sizeof	static	static_cast	struct	switch

```
template  this      throw   true      try      typedef
typeid    typename  union   unsigned  using    virtual
void     volatile wchar_t while
```

Pored toga ne preporučuje se korišćenje identifikatora koji počinju sa dva znaka podvučeno (_). U standardnoj biblioteci funkcija jezika C++ koriste se takvi identifikatori. Treba izbegavati i identifikatore koji počinju jednim znakom podvučeno (_) jer se takvi identifikatori koriste u standardnoj biblioteci funkcija jezika C.

2.2 Tipovi podataka

2.2.1 Logički podaci

Za predstavljanje logičkih vrednosti uveden je tip **bool** sa dve moguće vrednosti: **false** za logičku neistinu i **true** za logičku istinu.

Tip **bool** pripada grupi celobrojnih tipova i postoji automatska konverzija između tipa **bool** i numeričkih tipova. Kada se koristi u izrazima logička vrednost **false** se pretvara u celobrojnu vrednost nula, a **true** u jedan. Ako se promenljivoj tipa **bool** dodeljuje numerička vrednost, vrednost nula se pretvara u **false**, a bilo koja nenulta vrednost u **true**.

Smatra se da relacijski operatori daju rezultate tipa **bool** i da su operandi i rezultati logičkih operatora tipa **bool**. Zbog postojanja automatske konverzije između logičkih i numeričkih tipova, na mestima na kojima se očekuju logičke vrednosti i dalje mogu da se pišu numerički izrazi kao što se pisalo u jeziku C.

2.2.2 Znakovne konstante

Znakovne konstante, na primer 'A', u jeziku C++ su tipa **char** dok u jeziku C su tipa **int**. To na određenim mestima dovodi do različitog tretiranja 'A' i 65, bez obzira što oni imaju iste brojčane vrednosti u slučaju korišćenja *ASCII* koda. U jeziku C te dve vrednosti ne mogu da se razlikuju ni pod kojim uslovima.

2.2.3 Simboličke konstante

U jeziku C++, dodavanjem modifikatora **const** na početku naredbi za definisanje podataka dobijaju se prave konstante, pod uslovom da su inicijalizatori konstantni izrazi. Zato tako uvedeni identifikatori predstavljaju simboličke konstante koje mogu da se koriste i na mestima na kojima se izričito traže konstante.

Ova mogućnost u velikoj meri smanjuje potrebu za korišćenjem direktive pretprocесора **#define**, odnosno naredbe **enum** za uvođenje simboličkih konstanti.

Evo primera za definisanje i korišćenje simboličke konstante:

```
const int DIM = 50;
double a[DIM];
```

2.2.4 Definisanje podataka

Deklarativne naredbe u jeziku C++ smatraju se izvršnim naredbama i mogu da se koriste bilo gde u programu gde su naredbe dozvoljene, a ne samo na početku pojedinih blokova. Doseg uvedenih identifikatora proteže se od mesta definisanja do kraja bloka unutar kojeg su definisani.

2.2.4 Definisanje podataka

Ovo u većini slučajeva omogućava da se definisanim podacima dodeljuju (početne) vrednosti odmah pri definisanju. Drugim rečima, nove podatke treba vesti u program tek kada su potrebni, i onda nema razloga da im se odmah ne dodeljuju početne vrednosti.

Vidče se kasnije da je u jeziku C++ uložen veliki napor u borbi protiv grešaka prouzrokovanih korišćenjem podataka kojima nikada nisu dodeljene vrednosti. Ova mera je jedan doprinos toj borbi.

U Standardu se čak otislo toliko daleko da je dozvoljeno u nekim složenim naredbama definisanje novih podataka i na mestima gde su u jeziku C izričito traženi izrazi.

Izrazom uslova u naredbi **if** može da se definiše jedna promenljiva numeričkog ili pokazivačkog tipa. Vrednost inicijalizatora (=0 ili ≠0) predstavlja uslov za grananje. Doseg uvedenog identifikatora je do kraja naredbe **if** i proteže se na obe naredbe sadržane u naredbi **if**. Skreće se pažnja da je u delu **else** vrednost te promenljive uvek 0. Na primer:

```
if (int k = i+j) {
    s += k;
} else {
    double k = 0.5; // GREŠKA: k već postoji.
}
int m = k; // GREŠKA: k više nije u dosegu.
```

Izrazom za uslov grananja u naredbi **switch** može da se definiše jedna promenljiva celobrojnog tipa. Vrednost inicijalizatora određuje oznaku **case** na koju se skače. Doseg uvedenog identifikatora je do kraja naredbe **switch**. Na primer:

```
switch (int k = i+j) {
    case 1: case 3: s *= k; break;
    case 2: case 4: s /= k; break;
    default:         s += k; break;
}
```

Izrazom za uslov u naredbi **while** može da se definiše jedna promenljiva numeričkog ili pokazivačkog tipa. Vrednost inicijalizatora (=0 ili ≠0) predstavlja uslov za nastavljanje ciklusa. Doseg uvedenog identifikatora je do kraja naredbe **while**. Na kraju svakog prolaska kroz ciklus promenljiva se uništava i iznova se definiše na početku narednog prolaska. Na primer:

```
while (int y = f(x)) { s += y; x += x; }
```

Izrazom za postavljanje početnih vrednosti u naredbi **for** mogu da se definišu nekoliko promenljivih međusobno jednakog numeričkog ili pokazivačkog tipa. Vrednosti inicijalizatora predstavljaju početne vrednosti tih promenljivih. Početne vrednosti se postavljaju samo jednom. Doseg uvedenih identifikatora je do kraja naredbe **for**. Na primer:

```
for (int k=0; k<n; k++) { ... }
if (k < n) { ... } // GREŠKA: k više nije u dosegu.
for (int k=n; k>0; k--) { ... }
```

Skreće se pažnja da je pre uvođenja Standarda mogućnost definisanja promenljivih u složenim naredbama postojala samo u naredbi **for**. Nažalost, prema ranijim pravilima doseg ovako uvedenih identifikatora protezao se do kraja bloka unutar kojeg se nalazi sama naredba **for**, a ne do završetka ciklusa. Prema tim pravilima druga naredba je ispravna, jer je identifikator **k** još u dosegu, ali zato je treća naredba pogrešna.

Još postoje prevodioci za jezik C++ koji ne uvažavaju Standard.

2.2.5 Definisanje tipova

Identifikatori nabrajanja, struktura i unija u jeziku C++ mogu da se koriste kao identifikatori tipa bez službenih reči `enum`, `struct`, odnosno `union`, pod uslovom da su jedinstveni u svojim dosezima. Ovo može da se shvati tako kao da se za te identifikatore automatski izvršava i naredba `typedef` čime se proglašavaju identifikatorima tipa.

Ukoliko u datom dosegu s istim identifikatorom istovremeno postoji neki podatak, nabrajanje, struktura i/ili unija, identifikator sâm označava podatak. Za označavanje nabrajanja, strukture ili unije potrebno je tada koristiti službenu reč `enum`, `struct` ili `union` ispred identifikatora.

Evo nekoliko primera korišćenja identifikatora nabrajanja, strukture i unije kao identifikatora tipa:

```
enum Logical {F, T};
Logical kraj, dalje;
struct Elek {
    int broj;
    Elek* sled;
};
Elek elem;
elem.broj = 55; elem.sled = NULL;
union alfa { int a, b, c; };
int alfa;
struct alfa { double a, b, c; };
union alfa g, h;           // Unija alfa.
alfa = 55;                 // Podatak alfa.
struct alfa z = {1, 2, 3};  // Struktura alfa.
```

U prvoj grupi naredbi definiše se nabrajanje `Logical` za simboličko predstavljanje logičkih konstanti. Posle toga definišu se dve promenljive tog tipa.

Druга grupa naredbi počinje definisanjem strukture `Elek` za predstavljanje elemenata jednostruko lančane linearne liste. Već u toku te naredbe identifikator `Elek` može da se koristi kao identifikator tipa za definisanje člana za pokazivanje na naredni element liste.

U trećoj grupi identifikator `alfa` koristi se u više svrha. Zbog toga samo `alfa` predstavlja celobrojnu promenljivu, a u ostalim slučajevima koristi se i odgovarajuća službena reč za označavanje svrhe korišćenja identifikatora.

2.2.6 Nabrajanja

Svako nabrajanje u jeziku C++ je celobrojan tip podataka koji definiše niz simboličkih konstanti. Sva nabrajanja se smatraju različitim tipovima kako međusobno, tako i u odnosu na osnovne celobrojne tipove `char`, `int`, `short` i `long`.

Za podatke tipova nabrajanja nije definisana nijedna operacija osim dodele vrednosti tipa nabrajanja promenljivoj tipa istog nabrajanja. Naravno, eksplicitna konverzija tipa vrednosti iz proizvoljnog celobrojnog tipa u tip nabrojane promenljive je dozvoljena. Greška je ako je dodeljivana vrednost izvan opsega vrednosti određenog nabrajanja.

Ako nabrajanje sadrži samo pozitivne konstante opseg vrednosti nabrajanja je od 0 do $2^k - 1$, a ako nabrajanje sadrži i negativne konstante opseg vrednosti nabrajanja je od -2^k do $2^k - 1$, gde je k najmanji ceo broj takav da nijedna od konstanti u nabrajanju nije izvan navedenog opsega.

2.2.6 Nabrajanja

Podaci tipa nabrajanja mogu da se koriste u aritmetičkim i relacijskim izrazima. Tada se vrednost tipa nabrajanja automatski pretvoriti u tip `int` pre izračunavanja.

Evo primera za korišćenje nabrajanja:

```
enum Dani { PO=1, UT, SR, CE, PE, SU, NE };
Dani dan = SR;
dan++;
// GREŠKA: dodela vrednosti tipa int.
dan = (Dani) (dan + 1);           // Eksplicitna konverzija tipa.
if (dan < NE) { ... }
```

Prvom naredbom definiše se nabrajanje `Dani` za označavanje dana u nedelji. Vrednosti definisanih konstanti su 1, 2, ..., 7. Drugom naredbom definiše se promenljiva `dan` tipa `Dani` i dodeljuje joj se početna vrednost `SR` (3). Treća naredba nije dozvoljena jer je rezultat operatora `++` tipa `int`, a to se razlikuje od tipa `Dani`. Četvrta naredba pokazuje dozvoljeni način promene vrednosti promenljive `dan` iz jednog u naredni dan. Na kraju, u poslednjoj naredbi vrednosti promenljive `dan` i konstante `NE` prvo se pretvore u tip `int` i tek posle toga se upoređuju.

2.2.7 Uvek promenljiva polja u strukturama

Dodavanjem modifikatora `mutable` na početku definicije polja strukture postiže se to da će vrednost tog polja moći da se promeni čak i unutar nepromenljivih podataka tipa te strukture.

Na primer:

```
struct Alfa {
    int a;
    mutable int b;           // Uvek može da se promeni.
};

main () {
    Alfa p;
    const Alfa q={1,2};     // Promenljiva struktura.
    p.a = 1;                // Nepromenljiva struktura.
    p.b = 2;                // Menjanje polja promenljivog podatka.
    q.a = 3;
    q.b = 4;                // GREŠKA: Menjanje nepromenljivog podatka.
                           // Može čak i u nepromenljivom podatku.
```

2.2.8 Bezimene unije

Unije u jeziku C++ mogu da se definišu bez navođenja identifikatora iza reči `union`. Tačke unije se nazivaju bezimene unije i predstavljaju jedan bezimeni podatak (unije s imenima su tipovi!). Taj podatak u raznim trenucima može da sadrži podatke različitih tipova.

Identifikatori članova imaju datotečki ili blokovski doseg, a ne struktturni kao kod unija s imenima. Članovi bezimennih unija koriste se kao i obične promenljive, a ne na način kako je to uobičajeno za strukture i unije.

Evo primera za korišćenje bezimene unije:

```
union { int i; double d; char* c; };
i = 55;
d = 123.456;
c = "Zdravo!";
```

Unija za koju je definisan bar jedan podatak ili pokazivač ne smatra se bezimenom unijom iako nema ime. To je slučaj kada se u naredbama za definisanje promenljivih umesto identifikatora ranije definisane unije navodi opis cele unije. Članovima takve unije ne može da se pristupa samo pomoću identifikatora člana, već mora da se navede i podatak. Na primer:

```
union {int i; char* pc;} a, *pa = &a; // Podatak a i pokazivač pa.
i = 5;                                // GREŠKA: mora za konkretni podatak.
a.i = 5;                               // U redu.
pa->i = 5;                            // U redu.
```

2.2.9 Pokazivači

Pokazivač na podatke poznatih tipova u jeziku C++ ne može da se dodeli vrednost generičkog pokazivača (`void*`). Naravno, uz eksplisitnu konverziju tipa može i to da se uradi. Na primer:

```
int a;
void* gp = &a;                      // U redu.
int* pi = gp;                        // GREŠKA: ne može generički pokazivač.
int* pj = (int*)gp;                  // Ovako može na odgovornost programera.
```

2.2.10 Upućivači

Upućivač u jeziku C++ je alternativno ime za neki podatak. Mada formalno izgledaju kao i drugi podaci upućivači to nisu. Oni samo upućuju na podatke kojima su pridruženi.

Upućivači ne zauzimaju prostor u memoriji, ne može da se dobija njihova adresa, ne postoje pokazivači na upućivače niti nizovi upućivača.

Upućivači moraju da se inicijalizuju prilikom definisanja (stvaranja) nekim podatkom koji se nalazi u memoriji (*vrednost*) jer kasnije ne mogu da im se promene vrednosti. Naime, sve operacije nad upućivačima (uključujući i dodelje vrednosti) deluju na podatke kojima su pridruženi, a ne na upućivače same.

Upućivači su vrlo slični pokazivačima. Njihova ostvarenja su u obliku adresa podataka kojima su pridruženi. Međutim, svako pominjanje upućivača podrazumeva posredan pristup do upućivanog podatka. Kod pokazivača, da bi se pristupilo do pokazivanih podataka, potrebno je koristiti operator za indirektno adresiranje (*). Pored toga, pokazivači su pravi podaci, što znači da zauzimaju određeni prostor u memoriji i mogu da im se promene vrednosti.

2.2.10.1 Definisanje upućivača

Upućivači se definišu uobičajenim naredbama za definisanje promenljivih dodavanjem modifikatora `&` ispred identifikatora upućivača.

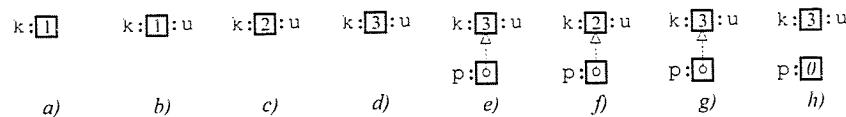
Slično pokazivačima, upućivači mogu da upućuju na promenljive, nepromenljive (`const`) i nepostojane (`volatile`) podatke. Upućivač na promenljive podatke može da se inicijalizuje samo promenljivim podatkom. Upućivač na nepromenljive podatke može da se inicijalizuje nepromenljivim i promenljivim podatkom (ali taj promenljiv podatak neće moći da se promeni pomoću tog upućivača). Upućivač na nepostojane podatke može da se inicijalizuje nepostojanim i promenljivim podatkom (prevodilac će preoprezno da rukuje tim podatkom, ali iz toga ne može da proizilazi nikakva šteta).

Evo primera za definisanje i korišćenje upućivača uz paralelno prikazivanje upotrebe pokazivača za izvođenje istih operacija kao sa upućivačem:

```
int k = 1;
int& u = k;
int x = u;
u = 2;
u++;
int* p = &u;
int y = *p;
*p = 2;
(*p)++;
p = 0;
const int& v = u;
v = 5; // GREŠKA: Ne može pomoći v.
```

// u i k predstavljaju isti int.
// x = 1;
// k = 2;
// k se povećava za 1.
// p pokazuje na k.
// y = 3;
// k = 2;
// k se povećava za 1.
// Promena vrednosti pokazivača.
// v upućuje na k.

U prvoj naredbi definiše se promenljiva `k` koja zauzima prostor u memoriji i ima početnu vrednost 1 (slika 2.1.a). U drugoj naredbi definiše se upućivač `u` koji nije podatak, ne zauzima prostor u memoriji, već se pridružuje kao drugo ime već postojećoj promenljivoj `k` (slika 2.1.b). Zbog toga, svako pominjanje upućivača u preostalim naredbama odnosi se na tu promenljivu `k`. Trećom naredbom se promenljiva `x`, u stvari, inicijalizuje kopijom sadržaja promenljive `k`. Četvrtom naredbom se promenljivoj `p` dodeljuje vrednost 2 (slika 2.1.c), a petom naredbom se povećava vrednost iste promenljive za 1 (slika 2.1.d).



Slika 2.1 – Upotreba upućivača i pokazivača

Na početku druge grupe naredbi definiše se pokazivač `p` koji se, u stvari, inicijalizuje adresom promenljive `k` (slika 2.1.e). Sledećim naredbama se pomoću tog pokazivača uzima vrednost promenljive `k`, dodeljuje se vrednost 2 toj promenljivoj (slika 2.1.f) i povećava se njena vrednost za 1 (slika 2.1.g).

Pošto su, za razliku od upućivača, pokazivači podaci, može im se promeniti vrednost. Tako je poslednjom naredbom u drugoj grupi pokazivaču `p` dodeljena vrednost 0 (slika 2.1.h).

U trećoj grupi je definisan upućivač na nepromenljive podatke u koji je inicijalizovan, posredstvom upućivača `u`, promenljivim podatkom `k`. Pomoću upućivača `v` ne može da se promeni vrednost, inače promenljivog, podatka `k`.

Upućivačkim tipovima mogu da se naredbom `typedef` pridružuju imena na isti način kao i ostalim tipovima podataka.

Evo primera kojim se prvo definise upućivački tip `U_double`, a kasnije i dva upućivača (`ua` i `ub`) tog tipa:

```
typedef double U_double;           // Tip za upućivače na double.
double a, b;
U_double ua = a, ub = b;          // Dva upućivača na double.
ua = ub = 5;                      // a = b = 5
```

2.2.10.2 Upućivači i funkcije

Upućivači mogu da budu parametri i vrednosti funkcija.

Kada je parametar funkcije upućivač, odgovarajući argument mora biti podatak (*lvrednost*) tipa upućivanih podataka. Nikakva automatska konverzija tipa ne može da se primeni, jer rezultat konverzije tipa nije *lvrednost*.

Upućivački parametar prilikom pozivanja funkcije inicijalizuje se podatkom koji je argument funkcije, pa on postaje alternativno ime za podatak koji se nalazi izvan funkcije. Zato, promene vrednosti parametra predstavljaju promene vrednosti odgovarajućeg argumenta. Drugim rečima, upućivači omogućuju stvaranje bočnih efekata funkcija. Naravno, za to argument mora da bude promenljiva (ne `const`) *lvrednost*.

Inicijalizacija upućivačkog parametra ne dovodi do kopiranja argumenta u parametar, kao što je to slučaj kod običnih parametara. Zbog toga se preporučuje upotreba upućivača i za parametre složenih tipova (na primer struktura), koji se inače prenose pomoću vrednosti. Time se štede i memoriski prostor i vreme. Ako se upućivač koristi samo radi uštede, parametar treba deklarisati (dodavanjem modifikatora `const`) kao upućivač na nepromenljive podatke. Argument prilikom pozivanja tada može da bude bilo nepromenljiv bilo promenljiv podatak.

Dodavanjem modifikatora `volatile` u deklaraciju upućivačkog parametra, funkcija će biti prevedena tako da bude pripravna na eventualne neočekivane promene vrednosti argumenta. Argument tada može da bude nepostojan ili promenljiv podatak.

Navodenjem oba modifikatora `const` i `volatile` u deklaraciji upućivačkog parametra se označava da funkcija ne menja vrednost parametra, ali da je pripravna na neočekivane promene vrednosti tog parametra. Argument može biti podatak bilo koje vrste: promenljiv, nepromenljiv ili nepostojan.

Evo primera funkcije čiji je parametar upućivač:

```
void povecaj (int& a) { a++; } // Parametar je upućivač.
int x = 1;
povecaj (x);                // x je sada 2.
povecaj (x+1);              // GREŠKA: x+1 nije lvrednost.
```

Isti rezultat pomoću pokazivača može da se dobije na sledeći način:

```
void povecaj (int* a) { (*a)++; } // Parametar je pokazivač.
int x = 1;
povecaj (&x);                // x je sada 2.
```

Skreće se pažnja na to koliko je rad s upućivačima prirodniji i lepši. Naravno, nije to glavni razlog za uvođenje upućivača.

Upućivač može da bude vrednost funkcije. Tada je vrednost funkcije podatak odgovarajućeg tipa i predstavlja *lvrednost*. Na rezultat funkcije tada mogu da se primene i operatori koji zahtevaju *lvrednosti* kao operande.

Skreće se pažnja da kod funkcije čija vrednost nije upućivač, rezultat je kopija vrednosti izraza u naredbi `return`. Ako je taj izraz samo jedna promenljiva, tada je to kopija sadržaja te promenljive, a ne promenljiva sama koja bi mogla dalje da se menja.

Vrednost funkcije ne sme da bude upućivač na lokalni prolazan podatak, jer će taj podatak biti uništen prilikom napuštanja funkcije, tj. pre nego što rezultat funkcije može da se iskoristi.

2.2.10.2 Upućivači i funkcije

Evo primera funkcije čija je vrednost upućivač na jedan od parametara:

```
int& max (int& p, int& q)           // Rezultat je upućivač.
{ return p>q ? p : q; }
int a = 2, b = 5;
max (a, b) ++;                      // o je sada 6.
```

Ako se vrednost funkcije ne deklariše kao upućivač rezultat je kopija vrednosti jednog od parametara, što nije *lvrednost*:

```
int max (int p, int q)           // Rezultat nije upućivač.
{ return p>q ? p : q; }
int a = 2, b = 5;
max (a, b) ++;                  // GREŠKA: max(a,b) nije lvrednost.
```

2.3 Operatori

Tablica 2.1 prikazuje zbirni pregled svih operatora u jeziku C++.

Prioritet	Broj operanada	Smer grupisanja	Operatori
19	1,2	→	::
18	2	→	[] () . ->
17	1		a++ a-- static_cast reinterpret_cast const_cast dynamic_cast typeid
16	1	←	! ~ ++a --a + - * & (tip) sizeof new delete
15	2	→	. * ->*
14	2	→	* / %
13	2	→	+ -
12	2	→	<< >>
11	2	→	< <= > >=
10	2	→	== !=
9	2	→	&
8	2	→	^
7	2	→	
6	2	→	&&
5	2	→	
4	3		? :
3	2	←	= += -= *= /= %= ^= = <<= >>=
2	1		throw
1	2	→	,

Tablica 2.1 – Osobine operatora u jeziku C++

Neki od novih operatora obrađeni su u narednim odeljcima ovog poglavlja, a preostali u kasnijim poglavljima.

Skreće se pažnja na to da postoje operatori za koje nije naveden smer grupisanja pošto ili ne mogu da se navedu jedan za drugim više takvih operatora, su ili pravila njihovog pisanja takva da je redosled izračunavanja jasan i bez smera grupisanja.

Pored toga, treba uočiti da su razdvojene prefiksne i postfiksne varijante operatora `++` i `--`. Postfiksne varijante imaju za jedan viši nivo prioriteta od prefiksne varijante.

2.3.1 Vrednost izraza kao ivrednost

Pored operatora za posredan pristup (`*`) i indeksiranje (`[]`), u jeziku C++ još nekoliko operatora daju rezultat koji je *ivrednost*.

Rezultat prefiksног oblika operatora `++` i `--` je *ivrednost* i predstavlja podatak koji je naveden kao operand, s izmenjenim sadržajem. Rezultat postfiksног oblika tih operatora nije *ivrednost*. Na primer:

```
++ ++k      // ++(++)k - k se povećava za 2.
k++ ++
// GREŠKA: (k++)++, a k++ nije ivrednost.
(--k) *=5   // k=(k-1)*5
```

Rezultat svakog operatora za dodelu vrednosti (`=, +=, -=, *=, /=, %=, &=, |=, ^=, <<= i <<=`) je *ivrednost* i predstavlja podatak koji je naveden kao prvi operand, s izmenjenim sadržajem. Na primer:

```
(d *= e) += f // d=d*e, d=d+f - d=(d*e)+f - e se ne menja, ali ...
d *= e += f // d=(e+f) - e=e+f, d=d*e - menja se i e.
(x = y) = 5   // x=y, x=5 - y se ne menja, ali ...
x = y = 5    // x=(y=5) - y=5, x=5 - menja se i y.
++ (p /= q)  // p=p/q, p=p+1 - p=p/q+1
++ p /= q    // (++p)/=q - p=p+1, p=p/q - p=(p+1)/q
```

Vrednost uslovnog izraza (`? :`) u jeziku C++ je *ivrednost* ako i samo ako su drugi i treći operand međusobno jednakih tipova i ako su oba *ivrednosti*. To omogućava da se, na primer, uslovnim izrazom odabere podatak (promenljiva) kome će biti dodeljena data vrednost. Vrednost uslovnog izraza u jeziku C nikada nije *ivrednost*. Na primer:

```
long x = 5;
int a, b;
x ? a : b = 1;           // x ? a=1 : b=1;
x ? x : b = 1;           // GREŠKA: x i b nisu istog tipa.
x ? 2 : b = 1;           // GREŠKA: 2 nije ivrednost.
```

2.3.2 Konverzija tipa

Operator za konverziju tipa (`tip`) iz jezika C ima nekoliko uloga. Neke od njih su bezopasne i rezultati ne zavise od računara na kome se izvršava program, dok su druge opasne ako se koriste neprimereno i rezultati mogu da budu različiti na različitim računarima.

Standardom za jezik C++ za pojedine uloge operatora konverzije tipa uvedeni su zasebni operatori. Opšti oblik za konverziju tipa pomoću tih operatora je:

```
static_cast < naziv_tipa > ( izraz )          ili
reinterpret_cast < naziv_tipa > ( izraz )        ili
const_cast   < naziv_tipa > ( izraz )
```

Naziv tipa označava tip u koji se pretvara vrednost *izraza*.

Operator `static_cast` je predviđen za konverzije između tipova podataka za koje je postupak konverzije jasno definisan i ishod konverzije, u najvećem broju slučajeva, ne

2.3.2 Konverzija tipa

zavisi od računara na kome se program izvršava. Tu spadaju konverzije između numeričkih tipova podataka (uključujući i nabranja – `enum`) i konverzije između pokazivača proizvoljnih tipova i generičkog pokazivača (tipa `void*`). Pored toga i nestandardne konverzije (koje definiše programer) spadaju u ovu grupu (definisanje konverzija za ne-standardne tipove obradeno je u odeljcima 3.4.4.3 i 4.3). Većina konverzija iz ove grupe primenjuje se i automatski. Eksplisitna konverzija je neophodna samo za pretvaranje generičkih pokazivača u pokazivače na podatke poznatih tipova i za pretvaranje numeričkih vrednosti u nabranja.

Evo primera za upotrebu operatora `static_cast`:

```
int a = static_cast<int> (5.2);
void* p = &a;
int* q = static_cast<int*>(p);
```

Prvom naredbom se podatak tipa `double` pretvara u podatak tipa `int` (ova eksplisitna konverzija nije neophodna, mada, ako se ne napiše neki prevodioci daju opomenu zbog mogućeg gubitka tačnosti). Drugom naredbom se pokazivač na ceo broj (`int*`) pretvara u generički pokazivač (moglo je da se napiše i eksplisitna konverzija `static_cast<void*>(&a)`). U poslednjoj naredbi eksplisitna konverzija je neophodna jer se vrednost generičkog pokazivača pretvara u pokazivač na cele brojeve. Ovo je jedina konverzija koja s ovim operatom može da bude opasna jer nije sigurno da taj pokazivač stvarno pokazuje na ceo broj.

Operator `reinterpret_cast` je predviđen za konverzije između tipova među kojima ne postoji nikakva logička veza. U ovu grupu spadaju konverzija celih brojeva u pokazivače i obrnuto, kao i konverzije između pokazivača i upućivača različitih tipova (osim `void*`). U ovim konverzijama, u stvari nema nikakvog preračunavanja vrednosti, već se odredena kombinacija bitova tumači na drugi način. Ovo su vrlo opasne radnje i malo je verovatno da se bilo koja od tih konverzija na isti način ponaša na različitim računarima. Srećom, za ovim konverzijama potreba se ukazuje samo pri krajnje profesionalnom pisanju sistemskog softvera, kao što su operativni sistemi.

Evo primera za upotrebu operatora `reinterpret_cast`:

```
int a = 1155;
short* p = reinterpret_cast<short*>(a);
int b = reinterpret_cast<int>(p);
float* q = reinterpret_cast<float*>(&a);
float& c = reinterpret_cast<float&>(a);
```

U drugoj naredbi celobrojna vrednost se stavlja u pokazivač, pri čemu tip pokazivanih podataka nije važan. Dobijena vrednost se u trećoj naredbi konverte nazad u tip `int`. Standard ne propisuje ishod pojedinačnih konverzija celog broja u pokazivač i obrnuto, ali zahteva da posle primene dve takve konverzije u suprotnim smerovima krajnji rezultat bude jednak početnoj vrednosti (dakle, promenljiva `b` treba da bude 1155).

U četvrtoj naredbi adresa promenljive `a` (tip `int*`) se dodeljuje pokazivaču `q` tipa `float*`, a u petoj naredbi istoj promenljivoj a se pridružuje kao drugo ime upućivača `c` tipa `float&`. Time se postiže samo da se pomoću pokazivača `q`, odnosno upućivača `c` nepromenjena kombinacija bitova tumači prema šemsi smeštanja realnih brojeva jednostrukice tačnosti. Ispisivanjem vrednosti `*q`, odnosno `c` na nekom računaru se ispisala vrednost `1.6185e-42`. Na nekom drugom računaru bi se možda dobila neka druga vrednost.

Operator `const_cast` je predviđen za skidanje modifikatora `const` ili `volatile` sa nekog tipa ili za njihovo dodavanje na neki tip. Skidanje tih modifikatora je opasna radnja jer može dovesti do nepoželjnih efekata. Na primer, omogućuje da se promeni vrednost nepromenljivog podatka ili da se dobije kôd koji nije primeren nepostojanosti podatka. Dodavanje tih modifikatora nije opasno, jer kao posledica neće moći posle da se promeni vrednost inače promenljivog podatka ili će se dobiti kôd koji previše oprezno koristi inače postojani podatak. Zbog toga za dodavanje modifikatora `const` i `volatile` nije neophodno koristiti eksplisitnu konverziju tipa.

Evo primera za upotrebu operatora `const_cast`:

```
const int a = 5;
int* p = const_cast<int*>(&a);
*p = 8; // PROBLEM: Menja se vrednost nepromenljivog podatka a!
double b = 1.23;
const double* c = &b;
c = 3.21; // GREŠKA: c je upućivač na nepromenljive podatke.
```

U drugoj naredbi pokazivaču `p` na promenljive celobrojne podatke (tip `int*`) dodeljuje se adresa nepromenljivog celog brojnog podatka `a` (tip `const int*`). Posle toga treća naredba će promeniti vrednost podatka `a` za koji je rečeno da ne sme da se menja!

Upućivač `c` na nepromenljive podatke u prethoslednjoj naredbi može bez problema da se inicijalizuje promenljivim podatkom `b`. Radi naglašavanja mogla je da se piše i eksplisitna konverzija u obliku `const_cast<const double*>(b)`. Pomoću upućivača `c` ne može da se promeni vrednost, inače promenljivog, podatka `b`.

Novouvedeni operatori za konverziju tipa namerno su birani da budu nezgrapni iz dva razloga. Prvo, da bi se što lakše uočila mesta njihovog korišćenja u tekstu programa pošto predstavljaju potencijalne izvore problema. Drugo, da bi se programeri odvraćali od njihove upotrebe, jer se smatra da je potreba za eksplisitnom konverzijom tipa znak nedovoljno pažljivo osmišljenog programa.

Upotreba operatora za konverziju tipa (tip) iz jezika C u sve tri razmatrane svrhe konverzije ne naglašava dovoljno nameru programera. Pored toga, taj operator se teže uočava u tekstu programa.

2.3.3 Ulaz i izlaz podataka

Izvorišta podataka odakle program može da čita potrebne podatke u jeziku C++ nazivaju se **ulazni tokovi**. Kao primjeri ulaznih tokova mogu da se navedu tastatura, datoteka na disku, modem za vezu s telekomunikacionom linijom.

Odredišta podataka na koja program može da pošalje svoje rezultate nazivaju se **izlazni tokovi**. Primeri izlaznih tokova su ekran monitora, štampač, datoteka na disku, modem za vezu s telekomunikacionom linijom.

Na početku izvršavanja programa automatski se stvaraju sledeće dve globalne promenljive tipa tokova:

- `cin` – tekstualni ulazni tok koji predstavlja glavni ulaz računara, koji je obično tastatura,
- `cout` – tekstualni izlazni tok koji predstavlja glavni izlaz računara, koji je obično ekran monitora.

2.3.3 Ulaz i izlaz podataka

U narednim odeljcima objašnjeni su osnovi čitanja i pisanja podataka pomoću tokova `cin` i `cout`. Detaljno objašnjenje rada s tokovima je dato u poglavljju 9.

Tokovi `cin` i `cout` mogu da se koriste i kao logičke veličine. Tada imaju vrednost logičke istine (`true`) ako nije bilo greške za vreme prenosa, a vrednost logičke neistine (`false`) ako je bilo neke greške. Posebno, tok `cin` imaće vrednost `false` i ako se za vreme čitanja stiglo do kraja toka podataka. Kraj datoteke preko tastature simulira se unošenjem znaka `ctrl-D` pod operativnim sistemom *UNIX*, odnosno znaka `ctrl-Z` pod operativnim sistemom *MS-DOS* i *MS-Windows*.

2.3.3.1 Operatori za čitanje i pisanje s konverzijom

Kao zamena funkcijama `scanf` i `printf` za čitanje i pisanje podataka uz ulaznu i izlaznu konverziju podataka u jeziku C++ dodeljena su i nova značenja operatorima `>>` i `<<`. Značenje tih operatora nije izmenjeno u slučajevima kada su im oba operanda celobrojnih tipova. I dalje znaće pomeranje prvog operanda za određeni broj binarnih mesta uлево, odnosno udesno.

Ako je prvi operand operatora `>>` tekstualni ulazni tok (u smislu koji je objašnjen u poglavljju 9) čita jedan podatak iz te datoteke. Pročitani podatak se smešta u drugi operand tog operatora uz primenu ulazne konverzije koja odgovara tipu drugog operanda. Slično tome, ako je prvi operand operatora `<<` tekstualni izlazni tok, upisuje se vrednost drugog operanda u tu datoteku. Pri tome, primenjuje se izlazna konverzija koja odgovara tipu drugog operanda.

Tip vrednosti izraza u oba slučaja je upućivač na tekstualni tok koji je prvi operand operatora `>>` ili `<<`. Ovo omogućava prenošenje više podataka pomoću jednog izraza. Na primer, izrazom `ut>>a>>b` čitaju se dva podatka iz ulaznog toka `ut` i smeštaju se redom u promenljive `a` i `b`.

Osim tumačenja operatora `>>` i `<<` za slučaj kada je prvi operand tok podataka, ostali njihovi parametri nisu izmenjeni. I u ovom slučaju imaju prioritet 12 i grupišu se sleva udesno. Ako se kao operandi navedu izrazi treba voditi računa o tome da logički operatori i operatori za dodelu vrednosti imaju niže prioritete od operatora `>>` i `<<`. Zbog toga je neophodna upotreba zagrade ako se u tim izrazima javljaju navedeni operatori.

Sve potrebne deklaracije za ovu primenu operatora `>>` i `<<`, kao i definicije promenljivih `cin` i `cout` nalaze se u standardnom zaglavljtu `<iostream>`. Iz razloga koji je objašnjen u odeljku 2.5.5, posle direktive `#include`, potrebno je još dodati naredbu

```
using namespace std;
```

Funkcije jezika C za čitanje i pisanje podataka, koje su opisane u zaglavljtu `<stdio.h>`, i dalje stoje na raspolaganju. Nije preporučljivo da se u istom programu koristi i jedan i drugi način čitanja i pisanja podataka.

Čitanje i pisanje podataka obavlja se po pravilima kako to, u jeziku C, rade funkcije `scanf` i `printf` kada se u opisima konverzija ne navedu nikakvi parametri. Dakle, primenjuje se jedna od konverzija `%d` (za tipove `int`, `short` i `long`), `%g` (za tipove `float`, `double` i `long double`), `%c` (za tip `char`), `%s` (za tip `char*`) ili `%p` (za tip `void*`). Skreće se pažnja na to da se i pri čitanju pojedinačnih znakova podrazumevano dobija prvi pročitani nebeli znak (što nije slučaj u jeziku C).

Evo primjera programa koji čita dva cela broja i ispiše njihov zbir:

```
#include <iostream>
using namespace std;
void main () {
    cout << "a,b? ";
    // Pisanje poruke pre čitanja.
    int a, b;
    cin >> a >> b;
    int c = a + b;
    cout << "a+b= " << c << '\n';
    // Pisanje rezultata.
}
```

2.3.3.2 Manipulatori za podešavanje parametara konverzije

Podešavanje parametara konverzije je moguće upotrebom specijalnih **manipulatora** kao desnih operandom operanada `>>` i `<<`. Ako se koristi neki od manipulatora s parametrom u prevodenje programa treba da bude uključeno i zaglavlje `<iomanip>`.

<code>skipws</code>	– uključivanje preskakanja vodećih belih znakova pri čitanju (podrazumevaju se).
<code>noskipws</code>	– isključivanje preskakanja vodećih belih znakova pri čitanju.
<code>ws</code>	– preskakanje belih znakova pri čitanju.
<code>endl</code>	– pisanje znaka ' <code>\n</code> ' i pražnjenje izlaznog bafera.
<code>setw(int š)</code>	– podešavanje širine polja na najmanje <code>š</code> znakova samo za naredni podatak (podrazumevano 1).
<code>right</code>	– podešavanje ravnjanja uz desnu ivicu izlaznog polja (podrazumevaju se).
<code>left</code>	– podešavanje ravnjanja uz levu ivicu izlaznog polja.
<code>setfill(int z)</code>	– zadavanje znaka <code>z</code> za popunjavanje izlaznog polja (podrazumevaju se razmak).
<code>boolalpha</code>	– uključivanje prenosa logičkih podataka u tekstualnom obliku.
<code>noboolalpha</code>	– isključivanje prenosa logičkih podataka u numeričkom obliku (podrazumevaju se).
<code>showpos</code>	– uključivanje pisanja predznaka + za nenegativne brojeve.
<code>noshowpos</code>	– isključivanje pisanja predznaka + za nenegativne brojeve (podrazumevaju se).
<code>dec</code>	– uključivanje decimalne konverzije celih brojeva (podrazumevaju se).
<code>oct</code>	– uključivanje oktalne konverzije celih brojeva.
<code>hex</code>	– uključivanje heksadecimalne konverzije celih brojeva.
<code>setbase(int b)</code>	– podešavanje baze brojevnog sistema za cele brojeve na <code>b</code> (= 0, 8, 10, 16). Nula pri čitanju označava da se baza brojevnog sistema određuje na osnovu oblika pročitanog broja, a pri pisanju decimalnu konverziju.
<code>showbase</code>	– uključivanje prikazivanja osnove brojevnog sistema (0 za oktalne, 0x za heksadecimalne) celih brojeva pri pisanju.

2.3.3.2 Manipulatori za podešavanje parametara konverzije

<code>noshowbase</code>	– isključivanje prikazivanja osnove brojevnog sistema celih brojeva pri pisanju (podrazumevaju se).
<code>showpoint</code>	– uključivanje pisanja decimalne tačke za realne brojeve bez razlomljenog dela.
<code>noshowpoint</code>	– isključivanje pisanja predznaka decimalne tačke za realne brojeve bez razlomljenog dela (podrazumevaju se).
<code>fixed</code>	– uključivanje pisanja realnih brojeva bez eksponenta (podrazumevaju se sa ili bez eksponenta zavisno od veličine broja).
<code>scientific</code>	– uključivanje pisanja realnih brojeva s eksponentom.
<code>setprecision(int t)</code>	– podešavanje tačnosti prikazivanja realnih brojeva sa <code>t</code> cifara iza decimalne tačke (fixed), odnosno sa <code>t</code> značajnih cifara (scientific). Podrazumevano se koriste 6 cifara.

Evo primera za upotrebu manipulatora:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main () {
    int i = -1; double r = 123.456789012345;
    cout << i << ' ' << oct << i << ' ' << hex << i << endl;
    cout << r << ' ' << setprecision(3) << r
        << ' ' << setprecision(15) << r << endl;
    cout << setw(10) << i << setfill('*') << setw(25) << r << endl;
    cout << true << ' ' << boolalpha << true << endl;
}
```

Rezultat ovog programa je:

```
-1 37777777777 ffffffff
123.457 123 123.456789012345
ffffffffff*****123.456789012345
1 true
```

U prvom redu je ista celobrojna vrednost ispisana u tri moguća brojevna sistema. U drugom redu ista realna vrednost je ispisana sa podrazumevanom tačnošću i sa dve zadate tačnosti. Treći red pokazuje podešavanje širine izlaznog polja i znaka za popunjavanje. Poslednji red prikazuje dva oblika ispisivanja logičkih vrednosti.

2.3.3.3 Čitanje i pisanje znakova bez konverzije

Za efikasno čitanje i pisanje znakovnih podataka u jeziku C++ postoje funkcije koji se pozivaju dole navedenim izrazima (zašto ti izrazi treba da izgledaju baš tako biće jasno posle sledećeg poglavlja o klasama).

`cin.get()`

Ovim izrazom čita se sledeći znak, uključujući i beline, s glavnog ulaza. Vrednost izraza (tip `int`) je kôd pročitanog znaka ili EOF ako je pročitan znak za kraj toka podataka (na primer: `while((zn=cin.get())!=EOF){...}`).

```
cin.get(znak)
```

Ovim izrazom čita se sledeći znak, uključujući i beline, s glavnog ulaza i smešta se u parametar *znak* (tipa `char&`). Vrednost izraza je `cin` u izmenjenom stanju. To omogućava čitanje više znakova jednim izrazom (na primer, izrazom `cin.get(zn1).get(zn2)` se čitaju dva uzastopna znaka).

```
cin.getline(niz,max)
```

Ovim izrazom čitaju se znakovi, uključujući i beline, s glavnog ulaza do kraja reda, ali najviše *max-1* znakova. Pročitani znakovi se smeštaju u *niz* (tip `char*`), a iza poslednjeg znaka se dodaje znak '`\0`'. Vrednost izraza je `cin` u izmenjenom stanju. To omogućava čitanje više redova jednim izrazom (na primer, izrazom `cin.getline(niz1, 100).getline(niz2, 100)` se čitaju dva uzastopna reda teksta).

```
cout.put(znak)
```

Ovim izrazom se *znak* ispisuje na glavnom izlazu. Vrednost izraza je `cout` u izmenjenom stanju. To omogućava pisanje više znakova jednim izrazom (na primer: `cout.put('A').put('B')`).

Evo primera kojim se prepisuje tekst s glavnog ulaza na glavni izlaz sve dok se ne pročita znak za kraj toka podataka:

```
while (cin.get(zn)) cout.put(zn);
```

2.3.4 Dinamička dodela memorije

Za potrebe dodele memorije u dinamičkoj zoni memorije u vreme izvršavanja programa, u jeziku C++ uvedeni su operatori `new` i `delete`.

Funkcije `malloc`, `calloc`, `realloc` i `free`, koje se u tu svrhu koriste u jeziku C, postoje i dalje. Njih, međutim, ne treba koristiti iz sledećih razloga:

- Veličina dodeljenog prostora pomoću operatora `new` automatski se određuje na osnovu veličine stvaranog podatka (kod funkcija `malloc`, `calloc` i `realloc` veličina mora da se navede, obično primenom operatora `sizeof`).
- Tip vrednosti operatora `new` je uvek pokazivač na upravo stvoreni podatak (tip vrednosti funkcija `malloc`, `calloc` i `realloc` je uvek `void*`).
- Operatorom `new` moguće je postaviti početne vrednosti stvaranim podacima po želji (kod funkcija `malloc` i `realloc` stvoreni podaci imaju neki slučajni sadržaj, a kod funkcije `calloc` podaci se inicijalizuju nulama).
- Operatorom `new` moguće je stvarati i nizove podataka.

Operatori `new` i `delete` su unarni prefiksni operatori, imaju prioritet 15 i grupišu se zdesna uleva.

Opšti oblik izraza za dodelu memorije operatorom `new` je:

```
new naziv_tipa ( početna_vrednost )
```

Naziv tipa je identifikator osnovnog ili izведенog tipa s eventualnim modifikatorima * ako se prostor dodeljuje pokazivaču, ili [*dužina*] ako se prostor dodeljuje nizu. *Dužina* predstavlja broj elemenata niza kojima treba dodeliti memoriju. Modifikatori se pišu iza identifikatora tipa.

2.3.4 Dinamička dodela memorije

U slučaju višedimenzionalnih nizova dužine za sve dimenzijske osim za prvu moraju da budu konstantni izrazi. Dužina za prvu ili jedinu dimenziju može da bude promenljiv izraz u kome sve promenljive imaju definisane vrednosti u momentu izvršavanja operatora `new`.

Početna vrednost je proizvoljan izraz odgovarajućeg tipa čija je vrednost koristi za inicijalizaciju stvorenog podatka, kao i u naredbama za definisanje podataka. Mogu da se inicijalizuju samo pojedinačni podaci, a ne i nizovi. Ako se (*početna_vrednost*) izostavi, stvoren podatak će imati neku slučajnu početnu vrednost. U slučaju struktura početna vrednost ne može da se zadaje u obliku niza vrednosti unutar zagrade `{...}`. Taj oblik može da se koristi samo u naredbama za definisanje podataka.

Vrednost izraza operatorom `new` je pokazivač na dodeljeni memorijski prostor. Prema Standardu, ako dodela memorije ne uspe, greška se prijavljuje izuzetkom tipa `bad_alloc`. Ako se taj izuzetak ne obrađuje od strane programa, program se prekida (izuzeci su objašnjeni u pogлављu 6). Tip `bad_alloc` je definisan u zaglavju `<new>`.

Pre uvođenja Standarda u slučaju neuspešne dodelje memorije rezultat operatora `new` je bio pokazivač 0 (NULL).

Opšti oblik izraza za oslobođanje memorije u dinamičkoj zoni pomoću operatora `delete` je:

```
delete [] izraz
```

Izraz mora da je adresni izraz koji pokazuje na podatak u dinamičkoj zoni memorije kome je memorija ranije dodeljena pomoću operatora `new`. U slučaju da se radi o dinamičkom nizu vrednost izraza mora da pokazuje na početak tog niza (ne može da se oslobođa prostor dodeljen samo nekom elementu niza!).

Posledice su nepredvidljive ako operand operatora `delete` ne pokazuje na memorijski prostor koji je ranije dodeljivan operatorom `new`. Nula (NULL) kao operand operatora `delete` je bezopasna, tada se jednostavno ništa ne dešava.

Uglaste zagrade `[]` su potrebne samo ako se radi o oslobođanju memorije koja je dodeljena nizu. Skreće se pažnja na to da ne treba navesti broj elemenata tog niza niti stavljati veći broj parova zagrada u slučaju višedimenzionalnih nizova. Posledice su nepredvidljive ako se zagrade navode u slučaju oslobođanja memorije koja je dodeljivana pojedinačnom podatku, odnosno, ako se ne navode pri oslobođanju memorije koja je dodeljivana nizu.

Primenom operatora `delete` ne dobija se nikakav rezultat (tip `void`).

Evo primera za korišćenje operatora `new` i `delete`:

```
double* pd = new double (5.2);
cout << *pd << endl;
delete pd;

struct S { double a, b; };
S s = {1, 2};
S* ps = new S (s);
cout << ps->a << ' ' << ps->b << endl;
delete ps;

int n; cin >> n;
float* a = new float [n];
float z = 0;
for (int i=0; i<n; i++) { cin >> a[i]; z += a[i]; }
cout << z << endl;
delete [] a;
```

U prvoj grupi naredbi memorija je dodeljena jednom prostom podatku, a u drugoj složenom podatku (strukturi) koji se smatra pojedinačnim podatkom. Zbog toga u oba slučaja je mogao da se navede inicijalizator odgovarajućeg tipa.

U trećoj grupi naredbi memorija se dodeljuje nizu. Sadržaj niza ne može da se inicijalizuje prilikom dinamičke dodele memorije. Njima je jedino moguće naknadno dodeljivati vrednosti element po element.

Na kraju sve tri grupe naredbi memorija dodeljena dinamičkom podatku, kada taj podatak više nije potreban, oslobađa se odgovarajućim oblikom operatora **delete**.

2.4 Funkcije

2.4.1 Ugrađene funkcije

Za vrlo male funkcije pozivanje funkcije uz prenos argumenata na mesto gde se nalazi jedini primerak njenog prevoda i povratak iz funkcije uz vraćanje rezultata može da traje zнатно duže od izvršavanja samog sadržaja funkcije. Za takve funkcije je efikasnije da se prevod tela funkcije ugrađi u kôd po jednom na svakom mestu pozivanja. To, pored uštедa u vremenu, često znači i manji utrošak memorije.

Funkcije koje se ugrađuju na svakom mestu pozivanja nazivaju se i **ugrađene funkcije**.

Sugestija prevodiocu da se na mestima poziva neke funkcije umesto poziva ugrađi prevod tela funkcije, u jeziku C++ se iskazuje dodavanjem modifikatora **inline** na početku definisanja funkcije. Nije sigurno da će prevodilac stvarno uvažiti taj zahtev.

Ako se takva funkcija koristi u više datoteka nekog programskog sistema u svakoj datoteci mora da postoji potpuna definicija te funkcije. Sama deklaracija (prototip) funkcije nije dovoljna. Prevodiocu, da bi mogao telo funkcije da ugrađi u kôd, mora da stoji na raspolaganju izvorni tekst tela funkcije!

Radi ušteda truda za umnožavanje izvornog teksta definicija ugrađenih funkcija, a i radi obezbeđivanja istovetnosti tih definicija, predlaže se njihovo stavljanje u zaglavljene datoteke (tipa .h). Te datoteke se posle uključuju u prevod odvojenih programskih modula direktivom pretpresosora **#include**.

Ako prevodilac ne uvažava zahtev za ugradivanje funkcije u kôd praviće se prava funkcija s unutrašnjim povezivanjem (**static**). To znači da ako se takva funkcija koristi u više datoteka, u svakoj od tih datoteka postojaće po jedan primerak njenog prevoda. Jasno je da je to štetno, jer se troši više memorije, a ne dobija se u brzini rada. Većina prevodilaca će dati opomenu ako je nastala ovakva situacija.

Mana ugrađenih funkcija je što one izlazu javnosti detalje ostvarivanja pojedinih funkcija. Time se odaju „poslovne tajne“ korisniku biblioteke funkcija opšte namene u kojoj su neke ili većina funkcija ugrađene funkcije. U slučaju pravih funkcija korisniku je dovoljno stavljati na raspolaganje samo prototipove, a ne i cele definicije funkcija!

U jeziku C efekat ugrađenih funkcija može da se postiže makroima zadatih direktivama pretpresosora **#define**. Međutim, obrada makroa je prosta obrada teksta, bez mogućnosti provore slaganja tipova parametara i argumenata. Pored toga, izrazi kao argumenti makroa, posle razvijanja, mogu da se izračunavaju i više puta, što je fatalno ako ti izrazi daju i boćne efekte. To se nikada ne dešava u slučaju ugrađenih funkcija.

U jeziku C++ direktiva **#define** praktično više nije potrebna. Njena jedina preostala upotrebljiva vrednost je definisanje identifikatora za potrebe uslovnog prevodenja delova

2.4.1 Ugrađene funkcije

izvornog teksta. U samim programima u potpunosti mogu da je zamene ovde razmatrane ugrađene funkcije i činjenica da se modifikatorom **const** mogu da definišu prave simboličke konstante (videti odeljak 2.2.2).

Evo primera za ugrađenu funkciju i odgovarajuće direktive **#define** za definisanje makroa za nalaženje veće od dve zadate vrednosti:

```
inline int max (int i, int j) { return (i > j) ? i : j; }
#define max (i, j) ((i) > (j)) ? (i) : (j)
```

Parametri i vrednost funkcije su poznatih tipova (**int**), što nije slučaj kod makroa. Ako se argumenti po tipu ne poklapaju s tipovima parametara funkcije, ako je to moguće, primeně se automatske konverzije tipova argumenata u tipove parametara pre izvršavanja tela funkcije. Pravila su ista kao i prilikom dodele vrednosti. Ako konverzija tipa nije moguća prevodilac će prijaviti grešku. To sve nije moguće u slučaju makroa.

Ako su argumenti funkcije izrazi vrednosti tih izraza izračunavaju se pre izvršavanja tela funkcije. U slučaju makroa izrazi kao argumenti se uvršćuju na svakom mestu pojavljivanja odgovarajućih parametara. Tako dobijeni tekst se posle toga prevodi. Na primer, u slučaju izraza `max (k++, l++)` obe promenljive `k` i `l` povećavaju se samo jednom, ako se koristi ugrađena funkcija. Izraz koji se dobije razvijanjem makroa glasi: `((k++) > (l++)) ? (k++) : (l++)`, pa će jedna od promenljivih `k` i `l` biti dva puta povećavana!

2.4.2 Podrazumevane vrednosti parametara funkcija

Prilikom definisanja funkcija u jeziku C++ mogu da se navedu podrazumevane vrednosti za neke parametre. Te vrednosti koriste se kao početne vrednosti tih parametara u slučajevima kada u pozivima funkcija nedostaju odgovarajući argumenti.

Podrazumevane vrednosti navode se na isti način kao i početne vrednosti u naredbama za definisanje podataka. Po formi mogu da budu proizvoljni izrazi koji se iznova izračunavaju pri svakom pozivanju funkcija.

Ako se za neki parametar navodi podrazumevana vrednost to mora da se uradi i za sve parametre iza njega. Na primer, ako se za treći parametar funkcije sa pet parametara navede podrazumevana vrednost, mora se navesti podrazumevana vrednost i za četvrti i peti parametar.

Prilikom pozivanja takvih funkcija mogu da budu izostavljeni samo nekoliko poslednjih argumenata od onih koji odgovaraju parametrima s podrazumevanim vrednostima. Ne može, na primer, da se izostavi pretposlednji, a da se ne izostavi poslednji argument.

Dozvoljeno je da se za sve parametre neke funkcije predvide podrazumevane vrednosti. U tom slučaju, funkcija može da se poziva i bez argumenata.

Evo primera korišćenja funkcije s jednim podrazumevanim parametrom:

```
struct Tačka { double x, y; }; // Koordinatni početak.
const Tačka ORG = {0, 0};
#include <math.h>
double rastojanje (Tačka a, Tačka b = ORG)
    { return sqrt (pow (a.x-b.x, 2) + pow (a.y-b.y, 2)); }
#include <iostream>
using namespace std;
void main ()
{ Tačka p = {1, 1}, q = {-1, -1}; // Između dve tačke.
cout << rastojanje (p, q) << endl; // Od koordinatnog početka.
cout << rastojanje (p) << endl;
```

Funkcija rastojanje izračunava rastojanje između dve tačke u ravni. Za drugi parametar, kao podrazumevana vrednost, naveden je koordinatni početak. Dakle, u slučaju poziva s jednim argumentom dobija se rastojanje te tačke od koordinatnog početka.

2.4.3 Preklapanje imena funkcija

Često se javlja potreba da se u suštini ista radnja primeni na podatke različitih tipova. Za svaki tip podataka, naravno, mora da se napišu odvojene funkcije. Bilo bi nepraktično kada bi sve te funkcije morale da imaju različita imena. U jeziku C nije bilo moguće definisati dve funkcije s istim imenom. Mechanizam preklapanja imena funkcija u jeziku C++ omogućava da se srodnim funkcijama daju ista imena.

Pod preklapanjem imena funkcija podrazumeva se definisanje više funkcija s istim identifikatorima. Sve te funkcije moraju međusobno da se razlikuju po broju i/ili tipovima parametara na način koji obezbeđuje njihovu jednoznačnu identifikaciju. Što se tipova vrednosti tih funkcija tiče, oni mogu da budu i međusobno isti.

Razlike u parametrima moraju da budu takve da prevodilac, na osnovu broja i tipova argumenata prilikom pozivanja, može jednoznačno da odredi koju od istoimenih funkcija da pozove. Posebnu pažnju treba posvetiti slučajevima funkcija koje imaju parametre s podrazumevanim vrednostima. Identifikacija funkcija mora da je moguća na osnovu navedenih, a ne na osnovu mogućih argumenata!

Pravila za izbor funkcije na osnovu tipova argumenata su:

- traži se potpuno slaganje tipova argumenata s tipovima parametara i ako se nađe takva funkcija ona se koristi (u ovom smislu nula, **char** i **short** se potpuno slažu s tipom **int**, a **float** s tipom **double**);
- inače, traži se slaganje korišćenjem standardnih konverzija za osnovne tipove podataka i koristi se bilo koja funkcija koja na taj način odgovara;
- inače, traži se slaganje korišćenjem nestandardnih konverzija koje je definisao programer i, ako postoji jedinstven skup konverzija za izbor funkcije, koristi se tako odabранa funkcija (definisanje konverzija za nestandardne tipove obrađeno je u odeljku 4.4.1).

Naravno, ako na osnovu prethodnih pravila prevodilac ne bude u stanju da odabere neku od raspoloživih funkcija, prijaviće grešku.

Evo nekoliko primera korišćenja funkcija s preklapljenim imenima:

```
int max (int i, int j) { return (i > j) ? i : j; }
double max (double i, double j) { return (i > j) ? i : j; }
char max (char i, char j) { return (i > j) ? i : j; }

int a = max (1, 2);           // Poziva se max(int,int)
double b = max (1.0, 2.0);    // Poziva se max(double,double)
char c = max ('a', '3');     // Poziva se max(char,char)
double d = max (1, 2.0);     // Poziva se max(double,int)
int e = max (1, '3');        // Poziva se max(int,char)

void alfa (char i, int j=0); // S podrazumevanim parametrom.
void alfa (char i, float j=0); // PROBLEM: Šta znači alfa('a')?
alpha ('5', 55);
alpha ('5', 55.0f);
alpha ('5');
```

2.4.3 Preklapanje imena funkcija

U prvoj grupi naredbi definisane su tri funkcije s imenom **max** za nalaženje većeg od dva podatka tipa **int**, **double** i **char**. U drugoj grupi naredbi navedeni su primjeri pozivanja tih funkcija. U prva tri slučaja izbor funkcije je na osnovu potpunog slaganja tipova argumenata i parametara. U četvrtom slučaju drugi argument je tipa **double** i pošto postoji standardna konverzija iz tipa **int** prvog argumenta u tip **double** prvog parametra, odabere se funkcija s parametrima tipa **double**. U petom slučaju prvi argument je tipa **int** a tip drugog argumenta **char** potpuno se slaže s tipom **int**, pa se odabere funkcija s parametrima tipa **int**.

U trećoj grupi naredbi deklarisane su dve funkcije sa po jednim parametrom s podrazumevano vrednošću. Problem je što je obavezni prvi parametar kod obe funkcije istog tipa. Ako se drugi argument ne navede, prevodilac nije u stanju da učini izbor. To je ilustrovano poslednjom grupom naredbi.

2.4.4 Dohvatanje delova složenih podataka

Često se javlja potreba za funkcijom čiji je zadatak da, ne menjajući niz, odabere neki element niza i kao rezultat vraća pokazivač ili upućivač na taj element niza. Jasno je da ima smisla primenjivati takvu funkciju i na promenljive i na nepromenljive nizove.

Rezultat indeksiranja u promenljivom nizu je upućivač na promenljive, a u nepromenljivom nizu upućivač na nepromenljive podatke tipa elemenata niza. Slično tome, adresa nekog elementa promenljivog niza je pokazivač na promenljive, a nepromenljivog niza pokazivač na nepromenljive podatke tipa elemenata niza. Zbog toga problem dohvatanja elementa niza ne može da se reši jednom zajedničkom funkcijom za promenljive i nepromenljive nizove, već treba napraviti dve funkcije, najčešće identičnog sadržaja.

Na primer, od funkcija:

```
int& elem (      int* a, int i) { return a[i]; }
const int& elem (const int* a, int i) { return a[i]; }
```

prva može da se pozove za promenljive, a druga za nepromenljive nizove. Skreće se pažnja na to da se tip **int*** smatra različitim od tipa **const int***, pa obe funkcije mogu da imaju isto ime. Prevodilac od te dve funkcije će na mestu pozivanja odabrati prvu ili drugu u zavisnosti od toga da li je argument promenljiv ili nepromenljiv niz (odnosno pokazivač na promenljive ili nepromenljive podatke).

Kada se kasnije u programu napiše:

```
int a[20], i=2;
const int b[20] = {0};
elem (a, i) = 55;
elem (b, i) = 55; // GREŠKA: Menjanje elementa nepromenljivog niza.
int x = elem (b, i);
```

u trećem redu će se pozvati prva varijanta funkcije **elem()** i dobit će se upućivač na promenljiv element niza kome može da se dodeli vrednost. U četvrtom redu se poziva druga varijanta funkcije **elem()**, koja kao rezultat vraća upućivač na nepromenljiv element. Time se, pravilno, sprečava da funkcija **elem()**, posredno, omogući promenu vrednosti dela nepromenljivog niza.

Naravno, nema nikakvog problema da se u poslednjem redu dohvati vrednost elementa nepromenljivog niza radi kopiranja sadržaja u podatak **x**.

Prikazano rešenje s dve funkcije istovetnog sadržaja nije najbolje kada je postupak biranja rezultujućeg elementa složen. Tada je bolje ostvariti jednu od dve verzije funkcije za

biranje i nju pozivati iz druge. Na primer, zadržavajući prvu verziju funkcije `elem()`, druga verzija može da bude:

```
const int& elem (const int* a, int i)
    { return elem (const_cast<int*>(a), i); }
```

Da bi se u telu ove druge varijante funkcije `elem()` pozivala prva varijanta te funkcije potrebno je skinuti, pomoću operatorka `const_cast`, modifikator `const` sa pokazivača `a`. Tip vrednost prve varijante funkcije `elem()` je `int*`, koji se prilikom povratka iz druge varijante, automatskom konverzijom, pretvori u tip `const int*`. Naravno, treba biti vrlo pažljiv da unutar prve varijante funkcije `elem()` stvarno ne dođe do nikakvog menjanja elemenata niza koji je parametar funkcije.

Po uzoru na prikazani postupak može da se ostvaruje i par funkcija koje kao vrednost vraćaju pokazivač na element niza:

```
int* adr (int* a, int i) { return a + i; }
const int* adr (const int* a, int i) { return a + i; }
```

Ove funkcije kasnije mogu da se koriste na sledeći način:

```
*adr (a, i) = 12;
*adr (b, i) = 12; // GREŠKA: Menjanje elementa nepromenljivog niza.
int y = *adr (b, i);
```

U slučaju struktura, kao druge vrste složenih podataka, takođe mogu da se prave parovi funkcija koje kao rezultat vraćaju pokazivač ili upućivač na polje strukture koje se u funkciji prenosi pomoću pokazivača ili upućivača.

Skreće se pažnja na to da ako se struktura u funkciju prenosi pomoću vrednosti, ne sme da se vraća pokazivač ili upućivač na polje parametra. Taj parametar je lokalna prolazna kopija argumenta i uništice se pre nego što se iz funkcije vrati u pozivajući program.

2.4.5 Povezivanje s drugim jezicima △

Konvencije prenošenja argumenata u funkcije i povezivanje identifikatora sa spoljašnjim povezivanjem menjaju se od jezika do jezika. To otežava povezivanje funkcija (potprograma) pisanih na različitim programskim jezicima u jedan izvodljivi program.

U jeziku C++ predviđena je mogućnost pozivanja potprograma pisanih na drugim jezicima, s tim da se od svakog prevodioca zahteva da omogući pozivanje bar funkcija pisanih na jeziku C.

Pošto se u jeziku C++ zahteva barem deklarisanje svih funkcija pre njihovog prvog pozivanja, to se zahteva i za funkcije koje su pisane na drugim jezicima. Opšti oblik navođenja prototipa za njih je:

```
extern "jezik" deklaracija
extern "jezik" { deklaracija deklaracija ... }
```

Jezik predstavlja oznaku programskega jezika čije konvencije pozivanja funkcija treba primeniti. Svaki prevodilac mora da prihvati oznaku C++ i C, s tim da se C++ podrazumeva.

Deklaracija predstavlja prototip funkcije pisane na drugom jeziku ili definiciju funkcije pisane na jeziku C++ ali koju treba pripremiti za pozivanje iz programa pisanih na drugom jeziku.

U slučaju preklapanja imena funkcija najviše jedna od istoimenih funkcija sme da bude po konvencijama pozivanja jezika C.

Evo primera za povezivanje programa pisanih na jeziku C++ s jezikom C:

```
extern "C" double sqrt (double); // C konvencije.
double exp (double); // C++ konvencije.
extern "C" {
    double sin (double);
    double cos (double);
    double atan (double);
}
extern "C" double log (double x) { // C konvencije pozivanja funkcije
    // Telo na jeziku C++. // pisane na jeziku C++.
}
```

2.4.6 Glavna funkcija

Prema *Standardu* glavna funkcija (`main`) bi trebalo da bude tipa `int` sa dva moguća prototipa, kao i u jeziku C (odeljak 1.3.5). Ako se dođe do kraja tela glavne funkcije bez naredbe `return`, program se završava završnim statusom programa 0.

Pre *Standarda* glavna funkcija tipa `int` morala je da se završi naredbom `return`. Ako se program nije završavao naredbom `return`, završni status je bila slučajna vrednost.

2.5 Prostori imena

U velikim programskim sistemima često se dolazi do problema pri korišćenju globalnih imena (s datotečkim dosegom) u raznim delovima programa od strane više programera. Istoime pojedini programeri mogu da koriste u različite svrhe što dovodi do konfliktova.

Namena prostora imena je da grupišu globalna imena u velikim programskim sistemima u više dosega. Svaki prostor imena čini zaseban doseg, takozvani **prostorski doseg**. Ako se delovi programa stave u različite prostore imena ne može doći do konflikta pri korišćenju imena.

2.5.1 Definisanje prostora imena

Opšti oblik za definisanje prostora imena u datotečkom dosegu je:

```
namespace Identifikator { /* Sadržaj prostora imena. */ }
```

Identifikator predstavlja ime prostora imena i mora da bude jedinstven u svom dosegu.

Sadržaj prostora imena je uobičajeni tekst programa na jeziku C++. Svi globalni identifikatori (definisani izvan funkcija) biće u dosegu prostora imena koji se definiše. Zbog toga mogu neposredno da se koriste samo unutar tog prostora imena. Osim ograničenog dosega njihovih identifikatora, svi podaci i funkcije imaju osobine (povezivanje identifikatora, trajnost podataka) uobičajene za globalne podatke.

Prostori imena su otvoreni. To znači da dати identifikator može da se koristi u više definicija prostora imena. Svaka nova definicija će dodati novi sadržaj prostoru imena s tim zajedničkim identifikatorom. Svakog momenta u programu mogu da se koriste identifikatori iz datog prostora imena čije definicije se nalaze pre mesta korišćenja. Kasnije dodati sadržaji ne mogu da se koriste u ranijem delu programa.

Sadržaj prostora imena se ne prenosi iz datoteke u datoteku. Ako su delovi sadržaja prostora imena potrebni u više datoteka, treba ih ponoviti u svakoj od tih datoteka. Sadržaj datoteci moćiće da se koristi ono što je za dati prostor imena navedeno. Ako su neke stvari (na primer deklaracije funkcija i globalnih podataka) potrebne u više datoteka treba ih stavljati u zaglavje (.h) datoteke, i uključivati te datoteke u prevodenje direktivom pretprocesora `#include` gde je to potrebno.

Evo primera za definisanje prostora imena:

```
namespace Alfa {
    int a = 55;           // Definicija globalne promenljive.
    float f () { return 5; } // Definicija funkcije.
    extern char b;        // Deklaracija globalne promenljive.
    double g (double);   // Deklaracija funkcije.
}

namespace Beta {
    double a = 99;        // Isto ime u drugom prostoru imena.
}

namespace Alfa {          // Dodavanje novog sadržaja.
    extern int x;
    double f (int);      // Preklapanje imena funkcije.
    char b = '!';         // Definicija ranije deklarisane ...
    double g (double x){ return x / 2; } // promenljive i funkcije.
}
```

Prvo se definiše prostor imena `Alfa` koji sadrži definiciju globalne promenljive `a`, definiciju funkcije `f()`, deklaraciju globalne promenljive `b` i deklaraciju funkcije `g()`.

Posle se definiše prostor imena `Beta` koji sadrži definiciju globalne promenljive `a` čije ime je korišćeno i u prostoru imena `Alfa`.

Na kraju se prostoru imena `Alfa` dodaju novi sadržaji: deklaracija nove globalne promenljive `x`, deklaracija nove funkcije `f(int)` (preklapanje imena funkcije i ranije uvedene funkcije `f()`), definicija ranije deklarisane globalne promenljive `b` i definicija ranije deklarisane funkcije `g()`.

2.5.2 Upotreba identifikatora u prostoru imena

Unutar prostora imena svi globalni identifikatori se mogu koristiti kao što se i u jeziku C koriste globalni identifikatori.

Identifikatori unutar nekog prostora imena mogu da se dohvate iz bilo kog dela programa (datotečkog dosega ili iz dosega drugih prostora imena) pomoću operatora za razrešenje dosega `::` izrazom čiji je opšti oblik:

Prostor_imena :: identifikator

Prostor_imena je identifikator prostora imena u čijem dosegu se nalazi željeni identifikator.

Operator za razrešenje dosega `::` je prioriteta 19 i grupiše se sleva udesno.

Identifikatori iz nekog prostora imena mogu naredbom `using` da se uvezu u doseg u kome se nalazi ta naredba. Opšti oblik naredbe `using` je:

```
using Prostor_imena :: identifikator
using namespace Prostor_imena
```

ili

2.5.2 Upotreba identifikatora u prostoru

Prvim oblikom uveze se navedeni identifikator posle čega on može da se koristi samostalno, bez navođenja prostora imena kome pripada.

Dругим oblikom uvezu se svi identifikatori iz navedenog prostora imena. Uvezeni identifikatori, pod uslovom da su jedinstveni u tekućem dosegu, u nastavku mogu da se koriste bez navođenja prostora imena kome pripadaju. Skreće se pažnja na to da se uvozi trenutno vidljivi sadržaj prostora imena. Eventualno kasnije dodati identifikatori neće automatski moći da se navedu samostalno. Naravno, novom naredbom `using` mogu i oni postati neposredno dohvatljivi.

Kao posledica uvoženja identifikatora iz prostora imena, može se desiti da neki identifikator ne bude jedinstven. Naime, taj identifikator može biti definisan u lokalnom dosegu i može biti uvezen iz nekoliko prostora imena. U takvom slučaju, sâm identifikator se odnosi na lokalnu definiciju, a za uvezeni identifikator, pomoću operatora za razrešenje dosega, treba naznačiti prostor imena kome pripada željeni identifikator.

Ako ne postoji lokalna definicija identifikatora koji je uvezen iz više prostora imena, prvim oblikom naredbe `using` može da se naznači prostor imena iz kojeg želi da se koristi taj identifikator ako se navodi sam, bez naznake pomoću operatora za razrešenje dosega prostora.

Evo primera za korišćenje identifikatora iz prostora imena iz prethodnog odeljka:

```
int p = Alfa::a;
double q = Beta::a;
using Alfa::x;
int r = x;                                // r = Alfa::x;
using namespace Alfa;
a = 99;                                     // Alfa::a = 99;
double s = f (x);                          // Poziva se Alfa::f(int).
using namespace Beta;
int t = a;                                  // GREŠKA: Alfa::a ili Beta::a?
int u = Alfa::a;
double v = Beta::a;
using Alfa::a;                            // Da se koristi Alfa::a.
int w = a;
```

U prvoj grupi naredbi dohvataju se dva podatka navođenjem prostora imena kojima pripadaju, korišćenjem operatora za razrešenje dosega `::`:

U drugoj grupi naredbi prvo se uveze identifikator `x` iz prostora imena `Alfa`. Posle tога taj identifikator može da se koristi i samostalno.

Posle uvoženja svih identifikatora iz prostora imena `Alfa` na početku treće grupe naredbi, svi identifikatori iz tog prostora imena mogu da se koriste bez navođenja identifikator `Alfa`.

Kada se na početku četvrte grupe naredbi uvoze i identifikatori iz prostora imena `Beta`, u tekućem dosegu postojaće dva identifikatora `a`, jedan iz prostora imena `Alfa` i jedan iz `Beta`. Zbog toga za njihovo korišćenje mora da se, operatorom za razrešenje dosega `::`, navodi i prostor imena iz kojeg želi da se koristi identifikator `a`.

U prvoj naredbi poslednje grupe naredbi je naredbom `using` naznačeno da pod identifikatorom a treba podrazumevati identifikator a iz prostora imena `Alfa`. Posle tога identifikator a može da se koristi neposredno.

Podaci i funkcije koji su deklarirani unutar nekog prostora imena mogu da se definisu i izvan prostora imena kome pripadaju. U tom slučaju mora da se koristi operator za razređivanje dosega da bi se naznačio prostor imena kome pripada podatak ili funkcija koja se definiše.

Navođenjem prostora imena ispred imena funkcije, doseg tog prostora imena se proširuje na celo telo funkcije. Zbog toga, unutar tela funkcije globalni identifikatori iz matičnog prostora imena mogu da se koriste neposredno.

Evo primera za definisanje jednog podatka i jedne funkcije koji su u primeru u prethodnom odeljku samo deklarirani:

```
int Alfa::x = 22;
double Alfa::f (int x) { return x + a; } // Alfa::a
```

2.5.3 Bezimeni prostor imena

Ako se u definiciji prostora imena izostavi *identifikator* prostora imena, dobija se **bezimeni prostor imena**.

Bezimeni prostor imena ne čini zaseban doseg imena već se utapa u okružujući datotečki doseg. Zbog toga sví identifikatori izvan prostora imena („obični“ globalni identifikatori) i u bezimenom prostoru imena moraju biti različiti. Svi oni mogu da se koriste neposredno, bez eventualne naznake da identifikator pripada bezimenom prostoru imena.

Redefinisanje globalnog identifikatora unutar nekog bloka (u lokalnom dosegu) pokriva (čini nevidljivim) te globalne identifikatore unutar lokalnog dosega. Pokriveni globalni identifikatori u jeziku C++ mogu da se dohvate korišćenjem unarnog prefiksнog oblika operatora za razrešenje dosega ::.

Obični globalni identifikatori na ovaj način mogu da se dohvate uvek. Globalni identifikatori iz bezimenog prostora imena na ovakav način mogu da se dohvate samo dok se isti identifikator ne uveze iz nekog drugog prostora imena. Ako se to desi, identifikator iz bezimenog prostora imena ne može nikako da se dohvati iz lokalnog dosega.

Jedina razlika između identifikatora u bezimenom prostoru imena i „običnih“ globalnih identifikatora je da prvi obavezno imaju unutrašnje povezivanje, tj. u svakoj datoteci programskog sistema predstavljaju druge stvari. Ta činjenica ne može nikako da se preinaci (na primer, dodavanjem modifikatora **extern**). Za razliku od njih, identifikatori „običnih“ globalnih podataka i funkcija mogu imati spoljašnje ili unutrašnje povezivanje.

Preporučuje se definisanje globalnih podataka i funkcija s unutrašnjim povezivanjem stavljanjem njihovih definicija u bezimeni prostor imena, umesto korišćenjem modifikatora **static**, kako se to radi u jeziku C.

Evo primera bezimenog prostora imena:

```
namespace {
    int a = 1; // Bezimeni prostor imena.
    // Unutrašnje povezivanje.
}

int b = 2; // Spoljašnje povezivanje.

namespace Alfa {
    int a = 3;
    int b = 4;
}
```

2.5.3 Bezimeni prostor

```
void f () {
    int p = a; // ::a
    int q = b; // ::b
    int r = Alfa::a; // GREŠKA: a ili Alfa::a?
    using namespace Alfa; // GREŠKA: b ili Alfa::b?
    int s = a;
    int t = Alfa::a;
    int u = b;
    int v = ::b;
    int w = Alfa::b; // Definisanje lokalnog b-a.
    int b = 5; // Lokalno b.
    int x = b; // Globalno b.
    int y = ::b;
    int z = Alfa::b;
}
```

U datotečkom dosegu definisane su dve globalne promenljive: a s unutrašnjim povezivanjem (jer se nalazi u bezimenom prostoru imena) i b sa spoljašnjim povezivanjem. I u prostorskom dosegu prostora imena Alfa definisane su dve globalne promenljive sa spoljašnjim povezivanjem i istih imena a i b.

U prvom bloku naredbi u funkciji f () prikazano je dohvatanje definisanih globalnih promenljivih. Skreće se pažnja da za dohvatanje identifikatora datotečkog dosega nije bitno da li se nalaze u bezimenom prostoru imena ili ne.

Na početku drugog bloka naredbi u datotečki doseg je uvezen sadržaj prostora imena Alfa. Posle toga identifikatori a i b više nisu jedinstveni, pa ne mogu da se napišu sami, bez operatora za razrešenje dosega. Šta više, identifikator a iz bezimenog prostora imena ne može ni na koji način da se dohvati.

Prvom naredbom u poslednjem bloku definiše se lokalna promenljiva b. Posle toga samo b označava tu lokalnu promenljivu, dok za druge dve promenljive imena b treba koristiti operator za razrešenje dosega.

2.5.4 Uklapanje prostora imena

Unutar prostora imena mogu da se definisu i drugi, uklapljeni, prostori imena. Takve definicije moraju biti u globalnom delu (izvan funkcija) spoljašnjeg prostora imena.

Identifikatori unutrašnjih prostora imena su u prostorskom dosegu spoljašnjeg prostora imena. Mogu samostalno da se koriste samo unutar spoljašnjeg prostora imena. Izvan spoljašnjeg prostora imena, kao i za sve identifikatore, mora se operatom za razrešenje dosega naznačiti pripadnost identifikatora unutrašnjeg prostora imena spoljašnjem prostoru imena.

Doseg spoljašnjeg prostora imena je globalan u odnosu na unutrašnji prostor imena. Zbog toga u unutrašnjem prostoru imena mogu samostalno da se dohvate identifikatori iz spoljašnjeg prostora imena, koji su definisani ispred unutrašnjeg prostora imena. Naravno, u slučaju redefinisanja identifikatora u unutrašnjem prostoru imena, istoimeni identifikator iz spoljašnjeg prostora imena može da se dohvati samo pomoću operatara za razrešenje dosega.

Doseg unutrašnjeg prostora imena je lokalan u odnosu na spoljašnji prostor imena. Zbog toga iz spoljašnjeg prostora imena identifikatori koji se nalaze u unutrašnjem prostoru imena mogu da se dohvate samo pomoću operatara za razrešenje dosega. Naravno, samo u delu spoljašnjeg prostora imena koji se nalazi iza definicije unutrašnjeg prostora imena.

Evo primera uklapanja prostora imena:

```
namespace Alfa {           // Spoljašnji prostor imena.
    int a = 1;
    int f () { return 2; }
    extern int b;
    int g ();
}

namespace Beta {           // Unutrašnji prostor imena.
    extern int a;
    int f ();
    int g () { return b; } // Alfa::b
}

int c = a + Beta::a;      // Alfa::a + Beta::a

void h () {
    int p = Alfa::a;
    int q = Alfa::f ();
    int r = Alfa::Beta::a;
    int s = Alfa::Beta::f ();
    using namespace Alfa;
    int t = a;             // Alfa::a
    int u = f ();          // Alfa::f()
    int v = Beta::a;       // Alfa::Beta::a
    int w = Beta::f ();   // Alfa::Beta::f()

    int Alfa::b = 3;
    int Alfa::g () { return 4; }
    int Alfa::Beta::a = 5;
    int Alfa::Beta::f () { return b; } // Alfa::b
}
```

Na početku je definisan prostor imena Alfa koji sadrži unutrašnji prostor imena Beta. Unutar prostora imena Beta, već definisani identifikatori prostora imena Alfa mogu da se koriste samostalno. Iza definicije prostora imena Beta, u dosegu prostora imena Alfa, mogu da se koriste i identifikatori iz prostora imena Beta, ali samo uz korišćenje operatora za razrešenje doseg-a.

U prvom delu funkcije h() prikazana je upotreba identifikatora iz prostora imena Alfa i Beta. Pošto se identifikator Beta nalazi u prostorskem, a ne u datotečkom dosegu, ni on nije samostalno dostupan kao ni bilo koja druga vrsta identifikatora iz prostora imena Alfa. Skreće se pažnja na način označavanja da neki identifikator pripada unutrašnjem prostoru imena Beta.

U drugom delu funkcije h() prikazana je upotreba istih identifikatora posle uvoženja identifikatora iz prostora imena Alfa u blokovski doseg funkcije h().

U poslednjem bloku naredbi prikazano je definisanje globalnih promenljivih i funkcija koje su ranije deklarisane unutar prostora imena Alfa, odnosno Beta.

2.5.5 Prostor imena std

Sva Standardom predviđena zaglavlj-a sve globalne identifikatore stavljuju u prostor imena std.

Standardom je predviđena mogućnost da usklađivanje standardnih zaglavlj-a ne bude, mada može, u vidu tekstualnih datoteka. Zbog toga imena standardnih zaglavlj-a ne sadrže proširenje imena .h, što je uobičajeno u jeziku C.

Evo primera za korišćenje zaglavlj-a <iostream> za čitanje i pisanje podataka.

```
#include <iostream>
int main () {
    int n;
    std::cout << "n? "; std::cin >> n;
    std::cout << "n^2= " << n*n << std::endl;
    using namespace std;
    cout << "n? "; cin >> n;
    cout << "n^2= " << n*n << endl;
}
```

U prvom delu programa uz svaki identifikator u vezi s čitanjem i pisanjem podataka navedeno je da pripada prostoru imena std. U drugom delu, posle uvoženja tih identifikatora u blokovski doseg glavne funkcije, isti identifikatori su mogli da se koriste i samostalno.

Za zaglavlj-a koja su postojala i pre uvođenja Standarda i dalje postoje varijante koje globalna imena ne stavljuju u nijedan prostor imena. U takva zaglavlj-a spadaju već pomenuta zaglavlj-a <iostream> i <iomanip>. Te varijante zaglavlj-a i dalje sadrže proširenje imena .h. Na primer:

```
#include <iostream.h>
int main () {
    int n;
    cout << "n? "; cin >> n;
    cout << "n^2= " << n*n << endl;
}
```

Treba još napomenuti da, mada Standard još dozvoljava upotrebu starog načina označavanja zaglavlj-a, neki prevodoci ih već ne prihvataju.

Za deklarisanje bibliotečkih funkcija nasleđenih iz jezika C, standardna zaglavlj-a, koja koriste prostor imena std, u imenu imaju dodato slovo c ispred ranije korišćenog imena i nemaju proširenje imena .h. I dalje mogu da se koriste i stara imena zaglavlj-a, koja definisana imena ne stavljuju ni u jedan prostor imena. Neki od parova novih i starih imena zaglavlj-a su:

<cstdlib>	<stdlib.h>
<cstdio>	<stdio.h>
<cmath>	<math.h>
<cctype>	<ctype.h>
<cstring>	<string.h>

Evo primera korišćenjem matematičkih funkcija:

```
#include <iostream>
using namespace std;
#include <cmath>

int main () {
    double x=2, y=5;
    cout << "x^y= " << std::pow(x,y) << endl;
    using namespace std;
    cout << "x^y= " << pow(x,y) << endl;
}
```

Skreće se pažnja na to da pošto se direktiva #include za zaglavlj-e <cmath> nalazi iza prve naredbe using, na mestu prvog pozivanja funkcije pow() njen identifikator još nije

uvezen u tekući doseg, pa je morao da se navede i prostor imena std. Iza druge naredbe `using` to više nije potrebno.

Da je u direktivi `#include` korišćeno ime `<math.h>`, imena svih matematičkih funkcija bi se automatski stavljala u datotečki doseg. Standard ne preporučuje da se ova mogućnost koristi u novim programima na jeziku C++.

Iz prethodnog primera može se zaključiti da je najbolje prvo navesti direktive `#include` za sva potrebna zaglavla i tek posle toga staviti naredbu `using`. Tako će sva potrebna imena postati odjednom raspoloživa.

2.6 Zadaci

2.6.1 Metoda podele za uređivanje

Zadatak:

Napisati na jeziku C++ funkciju za uređivanje niza celih brojeva po neopadajućem redosledu metodom podele (*quick sort*). Napisati na jeziku C++ program za prikazivanje rada te funkcije.

Rešenje:

Metoda podele za uređivanje je algoritam tipa „podeli i pobedi“ (*divide and conquer*). Ideja se sastoji u tome da se niz od n elemenata na nekom mestu i ($i = 0, 1, \dots, n-1$) podeli na dva dela tako da levo od elementa a_i ne bude nijedan element čija je vrednost veća od a_i , a desno nijedan element koji je manji od a_i (slika 2.2.a). Pošto se posle toga element a_i nalazi na svom konačnom mestu, potrebno je da se, međusobno nezavisno, uređuju elementi levo i desno od referentnog elementa a_i . Za što veću efikasnost bilo bi poželjno da ta dva dela budu iste veličine. To se, naravno, neće uvek dešavati.

Metoda podele je, očigledno, rekurzivan postupak. Postupak se završava kada se dobiju delovi manji od dva elementa.

Ostaje još pitanje kako se pravi podela niza? To pokazuje slika 2.2.b na primeru niza od deset elemenata. Red na vrhu prikazuje početni izgled niza.

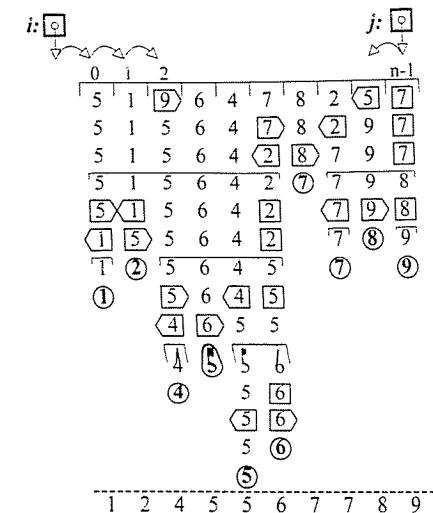
Na početku stvaranja podele uočava se neka referentna vrednost. Jedna mogućnost je da to bude vrednost poslednjeg elementa u nizu. U posmatranom primeru to je 7, na slici zaokružen kvadratičem.

Posle se, polazeći od početka niza, traži prvi element čija vrednost nije manja od referentne vrednosti. Za ovo traženje se na slici 2.2.b koristi brojač i . Na taj način, u posmatranom primeru, uočen je element čija je vrednost 9.

Zatim, polazeći od kraja niza, traži se prvi element čija vrednost nije veća od referentne vrednosti. Za ovo traženje na slici 2.2.b koristi se brojač j . Na taj način, u posmatranom primeru, uočen je element čija je vrednost 5.

Na kraju, ako je $i < j$, elementi a_i i a_j međusobno se zamenjuju i postupak stvaranja podele nastavlja se prema sredini niza. Ako se pomeranje brojača i i j u prethodna dva pasusa završi sa $i \geq j$, zamenjuju se referentni element i element a_i . Time je referentni element došao na svoje konačno mesto i završeno je stvaranje podele (četvrti red na slici 2.2.b).

2.6.1 Metoda podele za uređivanje



Slika 2.2 – Metoda podele za uređivanje

Može da se pokaže da je vreme potrebno za uređivanje niza od n elemenata u proseku proporcionalno sa $n \log n$. Vreme potrebno za uređivanje niza jednostavnijim metodama, na primer metodom umetanja, je proporcionalno sa n^2 . Pošto funkcija $n \log n$ raste mnogo sporije od n^2 , metoda podele je znatno efikasnija od drugih metoda za duže nizove. Za kraće nizove, do par desetina elemenata, jednostavnije metode su efikasnije, jer je za njih faktor proporcionalnosti uz navedene funkcije osetno manji nego za metodu podele.

Program 2.1 predstavlja ostvarenje navedenog postupka. Sastoјi se od tri funkcije od kojih samo jedna može da se koristi izvan datoteke `qsort.C`.

U bezimenom prostoru imena su definisane dve pomoćne funkcije, namenjene za korišćene samo u datoteci `qsort.C`.

Prva funkcija, `zameni`, je ugrađena (`inline`). Međusobno zamenjuje dva celobrojna podatka. Pošto funkcija treba da menja vrednosti svojih parametara, oni su deklarisani kao upućivači na cele brojeve (`int&`).

Dруга funkcija, `podeli`, pravi podelu niza a od n elemenata na dva dela. Referentnu vrednosti, koja se pominje u prethodnom izlaganju, pridružuje se upućivač `b`. Za međusobnu zamenu dva elementa u nizu, koristi se funkcija `zameni`. Drugi argument te funkcije je uslovni izraz kao `lvrednost`. Sve dok podela nije završena vrednost tog izraza je element niza `a[j]`, pa se međusobno zamenjuju sadržaji elemenata `a[i]` i `a[j]`. Samo u toku poslednjeg prolaza kroz ciklus vrednost uslovnog izraza je referentni podatak `b`, tj. element niza `a[n-1]`, što obezbeđuje da sadržaj tog elementa dode na svoje ispravno mesto. Vrednost funkcije `podeli` je indeks elementa u nizu a koji deli niz na dva dela.

Samo je funkcija `qsort` vidljiva izvan posmatrane datoteke. Stavljena je u prostor imena `Sort` s idejom da bi se u taj prostor imena mogle kasnije da dodaju i funkcije za uređivanje po drugim algoritmima. Funkcija ima šta da radi ako je broj elemenata niza veći od jedan. U tom slučaju niz se prvo razbije na dva dela pozivanjem funkcije `podeli`. Posle

```
// Uredivanje niza brojeva metodom podele (quick sort).

namespace {
    inline void zameni (int& x, int& y)
    { int z = x; x = y; y = z; }

    int podeli (int a[], int n) {
        int b = a[n-1], i = -1, j = n-1;
        while (i < j) {
            do i++; while (a[i] < b);
            do j--; while (j>=0 && a[j]>b);
            zameni (a[i], (i<j)?a[j]:b);
        }
        return i;
    }

    namespace Sort {
        void qsort (int a[], int n) {
            if (n > 1) {
                int i = podeli (a, n);
                qsort (a, i);
                qsort (a+i+1, n-i-1);
            }
        }
    }
}
```

Program 2.1 – Funkcija za uređivanje metodom podele (qsort.C)

toga, rekurzivnim pozivima uređuju se dobijena dva dela. Prvi deo počinje na adresi *a* i ima *i* elemenata, dok drugi deo počinje na adresi *a+i+1* i ima *n-i-1* elemenata (slika 2.2.a). Očigledno je da se niz rekurzivnih poziva prekida kada se, deleći niz brojeva na sve manje delove, dođe do dela od jednog ili nula elemenata.

Program 2.2 prikazuje rad funkcije *qsort*.

Ulez i izlaz podataka u programu ostvaruje se operatorima *>>* i *<<* pristupajući tekstualnim tokovima *cin* i *cout* koje predstavljaju glavni ulaz i glavni izlaz računara. Zbog toga u prevođenje treba da se uključi zaglavljje *<iostream>*.

Zaglavljje *<cstdlib>* je potrebno zbog korišćenja generatora slučajnih brojeva *rand()* za stvaranje slučajnih nizova za uređivanje.

Posle obe direktive *#include* nalazi se naredba *using* kojom se potrebne informacije učine neposredno dostupnim.

Pošto se definicija funkcije *qsort* nalazi u drugoj datoteci, u posmatranoj datoteci treba da se nalazi prototip te funkcije. U prototipu su navedeni samo tipovi parametara, a ne i identifikatori. Sam prototip je stavljen u prostor imena *Sort*, pošto se i definicija te funkcije nalazi u tom prostoru imena.

Struktura programa je ciklus s izlazom u sredini. Prvo se pročita željena dužina niza za obradu *n*, pa ako je pročitana vrednost jednakna ili manja od nule, izlazi se iz ciklusa, inače se pristupa obradi u drugom delu ciklusa.

Pre svega, operatorom *new* se dodeljuje memorija u dinamičkoj zoni za niz od *n* celobrojnih elemenata. Rezultat operacije dodeljuje se pokazivaču *a*, tipa *int**, kao početna

Prikaz rada funkcije *qsort*.

```
#include <iostream>
#include <cstdlib>
using namespace std;

namespace Sort {
    void qsort (int[], int);
}

int main () {
    while (true) {
        int n; cout << "\n\nDuzina niza? "; cin >> n;
        if (n <= 0) break;
        int* a = new int [n];
        cout << "\nPocetni niz:\n\n";
        for (int i=0; i<n; i++)
            cout << (a[i] = (int)rand() / ((RAND_MAX+1.) * 10)
                  << ((i%30==29 || i==n-1) ? '\n' : ' ');
        Sort::qsort (a, n);

        cout << "\nUredjeni niz:\n\n";
        for (int i=0; i<n; i++)
            cout << a[i] << ((i%30==29 || i==n-1) ? '\n' : ' ');
        delete [] a;
    }
}
```

Program 2.2 – Prikaz rada funkcije *qsort* (qsortt.C)

vrednost. Taj pokazivač u nastavku programa predstavlja obrađivani niz. Skreće se pažnja na definisanje pokazivača u nizu običnih naredbi.

Posebne dodele memorije niz se popunjava slučajnim jednocifrenim decimalnim celim brojevima. Generisani brojevi se istovremeno uskladištavaju i prikazuju na glavnom izlazu, po 30 brojeva u jednom redu. Skreće se pažnja na definisanje promenljive *i* u delu za pripremne radnje ciklusa *for*.

Oformljeni niz se uređuje pozivanjem funkcije *qsort*, pri čemu je korišćen i operator za razrešenje dosega *::*, pošto se funkcija nalazi u prostoru imena *Sort*.

Iza toga sledi sledeći ciklus *for* za prikazivanje rezultata uređivanja. Pošto je definicija promenljive *i*, koja je u sastavu prethodnog ciklusa *for*, prestala da važi po završetku tog ciklusa, ta promenljiva morala je sada da se definiše iznova.

Dobar program uvek zahteva oslobođanje memorije dodeljene dinamičkim podacima kad god ti podaci više nisu potrebni. U posmatranom slučaju potrebno je na kraju svakog ciklusa oslobođiti memoriju dodeljenu na početku ciklusa, jer u narednom prolasku kroz ciklus u promenljivu *a* smestice se nova adresa za novi niz. Ako bi se propustilo oslobođanje nepotrebne memorije, ista bi postala nepristupačna, mada i dalje zauzeta.

Nepotreban memoriski prostor se u programu 2.2 oslobađa operatorom *delete* u poslednjoj naredbi glavnog ciklusa.

Rezultat 2.1 prikazuje primer rada programa 2.2.

```
% CC qsortt.C qsort.C -o qsortt
% qsortt
```

Duzina niza? 40

Pocetni niz:

```
0 3 0 3 2 5 1 7 9 2 4 1 6 5 0 1 8 6 7 8 9 2 4 9 8 9 8 4 6 6
5 5 7 1 4 1 6 4 4 5
```

Uredjeni niz:

```
0 0 0 1 1 1 1 2 2 2 3 3 4 4 4 4 4 5 5 5 5 6 6 6 6 6 7
7 7 8 8 8 8 9 9 9 9
```

Duzina niza? 0

Rezultat 2.1 – Uređivanje niza programom 2.2

3 Klase

Klase u jeziku C++ su složeni tipovi podataka koji se sastoje od uredenih nizova elemenata koji mogu da budu međusobno različitih tipova. Ti elementi nazivaju se **članovi klase**.

Podaci klasnih tipova predstavljaju **primerke** date klase i nazivaju se **objekti**. Termin klasa često se koristi, skraćeno, za označavanje pojma primerak klase. Iz konteksta treba tada zaključiti, da li reč klasa označava tip ili objekat.

Pojam **podatak** obuhvata kako vrednosti standardnih tipova, tako i objekte klasnih tipova. Kvalitativno se razlikuju pojedinačni podaci i nizovi. Prosti (skalarni) podaci, pokazivači strukture i objekti smatraju se pojedinačnim podacima.

Članovi klase mogu da budu podaci ili funkcije. Vrednosti podataka članova čine **stanje** (vrednost) objekta i nazivaju se **polja** ili **atributi** klase, odnosno objekta. Funkcije članovi služe za izvođenje raznih operacija nad podacima članovima (poljima) čime menjaju stanje objekta na koji se primenjuju. Nazivaju se **metode** klase.

Članovi klase mogu da budu javni ili privatni. **Privatnim** članovima može da se pristupa samo iz unutrašnjosti posmatrane klase. To znači da privatne polja (attribute, podatke članove) mogu da koriste ili menjaju samo metode (funkcije članovi) date klase, a privatne metode mogu da pozivaju samo druge metode iste klase. **Javnim** članovima može da se pristupa bez ograničenja kako iz unutrašnjosti klase, tako i iz delova programa izvan posmatrane klase. Kaže se da ove osobine označavaju **prava pristupanja** ili **vidljivost** članova.

U većini slučajeva polja su privatna, dok su od metoda neke privatne a neke javne. Javne metode su tada jedina mogućnost da se dođe do podataka sadržanih u klasnim objektima. Upravo je i bila namera prilikom uvođenja klasa da se to postigne.

Klase su pravi tipovi jer:

- određuju moguće vrednosti objekata,
- određuju moguće operacije nad objektima,
- sprečavaju izvršavanje bilo koje druge operacije nad objektima,
- obezbeđuju obaveznu inicijalizaciju (dodelu početnih vrednosti) stvaranih objekata,
- obezbeđuju uništavanje objekata kada više nisu potrebni.

3.1 Definisanje i deklarisanje klase

Definicija klase predstavlja navođenje svih članova klase. Na osnovu te definicije mora da se zna veličina potrebnog memorijskog prostora za smeštanje pojedinih objekata tipa te klase. Klasa se definiše opisom **class** čiji je opšti oblik:

```
class Identifikator {           član član ...
    public: član član ...
    private: član član ...
    ...
};
```

Deklaracijom klase samo se naznači da neki identifikator predstavlja klasu, ali se ništa ne kaže o sadržaju klase. Opšti oblik deklaracije klase je:

```
class Identifikator;
```

Posle deklaracije klase mogu da se definišu pokazivači i upućivači na objekte tipa te klase, ali ne mogu da se definišu objekti tipa te klase, niti da se pristupa članovima klase. Za definisanje objekata neophodno je da prethodno bude navedena potpuna definicija klase. Pri definisanju objekata dodeljuje se i memorija, a za to je potrebno da se zna veličina objekata, a pri pristupanju članovima potrebno je da se zna koji članovi postoje!

Identifikator klase služi za identifikaciju klase koja se definiše ili deklariše. Ima status identifikatora tipa, pa može samostalno da se koristi u naredbama za definisanje podataka i na drugim mestima gde se očekuje oznaka tipa (videti i odeljak 2.2.5).

Javni i privatni delovi klase razgraničavaju se oznakama **public** i **private**. Ista oznaka sme i više puta da se koristi. Početni deo klase, pre prve oznake **public**, je privatna. Članovi klase navedeni u privatnim delovima su privatni članovi, a članovi navedeni u javnim delovima su javni članovi.

Član u definiciji klase može da bude:

- **Definicija polja** u obliku naredbe za definisanje podataka. Odjednom mogu da se definišu više polja zajedničkog osnovnog tipa (neki od njih mogu da budu pokazivači, upućivači ili nizovi). Ne mogu da se navedu inicijalizatori, niti polja mogu da budu nepostojana.
- Polja ne mogu da budu tipa klase koja se upravo definiše, ali mogu da budu pokazivači ili upućivači na primerke te klase.
- **Definicija metode** koja se po formi poklapa s definicijom običnih (globalnih) funkcija.

Za metode koje se definišu u definiciji klase podrazumeva se modifikator **inline**. One se, dakle, ugrađuju u kôd i nazivaju se i *ugrađene metode*. Tumačenje teha metode odlaze se do kraja definicije klase. To omogućava da se u telu metode koriste i članovi klase čiji će identifikatori biti uvedeni tek u nastavku definicije klase.

Vrednost metode može da bude tipa klase koja se upravo definiše. Takođe, i parametri tih funkcija mogu da budu tipa tekuće klase. Naravno, mogu da budu i pokazivači ili upućivači na primerke te klase.

- **Deklaracija metode** koja se po formi poklapa s prototipom običnih funkcija.

Metode koje se u definiciji klase samo deklarišu moraju da budu definisane na nekom drugom mestu, izvan definicije klase. Te metode, takođe, mogu biti

3.1 Definisanje i deklarisanje klasa

ugrađene, ali to treba eksplicitno tražiti modifikatorom **inline** prilikom definisanja.

- Naredbe **typedef** i **enum** kojima se uvode identifikatori tipa ili identifikatori simboličkih konstanti, ali koje ne stvaraju članove klase (ne utiču na veličinu objekata niti na funkcionalnost klase).

Doseg svih identifikatora unutar klase je od mesta definisanja do kraja klase. Kaže se da članovi klase imaju **klasni doseg**.

Definicija klase mora da bude dostupna prevodiocu prilikom prevodenja svakog programskog modula koji stvara objekte date klase. Zato se definicije klasa obično stavljaju u zaglavlja (datoteke .h), čiji se sadržaji u prevodenje uključuju direktivama **#include** pretprocesora.

Evo primera definicije klase:

```
class Nesto {
    int a, b;           // Privatna polja.
    int f (int);        // Privatna metoda.
    Nesto n;            // GREŠKA: ne može polje tipa Nesto!
    Nesto* pn;          // Pokazivač na Nesto može.
    Nesto& un;          // Upućivač na Nesto može.
    public:
        int c;           // Javno polje.
        int g (int);     // Javna metoda.
        Nesto h (Nesto); // Metoda tipa Nesto može, takođe
                           // i parametar tipa Nesto.
};
```

3.2 Objekti klasnih tipova

Posle definisanja neke klase automatski stoje na raspolaganju sledeće radnje nad tom klasmom:

- definisanje (stvaranje) objekata i nizova objekata ([]),
- definisanje pokazivača (*) i upućivača (&) na objekte,
- dodeljivanje vrednosti (=) jednog objekta drugom,
- nalaženje adresa objekata (&) i pristup objektima na osnovu adrese (*) ili indeksiranjem ([]), i
- pristup članovima objekata neposredno (.) ili posredno (->).

Objekti klasnih tipova (primerci klasa) definišu se uobičajenim naredbama za definisanje podataka. Za oznaku tipa treba da bude odabran identifikator željene klase.

Za svaki objekat date klase stvara se zaseban komplet svih polja te klase.

Mada se kaže da objekti sadrže i metode, to ne treba doslovce shvatiti. Prevod svake metode se, naravno, smešta u memoriju samo jednom. Taj jedini primerak se koristi kad god se metoda pozove za bilo koji objekat. Pripadanje metode objektu treba shvatiti u logičkom smislu: kada se metoda pozove za neki objekat obraduje polja tog objekta.

Za pokazivače na objekte važe sva pravila adresne aritmetike.

Dodela vrednosti jednog objekta drugom podrazumeva kopiranje vrednosti svih polja izvorišnog objekta u odredišni objekat, polje po polje. Za slučaj pokazivačkih polja prenosi se samo vrednost pokazivača, a ne i pokazivani podatak (podobjekat).

Pristup metodama operatorima . ili -> podrazumeva pozivanje te metode (funkcije).

Objekti mogu da budu parametri funkcija kao i vrednosti funkcija. Objekti kao parametri funkcija prenose se pomoću vrednosti, tj. objekat koji se navede kao argument kopira se u odgovarajući parametar na isti način kao i prilikom dodelje vrednosti. Ako to ne odgovara, treba koristiti pokazivače ili upućivače.

Evo primera definisanja i korišćenja primeraka klase `Nesto` iz prethodnog odeljka:

```
Nesto n, *pn, &un=n; // Objekat, pokazivač i upućivač.
n.c = 55;           // Dodela vrednosti javnom polju.
int i = n.g (12); // Poziv javne metode.
n.a = 0;           // GREŠKA: dodela privatnom polju.
int j = n.f (-2); // GREŠKA: poziv privatne metode.
pn = new Nesto;   // Stvaranje dinamičkog objekta.
*pn = un;          // *pn = n; (un je upućivač!)
un = n.n (*pn);  // Argument i rezultat su tipa Nesto.
delete pn;         // Uništavanje dinamičkog objekta.
```

Postoje slučajevi kada neki identifikatori klasnog dosega mogu da se koriste izvan njihove klase i bez pominjanja konkretnog objekta. Takvi su, na primer, identifikatori uvedeni naredbama `typedef` ili `enum`. Ti identifikatori, slično identifikatorima prostorskog dosega (odeljak 2.5.2), mogu da se dohvate pomoću operatora za razrešenje dosega (`::`). Prvi operand treba da bude identifikator klase, a drugi operand identifikator koji je uveden unutar te klase.

Evo primera za korišćenje nabranja izvan klase u kojoj je ono definisano:

```
class Prozor {
    ...
public:
    enum Boja {CRNA, PLAVA, ZELENA, CRVENA, BELA};
    ...
};

Prozor::Boja boja = Prozor::BELA;
if (boja != Prozor::CRNA) { ... }
```

U klasi `Prozor` definisano je nabranje `Boja` s nekoliko simboličkih konstanti. Izvan klase je definisan podatak `boja` tipa tog nabranja koji je inicijalizovan jednom od simboličkih konstanti. U poslednjoj naredbi promenljiva `boja` se upoređuje s jednom od mogućih vrednosti nabranja `Boja`. Za sve identifikatore iz klase `Prozor` korišćen je operator `::`.

3.3 Metode klasa

Metode klase pored svojih parametara imaju još jedan „skriven” parametar. Skriven, zato što se ne vidi u prototipu funkcije. Taj parametar je adresa objekta (primerka klase) za koji je metoda pozvana, tj. adresa prvog operanda operatora `.` ili vrednost prvog operanda operatora `->`. Taj objekat se naziva **i tekući objekat** jer je to objekat koji metoda obrađuje.

Za pristupanje članovima tekućeg objekta dovoljno je navesti samo identifikator člana, kao što se to radi s običnim podacima ili funkcijama.

Identifikator skrivenog parametra u svakoj metodi je `this`. Tip skrivenog parametra u svakoj klasi je nepromenljiv pokazivač na tu klasu (u klasi `T` to je `T*const`). Vrednost tog pokazivača ne može da se promeni već samo eventualno vrednost pokazivanog (tekućeg) objekta.

Kad god se unutar metode klase navodi identifikator nekog člana bez navođenja konkretnog objekta, podrazumeva se `this->` ispred tog identifikatora, tj. pristupa se odgovarajućem članu tekućeg (skrivenog) objekta. Tekući objekat u celini može da se dohvati posrednim pristupom (indirektnim adresiranjem) pomoću izraza `*this`.

Pokazivač `this` se eksplisitno koristi relativno retko. Uglavnom se koristi samo ako tekući objekat (za koji je metoda pozvana) treba da bude vrednost funkcije metode koja se upravo izvršava. Drugi primer korišćenja je kada tekući objekat ili njegova adresa treba da bude argument neke funkcije ili, pak, ako tekući objekat treba uključiti u neku ulančanu listu.

Kao i svi parametri metode tako i tekući objekat može da bude nepromenljiv (`const`) ili nepostojan (`volatile`). To se u deklaraciji ili definiciji metode označava dodavanjem odgovarajućeg modifikatora iza zatvorene oble zgrade na kraju spiska parametara:

```
tip funkcija (parametri) modifikator;
tip funkcija (parametri) modifikator blok
```

ili

Modifikator može da bude `const`, `volatile` ili oba.

Modifikator `const` označava da metoda neće promeniti vrednost nijednog polja u objektu za koji je pozvana (tekućem objektu), a koji nisu obeleženi modifikatorom `mutable`. Metoda može da se poziva za nepromenljive i promenljive objekte. Ponekad se kaže da je metoda „nepromenljiva”.

Modifikator `volatile` označava da je metoda svesna činjenice da vrednost tekućeg objekta može da se promeni bilo kad u toku izvršavanja metode, mimo njene kontrole. Metoda može da se poziva za nepostojane i promenljive objekte. Ponekad se kaže da je metoda „nepostojana”.

Ako su oba modifikatora prisutna metoda može da se poziva za sve vrste objekata: promenljive, nepromenljive i nepostojane. U odsustvu oba modifikatora metoda može da se poziva samo za promenljive objekte.

Ako se metoda definiše unutar definicije klase nema nikakvog problema u korišćenju identifikatora ostalih članova te klase. Svi se oni nalaze unutar istog dosega.

Prilikom odvojenog definisanja metode definicija se nalazi izvan dosega kome pripada metoda. Zato je neophodno operatorom za razrešenje dosega (`::`) navesti ime klase kojoj pripada metoda. Drugi operand u posmatranom slučaju, treba da je identifikator metode koja se definiše. Time se doseg navedene klase proširuje na celu definiciju metode, pa unutar tela metode članovi klase mogu da se koriste navodenjem samo njihovih identifikatora.

Evo primera kojim se definišu metode klase `Nesto` iz odeljka 3.1:

```
int Nesto::f (int x) {
    int z = x + a;                                // z = x + this->a
    return z / b;                                  // z / this->b
}

int Nesto::g (int x) {
    this->a = x * this->b;                         // Ovako se obično ne
                                                       // piše (a = x * b).
    return x / a;
}

Nesto Nesto::h (Nesto x) {
    Nesto y;
    y.a = x.a + a;
    y.b = x.b + b;
    return y;
}
```

Metode koje kao vrednost funkcije vraćaju upućivač ili pokazivač na neko polje tekućeg objekta i pri tome ne menjaju tekući objekat ne treba ostvariti kao:

`tip& m (...) const;`

jer pomoću dobijenog pokazivača ili upućivača moglo bi da se promeni vrednost odgovarajućeg polja, čak i ako je tekući objekat nepromenljiv (samo polje u klasi, pri tome, nije i ne može biti označen kao nepromenljiv, jer nisu svi objekti nepromenljivi). Metoda bi na posredan način omogućila promenu nepromenljivog objekta, a prevodilac ne bi mogao to da spreći. Umesto toga treba praviti dve metode:

`tip& m (...);
const tip& m (...) const;`

Za promenljive objekte će se pozivati prva, a za nepromenljive druga varijanta. Pomoću rezultata druge metode neće moći da se promeni vrednost upućivanog ili pokazivanog polja unutar nepromenljivog objekta. Naravno, jedna od te dve metode uvek može da se ostvari tako da samo poziva onu drugu metodu. Sličan problem je obrađivan i u odeljku 2.4.4 za slučaj običnih funkcija.

3.4 Konstruktori

Novi objekti (primerci klasa) mogu da se stvaraju:

- izvršavanjem naredbi za definisanje podataka (trajni objekti koji se stvaraju prilikom prvog nailaska na naredbu za definisanje i prolazni objekti koji se stvaraju svaki put kada se dođe do naredbe za definisanje),
- stvaranjem lokalnih podataka za parametre prilikom pozivanja funkcija (prolazni objekti),
- u toku izvršavanja složenih izraza za odlaganje međurezultata (privremeni objekti), ili
- izvršavanjem operatora `new` (dinamički objekti).

Stvaranje objekata podrazumeva dodelu memorijskog prostora i, eventualno, inicijalizaciju dodeljivanjem nekih početnih vrednosti.

Memorijski prostor se dodeljuje automatski za sva polja klase prema definiciji klase.

Inicijalizaciju, tj. dodelu početnih vrednosti, poljima stvaranih objekata treba da predviđa programer. Izuzetno, za trajne (`static`) objekte podrazumevane početne vrednosti su nule u svim bajtovima svih polja klase.

Naročito je važna ispravna inicijalizacija objekata klase koje imaju i pokazivačka polja. To može da zahteva i dodatnu dodelu memorijskog prostora u dinamičkoj zoni, što ne može nikako da se uradi automatski.

Inicijalizaciju trajnih, prolaznih i dinamičkih objekata može da predviđi programer. Postoji, međutim, opasnost da on to propusti da učini. Što se privremenih objekata tiče njih čak i da hoće programer ne može da inicijalizuje. Da li će se i koliko privremenih objekata koristiti zavisi od prevodilca. Privremeni objekti nemaju ni imena, niti pokazivače na njih pomoću kojih bi programer mogao da im pristupi.

Mehanizam koji jezik C++ nudi za zagaranovanu inicijalizaciju svih kategorija objekata zasniva se na specijalnim metodama klase koje se nazivaju **konstruktori**. Ako su ispunjeni određeni uslovi (koji su izloženi u narednim odeljcima), konstruktori se pozivaju

automatski kad god se stvara neki objekat date klase, bez obzira da li se radi o stalnim, prolaznim, dinamičkim ili pak privremenim objektima.

Neinicijalizovani objekat u stvari nije objekat, već samo jedno parče memorije čiji sadržaj nema nikakvog smisla. Pravi objekat je nešto drugo, ima „inteligentan“ sadržaj koji zadovoljava osobine svoje klase. Zadatak konstruktora je da parče dodeljene memorije pretvoriti u objekat sa svim obeležjima svoje klase.

Konstruktori su pojedinačne metode klase koje imaju iste identifikatore kao i klase kojima pripadaju (`T()` za klasu `T`). Koriste se za automatsku inicijalizaciju primeraka svojih klasa u momentima stvaranja tih primeraka.

3.4.1 Definisanje klase s konstruktorima

Data klasa može da ima više konstruktora. Znači, preklapanje imena funkcija dozvoljeno je i za slučaj konstruktora. Konstruktori mogu da imaju proizvoljan broj parametara, a mogu da budu i bez jednog parametra. Parametar konstruktora ne može da bude tipa svoje klase, ali može da bude pokazivač ili upućivač na primerke sopstvene klase. Konstruktori mogu da imaju podrazumevane parametre. Ne daju nikakvu vrednost funkcije, šta više prilikom njihovog definisanja nije dozvoljena upotreba ni reči `void` za označavanje tipa rezultata.

Evo primera za definisanje klase s konstruktorima:

```
class T {
    char a; int b;
public:
    T ();                                // Bez parametara.
    T (int);                             // S parametrom.
    T (char, char=0);                    // S podrazumevanim parametrom.
    T (T);                               // GREŠKA: ne može primerak tekuce klase.
    T (T*);                             // Pokazivač na tekucu klasu može.
    T (T&);                            // Upućivač na tekucu klasu može.
};
```

Kao što se iz ovog primera vidi, jedina razlika pri deklarisanju konstruktora u odnosu na obične funkcije i metode je nedostatak oznake tipa rezultata. Naravno, i konstruktori mogu da se definišu u sastavu definicije klase u kom slučaju se i za njih podrazumeva da su ugradene metode.

3.4.2 Pozivanje konstruktora

Konstruktori se pozivaju automatski u momentima stvaranja stalnih i prolaznih objekata u naredbama za definisanje podataka. Tada je potrebno iza identifikatora objekta navesti inicijalizator čiji je opšti oblik:

```
( izraz , izraz , ... )
= izraz
= { vrednost , vrednost , ... }
```

ili

ili

Vrednosti `izraza` predstavljaju argumente koji će da se prenose u odgovarajući konstruktor. U slučaju da se inicijalizator sastoji samo od jednog izraza može da se koristi oblik s operatorom `=`. Ta notacija je, formalno, ista kao ona koja se koristi u jeziku C za inicijalizaciju standardnih prostih tipova podataka. U jeziku C++, naravno, time mogu da se pokreću znatno složeniji postupci za inicijalizaciju.

U slučaju preklopnih konstruktora izbor među njima se vrši na osnovu broja i tipova argumenata na mestima pozivanja, kao i u slučaju običnih funkcija s preklopnim imenima. Greška je ako se ne pronađe takav konstruktor.

Treći oblik inicijalizatora (s vitičastim zagradama `():`) može da se koristi samo za objekte klasa bez konstruktora. Taj oblik je postojao i u jeziku C (videti odeljak 1.3.2) za inicijalizaciju struktura i unija. Vrednosti u nizu vrednosti moraju da budu konstantni izrazi.

Konstruktori se pozivaju automatski i prilikom dodeljivanja memorije u dinamičkoj zoni operatorom `new`. Početna vrednost u opštem obliku izraza za dodeljivanje memorije u odeljku 2.3.4, u stvari, predstavlja prvi od navedenih oblika inicijalizatora. Dakle, unutar oblih zagrada može da bude niz izraza čije vrednosti će da se proslede odgovarajućem konstruktoru klase za čiji primerak se dodeljuje memorija.

Konstruktori mogu da se pozivaju i eksplisitno, kao i druge funkcije, u sastavu izraza. Rezultat je tada bezimeni primerak klase koji se smatra privremenim objektom. Na primer, za slučaj klase T može da se napiše `T(5, y)`, pod pretpostavkom da postoji konstruktor s dva parametra odgovarajućih tipova.

Eksplisitno pozivanje konstruktora se dešava znatno ređe od automatskog pozivanja.

Evo primera za definisanje nekoliko objekata tipa T iz primera u odeljku 3.4.1:

```
T a;           // Poziva se T () .
T b = 15;      // Poziva se T (int).
T c ('0');     // Poziva se T (char, char).
T d ('0', 'a'); // Poziva se T (char, char).
T e (&a);       // Poziva se T (T*).
T f (a);        // Poziva se T (T): kopira se a u f;
T g = a;        // - može i ovako da se piše.
T h = ('0', 15); // GREŠKA: treba koristiti konstruktor.
```

3.4.3 Definisanje konstruktora

Konstruktori se definišu na sličan način kao i obične funkcije, s tim da mogu da se navedu i inicijalizatori za pojedina polja klase. Opšti oblik definicije konstruktora je:

`Klase (parametri) : inicijalizator , ... , inicijalizator telo`

Kao i prilikom definisanja bilo koje funkcije ili metode, pored identifikatora konstruktora (`Klase`, koji mora da bude jednak identifikatoru klase za koju se definiše konstruktor) i niza parametara (`parametri`) unutar para oblih zagrada, mora da se navede i telo konstruktora (`telo`). Može da ne bude nijednog parametra, ali obile zagrade `(())` moraju da se navedu. Telo konstruktora može da bude prazno, ali par vitičastih zagrada `({})` mora da se navede. Kao što je već rečeno, konstruktori nemaju vrednost funkcije, tako da ne postoji oznaka tipa funkcije.

U slučaju da je definicija konstruktora izvan definicije klase, neophodno je pomoću operatora za razrešenje dosega `(::)` proširiti doseg klase na tu definiciju, isto kao i prilikom definicije običnih metoda.

Specifičnost definisanja konstruktora je u mogućnosti navođenja niza inicijalizatora između niza parametara i tela konstruktora. Postojanje inicijalizatora označava se s dve tačke `(:)` između zagrade na kraju niza parametara. Ako se ne navede nijedan inicijalizator treba izostaviti i dve tačke.

3.4.3 Definisanje konstruktora

Opšti oblik inicijalizatora je:

`polje (izraz , izraz , ... , izraz)`

`polje` je identifikator pojedinačnog polja koje želi da se inicijalizuje, a niz `izraza` predstavlja argumente konstruktora iz klase čiji je `polje` primerak. Ako je `polje` nekog od prostih tipova (nije klasa ili struktura), sme da se navede samo jedan `izraz` odgovarajućeg tipa.

U pojedinim izrazima parametri konstruktora mogu da se koriste kao operandi.

Treba posebno naglasiti da ne mogu da se navedu inicijalizatori za zajednička polja klase, jer oni se ne stvaraju prilikom stvaranja primeraka klase. Oni postoje još pre stvaranja prvog primerka klase. Nema, međutim, nikakve prepreke da se vrednosti zajedničkih polja promene u telu konstruktora.

Evo prve definicije nekih od konstruktora za klasu T iz primera u odeljku 3.4.1:

```
T::T () : a(0), b(1) {}           // ili ...
T::T () { a = 0; b = 1; }
T::T (int i) : a (i), b (2*i) {}   // ili ...
T::T (int i) { a = i; b = 2 * i; } // ili ...
T::T (int i) : b (2*i) { a = i; }
```

Prve dve naredbe definisu konstruktor bez parametara pomoću inicijalizatora, odnosno pomoću dodela vrednosti u telu konstruktora. U odsustvu parametara izrazi se svode na konstantne izraze.

Poslednje tri naredbe predstavljaju razne kombinacije korišćenja inicijalizatora i dodelе vrednosti u telu funkcije za konstruktor s jednim parametrom. Parametar konstruktora se koristi kao operand u izrazima za inicijalizaciju.

3.4.4 Konstruktori posebne namene

Među svim mogućim konstruktorma postoje neki koji imaju specifičnu namenu i koji se automatski koriste u odgovarajućim trenucima.

3.4.4.1 Podrazumevani konstruktor

Ako u klasi postoji konstruktor koji može da se poziva bez argumenata (koji nema nijedan parametar ili kod kojeg svi parametri imaju podrazumevane vrednosti), pozivaće se kad god se stvara objekat bez inicijalizatora. Zato se takav konstruktor naziva podrazumevani konstruktor. Postojanje podrazumevanog konstruktora je garancija da će svi primerci date klase biti inicijalizovani, čak i u slučajevima kada programer to, slučajno ili namerno, ne traži eksplisitno.

Ako u nekoj klasi nije napisan nijedan konstruktor, podrazumevani konstruktor se generiše s praznim telom i to kao javan konstruktor. Taj konstruktor može da obezbedi delimičnu, a ponekad i potpunu inicijalizaciju objekata (videti odeljak 3.4.5). Ako je u klasi definisan bar jedan konstruktor podrazumevani konstruktor se ne generiše. Možda ta vrsta inicijalizacije u takvoj klasi ne odgovara. Nema prepreke, naravno, da se eksplisitno definise podrazumevani konstruktor praznog tela.

Pošto su neinicijalizovani pokazivači vrlo opasni, u klasama koje poseduju pokazivačka polja, postojanje nepraznog podrazumevanog konstruktora može da se smatra obaveznim. Ako ništa drugo, potrebno je inicijalizovati sva pokazivačka polja nulama.

Na primer:

```
class Niz {
    double* a; int n;
public:
    Niz () { a = 0; n = 0; }
    ...
};
```

Postoje, doduše retki, slučajevi kada nema smisla stvaranje primeraka neke klase uopšte (na primer, svi članovi su zajednički) ili ne izvan klase (jer korisnik vidi samo klasu u celini, a ne i pojedine njene primerke). U takvoj klasi se dešava da nema potrebe ni za jednim konstruktorom. Ako bi se klasa ostavila bez i jednog konstruktora automatski bi se generisao javan podrazumevani konstruktor i ipak bi mogli da se izvan klase stvaraju primerci klase. Da bi se to sprečilo potrebno je definisati privatani podrazumevani konstruktor s praznim telom. U tom slučaju izvan klase neće moći da se pozove taj konstruktor, čak ni automatski, i time se onemogućava stvaranje objekata bez inicijalizatora. Unutar klase, naravno, projektant klase mora da vodi računa o tome da li sme ili ne da stvara primerke klase.

3.4.4.2 Konstruktor kopije

Konstruktor kopije je konstruktor koji može da se poziva s jednim argumentom tipa svoje klase (koji ima jedan parametar ili kod kojeg svi parametri osim prvog imaju podrazumevane vrednosti). Taj parametar, mora biti upućivač na primerke, jer konstruktor ne može da ima parametar tipa svoje klase. Odgovarajući argument prilikom pozivanja, naravno, može da bude objekat ili upućivač na objekat.

Konstruktor kopije služi za inicijalizaciju primeraka klase kopijom sadržaja drugog objekta istog tipa. Drugim rečima, koristi se kada je inicijalizator objekat istog tipa kao i objekat koji se stvara.

Ako u klasi nije definisan konstruktor kopije on se generiše automatski tako da kopira sva polja objekta koji se navodi kao inicijalizator u objekat koji se stvara. Generisani konstruktor je javan. Skreće se pažnja da automatski generisan konstruktor kopije ne sprečava eventualno automatsko generisanja podrazumevanog konstruktora.

Ako su neki od polja tipa (drugih) klasa, za kopiranje će se pozivati konstruktori kopija tih klasa.

Ako su neka od polja pokazivači, generisani konstruktor će kopirati samo pokazivače, a neće praviti kopije pokazivanih podobjekata.

Konstruktor kopije treba da obezbedi kopiranje celog (složenog) objekta, a ne samo polje klase. Zbog toga za klase s pokazivačkim poljima automatski generisani konstruktor kopije nije zadovoljavajući.

Na primer:

```
class Niz {
    double* a; int n;
public:
    ...
    Niz (const Niz& niz) {
        if (niz.a) {
            a = new double [n = niz.n];
            for (int i=0; i<n; i++) a[i] = niz.a[i];
        } else { a = 0; n = 0; }
    }
    ...
};
```

Skreće se pažnja na to da se konstruktor kopije, pored u naredbama za definisanje podataka i u izrazima s operatorom **new**, automatski poziva i prilikom inicijalizacije parametra funkcije, koji je klasnog tipa, kopijom sadržaja odgovarajućeg argumenta. Ta činjenica se vrlo često ispušti izvida prilikom odlučivanja da li da se za neku klasu napravi konstruktor kopije ili ne.

Smatra se da dobro opremljena klasa, koja među poljima ima i pokazivače, mora obavezano da ima i konstruktor kopije koji nije generisan automatski.

Ako u nekoj klasi nema smisla stvarati kopije objekata potrebno je definisati privatani konstruktor kopije s praznim telom. U tom slučaju izvan klase neće moći da se pozove taj konstruktor, čak ni automatski. Unutar klase, naravno, projektant klase mora da vodi računa o tome da ne dode do inicijalizacije novog objekta kopijom već postojećeg.

3.4.4.3 Konstruktor konverzije

Konstruktor konverzije je konstruktor koji može da se pozove s jednim argumentom čiji tip nije jednak posmatranoj klasi. Naravno, konstruktor može da ima i dodatne parametre s podrazumevanim vrednostima.

Ako u klasi T postoji konstruktor T::T(U&), gde je U klasa ili standardni tip čiji je jedan primerak u, vrednost izraza T(u) je privremen (bezimen) objekat tipa T. Zbog toga, taj izraz predstavlja konverziju iz tipa U u tip T.

Na primer:

```
class Kompl {
    double re, im;
public:
    // Istovremeno: podrazumevani konstruktor,
    //                 konstruktor konverzije i
    //                 „običan“ konstruktor:
    Kompl (double r=0, double i=0) { re = r; im = i; }
    ...
    Kompl z (1.2);           // Konverzija realnog broja.
```

Prilikom pozivanja funkcija, u slučaju kad tipovi argumenata i odgovarajućih parametara nisu isti, primenjuje se automatska konverzija iz tipa argumenta u tip parametra. Očigledno, konstruktor konverzije u takvim slučajevima treba da pripada klasi parametra, a njegov argument klasi argumenta. Automatska konverzija tipa je obradena u odeljku 4.4.1.

Naglašava se da konstruktorima konverzije može da se konvertuje iz standardnih tipova u klasne tipove, ali ne i obrnuto. Naime, konstruktorom ne može da se konvertuje u neki od standardnih tipova, jer standardni tipovi nisu klase za koje bi programer mogao da definiše konstruktore za konverziju.

Konstruktor konverzije, kao svi ostali konstruktori, mogu da se pozivaju i eksplisitno. Radi usaglašavanja notacije za jasno definisane konverzije tipova, konverzija podatka u tipa U u objekat tipa T konstruktorom konverzije u klasi T može da se ostvari bilo kojim od izraza T(u), (T) u i static_cast<T>(u).

Nije svako stvaranje objekta na osnovu jednog argumenta konverzija tipa. Automatsko pozivanje odgovarajućeg konstruktora s jednim parametrom radi konverzije tipa tada ukazuje na logičku grešku u programu.

Na početku definicije ili deklaracije konstruktora koji može da se poziva s jednim argumentom, a ne predstavlja konstruktor konverzije, potrebno je dodati modifikator **explicit**. Prevodilac će prijaviti grešku ako dode u situaciju da treba da pozove takav

konstruktor za automatsku konverziju tipa. Automatski će da pozove samo za inicijalizaciju novih objekata u naredbama za definisanje podataka i u izrazu `new`. Naravno, eksplisitno pozivanje konstruktora je uvek dozvoljeno.

Na primer:

```
class Niz {
    double* a; int n;
public:
    explicit Niz (int d) { // Nije konverzija tipa.
        a = new double [n = d];
    }
    ...
Niz p (10);           // Stvaranje niza.
Niz* q = new Niz (8);
void f (Niz);
f (5);               // GREŠKA: Konverzija int u Niz!
f (Niz(5));          // Ovako može: eksplisitni poziv konstr.
```

3.4.5 Tok izvršavanja konstruktora

Stvaranje objekata podrazumeva dodelu memorijskog prostora potrebne veličine i inicijalizaciju polja. Neinicijalizovani objekat nije objekat u pravom smislu te reči, jer ne poseduje sve odlike svoje klase.

Polja klase mogu i sami da budu primerci nekih (drugih) klase.

U toku stvaranja objekta najpre se inicijalizuju polja klasnih tipova po redosledu navođenja u definiciji klase. Ako za neko polje postoji inicijalizator u definiciji konstruktora klase objekta koji se stvara, pozivaće se odgovarajući konstruktor iz klase polja. Ako ne postoji inicijalizator, pozivaće se podrazumevani konstruktor iz klase polja. Greška je ako u klasi polja ne postoji bar automatski generisani podrazumevani konstruktor. Naglašava se da se polja inicijalizuju po redosledu navođenja u klasi, bez obzira na redosled navođenja eventualnih inicijalizatora.

Posle inicijalizacije polja klasnih tipova inicijalizuju se polja standardnih tipova za koje postoji inicijalizator u definiciji konstruktora. Redosled je, opet, po redosledu navođenja polja u definiciji klase, a ne inicijalizatora u definiciji konstruktora.

Na kraju, izvršava se telo konstruktora. Menjanje vrednosti polja unutar tela konstruktora se, međutim, ne smatra inicijalizacijom polja već dodelom vrednosti. Razlika između inicijalizacije i dodele vrednosti je detaljnije razmatrana u odeljku 4.5.1.

Opisani način stvaranja objekata opravdava automatsko generisanje podrazumevanog konstruktora s praznim telom. Takav konstruktor će, naime, obezbediti inicijalizaciju polja klasnih tipova njihovim podrazumevanim konstruktorma. U slučajevima kada su sva polja klasnih tipova ništa više i nije potrebno za inicijalizaciju celokupnog objekta. Eventualna polja standardnih tipova, naravno, остаće neinicijalizovani, ako se inicijalizacija svodi na korišćenje automatski generisanog podrazumevanog konstruktora.

Pošto se polja klasnih tipova obavezno inicijalizuju, makar podrazumevanim konstruktorma, pre izvršavanja tela konstruktora, izričito se preporučuje njihovo postavljanje inicijalizatorima pre tela konstruktora, a ne dodelama vrednosti u telu konstruktora. Time se uštedi nepotrebna inicijalizacija vrednostima koje će odmah biti zamenjene novim vrednostima.

3.4.5 Tok izvršavanja konstruktora

Za polja neklasnih tipova svejedno je da li se njihove vrednosti postavljaju inicijalizatorima ili dodelama vrednosti, jer u odsustvu inicijalizatora s njima se ništa ne radi u fazi inicijalizacije objekta (pre izvršavanja tela konstruktora).

Među poljima mogu da budu i nepromenljiva (`const`) i upućivačka (`Klasa&`) polja. Kao što je za obične nepromenljive podatke i upućivače postavljanje vrednosti moguće samo inicijalizacijom, tako je i za polja to moguće samo inicijalizatorima navedenim u definiciji konstruktora, a ne u naredbama za dodelu vrednosti unutar tela konstruktora. Ovo važi za sve tipove polja, uključujući i standardne tipove.

3.4.6 Inicijalizacija nizova objekata

Kao i u slučaju nizova standardnih tipova, prilikom definisanja nizova objekata klasnih tipova mogu da se navedu inicijalizatori u obliku niza konstantnih izraza unutar para vitičastih zagradki (`{ }`). Ti izrazi mogu da budu i pozivi konstruktora s konstantnim argumentima. Ako je neki od vrednosti standardnog tipa pozivaće se konstruktor s jednim parametrom. Na primer:

```
T t = { }; t ('a', '0'); // Za prvi element poziva se T (int).
```

U odsustvu inicijalizatora elementi nizova objekata mogu da se inicijalizuju samo pomoću podrazumevanog konstruktora. Greška je ako takav konstruktor ne postoji. Ako postoji, pozivaće se po jednom za inicijalizaciju svakog elementa niza. Redosled inicijalizacije je po rastućim vrednostima indeksa (tačnije, po rastućim adresama elemenata). Ovo je važno ako konstruktor daje i bočne efekte!

Ako se memorija dodeljuje nizu objekata pomoću operatora `new`, inicijalizator ne sme da se navodi i koristiće se podrazumevani konstruktor za inicijalizaciju elemenata niza. Naravno, opet po rastućim vrednostima indeksa elemenata. Greška je ako ne postoji podrazumevani konstruktor.

3.4.7 Konstruktori za proste tipove podataka

Radi što jednoobraznijeg korišćenja prostih (standardnih i nabrojanih) i klasnih tipova, smatra se da i za proste tipove podataka postoje podrazumevani konstruktori, konstruktori kopije i konstruktori konverzije.

Podrazumevani konstruktor postavlja vrednost novog podatka na nulu odgovarajućeg tipa. Za razliku od podrazumevanih konstruktora za klase, podrazumevani konstruktori za proste tipove se ne pozivaju nikada automatski, tako ni za elemente nizova. Eksplisitno mogu da se pozivaju samo za tipove čiji se nazivi sastoje samo od jednog leksičkog simbola (reči). Na primer:

```
int i = int(), j;           // i=0, j=?
double a = double();        // a=0.0
short int k = short int();  // GREŠKA: ime tipa od dve reči.
typedef int* Pi;
Pi pi = Pi();              // pi = NULL
```

Konstruktor kopije postavlja vrednost novog podatka na vrednost izvorišnog podatka. Može da se poziva i automatski pri inicijalizaciji podataka. Na primer:

```
int i (5), j = 6, k = int (i);
```

Konstruktor konverzije postavlja vrednost novog podatka na vrednost izvorišnog podatka prema pravilima konverzije za standardne tipove podataka. Može da se poziva i automatski pri inicijalizaciji podataka. Na primer:

```
int i (5.1), j = 6.2, k = int (7.7);
```

Konstruktor konverzije za proste tipove podataka može da se koristi umesto operadora za konverziju (operator *cast*) koji je poznat iz jezika C. Na primer, konverzija tipa u tip **int** pored načina (**int**)*x*, može da se piše i kao **int**(*x*). Kao i u slučaju podrazumevanog konstruktora, ova notacija je moguća samo ako se naziv tipa sastoji samo od jednog leksičkog simbola. Na primer, (**unsigned long**)*x* ne može da se piše kao **unsigned long**(*x*). Isto tako, (**int***)*x* ne može da se piše kao **int ***(*x*), a ni bez razmaka kao **int*(x)**, jer se oznaka tipa sastoji od dva leksička simbola: službene reči **int** i modifikatora *****.

3.5 Destruktori

Objekte koji više nisu potrebni treba ispravno uništiti. Pod tim se, prvenstveno, podrazumeva oslobađanje memoriskog prostora koji je bio dodeljen prilikom stvaranja objekata.

Uništavanje objekata se vrši:

- za trajne objekte automatski na kraju programa,
- za prolazne objekte automatski u momentima napuštanja doseg-a njihovih identifikatora,
- za privremene objekte automatski čim je to moguće (obično po završetku izračunavanja izraza u toku kojeg su stvoreni) i
- za dinamičke objekte na zahtev programera pomoću operatara **delete** ili automatski na kraju programa.

Memorijski prostor koji je objektima dodeljen za polja klase, na osnovu definicije klase, oslobada se automatski, bez intervencije programera.

Oslobađanje eventualnog dodatnog memoriskog prostora u dinamičkoj zoni mora da predviđi programer. Kao i pri stvaranju objekata uništavanje može da se predviđi samo za trajne, prolazne i dinamičke objekte, a ne i za privremene objekte. Pored toga, programer može i da propusti da izda zahtev za uništavanje nekih objekata.

Mehanizam koji jezik C++ nudi za zagarantovano uništavanje svih kategorija objekata zasniva se na specijalnim metodama klase koje se nazivaju **destruktori**. Ako u nekoj klasi postoji destrukturor biće sigurno automatski pozvan kad god se uništava neki objekat te klase.

Posle primene destruktora na neki objekat on gubi obeležja svoje klase. Objekat se time opet pretvorí u parče „neinteligentne“ memorije, kakav je bio pre primene konstruktora.

Destruktori su metode klase koje imaju iste identifikatore kao i klase kojima pripadaju s dodatkom znaka ~ na početku (~T() za klasu T). Koriste se za automatsko uništavanje primeraka svojih klasa.

Data klasa može da ima samo jedan destrukturor. Destruktori ne mogu da imaju parametre. Ne daju nikakvu vrednost funkcije, šta više prilikom njihovog definisanja nije dozvoljena upotreba ni reči **void** za označavanje tipa rezultata.

Destruktori se pozivaju automatski kad god treba uništiti neki objekat.

Kao što se negenerisani podrazumevani konstruktor i konstruktor kopije smatraju obaveznim sastavnim delovima klase s pokazivačkim poljima, tako i destruktori treba da postoje u takvim klasama. To garantuje sigurno oslobađanje memorije u dinamičkoj zoni koja je dodeljena objektu u toku stvaranja i menjanja sadržaja. Upravo zbog destruktora je vrlo važno da ovakvi objekti budu propisno inicijalizovani, makar i nultim vrednostima pokazivača (operator **delete** primenjen na pokazivač nulte vrednosti je bezopasan!).

Na primer:

```
class Niz {
    double* a; int n;
public:
    ...
    ~Niz () { delete [] a; }
    ...
};
```

Ako objekat koji se uništava ima polja tipa klasa s destruktorma prvo se izvršava telo destruktora posmatranog objekta, pa tek onda destruktori klasa tih polja. Polja se uništavaju po suprotnom redosledu od njihovog navođenja u definiciji klase.

Ako u klasi nije definisan destrukturor on se automatski generiše s praznim telom. To obezbeđuje uništavanje eventualnih polja klasnih tipova pozivanjem njihovih destruktora. Generisani destruktur je javan. Naglašava se da generisani destruktur nije zadovoljavajući za klase s pokazivačkim poljima.

U slučaju uništavanja nizova, destruktori se pozivaju za svaki element niza po jednom. Elementi niza se uništavaju po suprotnom redosledu od njihovog stvaranja. Drugim rečima, po opadajućem redosledu vrednosti indeksa (preciznije, po opadajućim adresama) elemenata.

Destruktori mogu da se pozivaju i eksplisitno.

Pošto se izraz ~T() smatra primenom operatora ~ na rezultat konstruktora T() (dodata tumačenja operatorima za klasne tipove objašnjena je u poglavljju 4), destruktur uvek mora da se poziva notacijom u kojoj jedino dozvoljeno tumačenje za ~T je da je to identifikator. Za to postoje tri mogućnosti: T::~T(), t.~T() ili pt->~T(), gde su t objekat tipa T i pt pokazivač na objekte tipa T. Ovo znači i to, da unutar metoda mora da se piše this->~T().

Eksplisitno pozivanje destruktora treba maksimalno izbegavati, jer posle toga objekat će i dalje postojati, što nije prirodno s obzirom na osnovnu ulogu destruktora. Ako se destruktur pozove eksplisitno, mora da ostavi objekat u ispravnom „praznom“ stanju, jer program mora da bude u stanju i dalje da ga koristi sve do konačnog uništenja. Ako se destruktur poziva automatski, pri uništavanju objekta, o takvim detaljima ne treba voditi računa.

Umesto eksplisitnog pozivanja destruktora, bolje je napraviti metodu za „pražnjenje“ objekta i nju pozivati kada je to potrebno. U tom slučaju, destruktur može da se sastoji samo od pozivanja te metode. Na primer:

```
class Niz {
    double* a; int n;
public:
    ...
    void brisi () { delete [] a; a = 0; n = 0; }
    ~Niz () { brisi (); }
    ...
};
```

Program 3.1 prikazuje primer klase niski s konstruktorima i destruktorm.

```

class Tekst {
    char* niz; // Pokazivač na sam Tekst.
public:
    Tekst () : niz = 0; // Inicijalizacija praznog niza.
    Tekst (const char*); // Inicijalizacija nizom znakova.
    Tekst (const Tekst&); // Inicijalizacija tipom Tekst.
    ~Tekst ();
    void pisi () const; // Uništavanje objekta tipa Tekst.
};

#include <cstring>
#include <iostream>
using namespace std;

Tekst::Tekst (const char* t)
{
    niz = new char [strlen(t)+1]; strcpy (niz, t);
}

Tekst::Tekst (const Tekst& t)
{
    niz = new char [strlen(t.niz)+1]; strcpy (niz, t.niz);
}

Tekst::~Tekst () { delete [] niz; niz = 0; }

void Tekst::pisi () const { cout << niz; }

int main ()
{
    Tekst pozdrav ("Pozdrav svima"); // Poziva se Tekst (char*).
    Tekst a = pozdrav; // Poziva se Tekst (Tekst&).
    Tekst b; // Poziva se Tekst ().

    cout << "pozdrav = "; pozdrav.pisi (); cout << endl;
    cout << "a = "; a.pisi (); cout << endl;
} // Ovde se poziva destruktur za sva tri objekta.

```

Program 3.1 – Klasa s konstruktorima i destruktorm (tekst1.C)

Pošto su niske međusobno vrlo različitih dužina, objekti za njihovo predstavljanje obično sadrže samo pokazivač na sam tekst. To je učinjeno i u ovom primeru.

Jedino polje klase **Tekst** je pokazivač na sam niz znakova. Zbog toga se prilikom inicijalizacije (konstruktorom **Tekst (char*)**) dodeljuje memorijski prostor potrebne veličine i kopira se niz znakova u tako dodeljenu memoriju. Zadatak destruktora (**~Tekst ()**) je da oslobodi taj memorijski prostor prilikom uništavanja objekta.

Podrazumevani konstruktor (**Tekst ()**) osigurava da svaki objekat tipa **Tekst** bude inicijalizovan do te mere da bi kasnije smeо da se na njega primeni destruktur. Naime, posledice primene operatora **delete** na pokazivač slučajnog sadržaja su nepredvidljive, ali primena operatora **delete** na pokazivač **NULL (0)** uvek je bezbedna.

Naredba **niz=0**: u destrukturu nije potrebna ako se destruktur poziva samo automatski prilikom uništavanja objekata. Međutim, poželjno je da objekat bude ostavljen u „ispravnom praznom“ stanju zbog eventualnog eksplicitnog pozivanja destruktora. Tada će objekat, verovatno, i dalje postojati pa bi moglo da zasmeta ako pokazivačko polje pokazuje nešto u dinamičkoj zoni memorije što više ne postoji.

3.5 Destruktori

Konstruktor kopije (**Tekst (Tekst&)**) samo treba da prekopira celokupan sadržaj svog pravog parametra u objekat na koji pokazuje skriveni parametar **this** (to je objekat koji se inicijalizuje). Pošto se stvara nov objekat, ne prepostavlja se ništa o zatečenom sadržaju parčeta memorije koje se inicijalizuje.

U glavnoj funkciji na kraju programa 3.1 prvo se stvara objekat pozdrav tipa **Tekst** koji se inicijalizuje običnom niskom (**char***). Tu će se pozivati konstruktor **Tekst (char*)** koji je, u stvari, konverzija iz tipa **char*** u tip **Tekst**. Posle toga, objekat **Tekst (Tekst&)** se inicijalizuje kopijom sadržaja objekta pozdrav pomoću konstruktora a tipa **Tekst** se inicijalizuje kopijom sadržaja objekta tipa **Tekst**. Dok se objekat b podrazumevanim konstruktorom **Tekst ()** inicijalizuje kao „prazan“ objekat. Ono što je bitno, polje **niz** u njemu se postavlja na nulu, pa automatsko pozivanje destruktora (b. **~Tekst ()**) na kraju programa neće da napravi nikakvu štetu.

3.6 Konstante klasnih tipova

Konstante klasnih tipova ne postoje, ali ako su konstruktori neke klase dovoljno jednostavni da se ugrade u kôd, moguće je dobiti efekat kao da postoje.

Na primer, ako se za klasu kompleksnih brojeva

```

class Kompl {
    double re, im;
public:
    Kompl (double r=0, double i=0) { re = r; im = i; }
    ...
};

```

napiše **Kompl (3, 4)** kao operand u nekom izrazu s kompleksnim brojevima, vrednosti 3 i 4 biće, najverovatnije, ugrađene u same mašinske naredbe računara kao neposredni argumenti. Obično se tako postupa i s konstantama standardnih tipova, kao što su **int** ili **double**.

S druge strane, gde god se u programu očekuje konstanta klasnog tipa može da se koristi konstanta standardnog tipa, pod uslovom da u klasi postoji konstruktor koji može da konvertuje tu konstantu. Kao i u prethodnom slučaju, ako je taj konstruktor ugrađen, dobije se program isto tako efikasan kao da je prevodilac u program ugradio konstantu klasnog tipa.

3.7 Zajednički članovi klase

Stavljanjem modifikatora **static** na početak definicije ili deklaracije nekog člana date klase, taj član postaje zajednički za sve objekte te klase koji će biti stvoreni u toku izvršavanja programa. Članovi koji nisu zajednički nazivaju se *pojedinačni članovi klase*. Očigledno, bez eksplicitnog zahteva članovi klase su pojedinačni članovi.

U slučaju zajedničkog polja postoјe samo jedan primerak tog člana, bez obzira na broj objekata tipa te klase. Pristup tom polju u sastavu bilo kog od tih objekata označava pristup fizički istoj memorijskoj lokaciji. Menjanjem vrednosti zajedničkih polja deluje se na stanje svih objekata te klase, a ne samo na stanje jednog od njih. Zajednička polja su

specijalni globalni podaci čije vrednosti mogu da se menjaju samo na način koji odgovara opisu klase (ako su polja privatna).

Dok definisanje primerka date klase podrazumeva definisanje svih pojedinačnih polja (dodelu memoriskog prostora i eventualnu inicijalizaciju), to nije slučaj za zajednička polja. Ona se smatraju zasebnim trajnim podacima sa spoljašnjim povezivanjem. Njihov opis unutar klase smatra se samo deklaracijom, što ne obuhvata dodelu memorije.

Zajednička polja se definišu zasebnim naredbama za definisanje podataka koje se pišu izvan definicija klasa. To podrazumeva i dodelu memorije. Pošto se definicije nalaze izvan dosega klasa kojima ta polja pripadaju, mora da se koristi operator za razrešenje dosega (:) da bi bilo označeno kojoj klasi pripadaju identifikator koji se definišu.

Prilikom definisanja zajedničkih polja mogu da se navedu inicijalizatori za određivanje početnih vrednosti. U nedostatku inicijalizatora, nule se podrazumevaju. Početne vrednosti se dodeljuju pre pozivanja funkcije `main()`.

Zajedničke metode ne poseduju pokazivač `this`. Zbog toga neposredno, navođenjem samo identifikatora člana, mogu da pristupaju samo zajedničkim članovima svojih klasa. Pojedinačnim članovima mogu da pristupaju samo za konkretnе objekte. Ti objekti mogu da budu parametri zajedničkih metoda, lokalni objekti u njima ili globalni objekti.

Pristupanje zajedničkim članovima (poljima i metodama) iz svih vrsta funkcija (metoda i globalnih funkcija) moguće je navođenjem konkretnog objekta (`objekat.član` ili `pokazivač->član`) ili bez navođenja objekta (`Klasa::član`). U metodama klase može da se koristi i samo identifikator člana (`član`). Naravno, `objekat` i `pokazivač` mogu da budu izrazi čije su vrednosti objekat odnosno pokazivač na objekat posmatrane klase. `Klasa` može da bude samo identifikator klase.

Bez obzira na navedene mogućnosti pristupanja zajedničkim članovima, izričito se preporučuje korišćenje oblika `Klasa::član`. Time se naglašava da se pristupa klasi u celini, a ne samo navedenom objektu. Ako dođe do bilo kakve promene to će da utiče na stanje cele klase, tj. svih objekata te klase.

Zajednički članovi postoje i pre stvaranja prvog objekta date klase. Razume se, tada im je moguće pristupiti samo izrazom oblika `Klasa::član`.

Zajednički članovi koriste se u slučajevima kada svi primerci neke klase čine neku logičku celinu (zbirku podataka) uboličenu u neku složenu strukturu podataka.

Kao primer može da se navede lista kod koje primerci klase čine elemente jedne liste. Zajednički član može da bude pokazivač na prvi element liste, a možda i broj elemenata u listi. Pojedinačni članovi te klase mogu da budu sadržaj elementa i pokazivač na naredni element liste.

Evo primera klase sa zajedničkim članovima:

```
class Abc {
    static int a;           // Zajedničko polje.
    int b;                 // Pojedinačno polje.
public:
    static int f();         // Zajedničke metode.
    static void g (Abc, Abc*, Abc&); // Pojedinačna metoda.
    int h (int);           // Pojedinačna metoda.
};

int Abc::a = 55;          // Definicija zajedničkog polja.
```

3.7 Zajednički članovi klasa

```
int Abc::f () {
    int i = a;
    int j = b;
    return i + j;
}

void Abc::g (Abc x, Abc* y, Abc& z) {
    int i = x.b;
    int j = y->b;
    z.b = i + j;
}

int Abc::h (int x) {
    return (a + b) * x;
}

void main () {
    int p = Abc::f ();
    int q = Abc::h (5);
    Abc k;
    int r = k.f ();
    Abc::g (k, &k, k);
    int s = k.h (7);
}
```

*// Donjatanje zajedničkog polja.
// GREŠKA: Ne može pojedinačno polje
// (this->b, a this ne postoji).*

*// Pristup pojedinačnim poljima
// objekata koji su parametri funkcije
// (vrednost, pokazivač, upućivač).*

*// Pojedinačna metoda može ravnopravno
// da koristi i pojedinačna i zajednička
// polja.*

*// Može mada još ne postoji nijedan objekat.
// GREŠKA: mora konkretan objekat.
// Stvaranje prvog objekta.
// Sada vec može i ovako (izbegavati).*

// Konkretan objekat je k.

Klasa `Abc` ima jedno zajedničko i jedno pojedinačno polje. Zajedničko polje `a` je definisano zasebnom naredbom odmah iza definicije klase. Tom prilikom je i inicijalizovan vrednošću 55.

Zajednička metoda `f()` sme da koristi zajedničko polje `a`, a ne i pojedinačno polje `b`. Za pojedinačne članove, naime, podrazumeva se pristup pomoću pokazivača `this` (tj. `this->b`), a za zajedničke metode taj pokazivač nije definisan.

Metoda `g()` je primer za mogućnosti korišćenja objekata tipa `Abc` unutar zajedničkih metoda. Konkretni objekti tog tipa mogu da budu parametri funkcije, bilo u obliku vrednosti (`x`), pokazivača na objekte (`y`) ili upućivača na objekte (`z`).

Metoda `h()` je pojedinačna metoda. Unutar takvih metoda, formalno, na ravnopravan način mogu da se koriste i pojedinačni i zajednički članovi matične klase. Pokazivač `this` koristiće se za pristup do pojedinačnih članova.

Na kraju, glavna funkcija prikazuje nekoliko primera korišćenja objekata klase `Abc`.

Zajednička metoda `f()` može da se poziva pre stvaranja bilo kog objekta klase `Abc`. Ona koristi samo zajednička polja klase, a oni postoje od samog početka programa (nارavno pod pretpostavkom da se prethodno otkloni greška koja je u navedenom tekstu namereno učinjena).

Pojedinačna metoda `h()` ne može da se poziva u drugoj naredbi glavne funkcije. Za nju mora da se navede konkretan objekat tipa `Abc`, a takav objekat još ne postoji. Iz istih razloga na tom mestu ne bi smela da se pozove ni zajednička metoda `g()`. Njoj su konkretni objekti potrebni kao argumenti.

Posle definisanja objekta `k` tipa `Abc` u trećoj naredbi, sve tri metode klase `Abc` mogu da se pozivaju za konkretan objekat. Ne preporučuje se, međutim, pozivanje zajedničke metode `f()` na način kako se pojedinačne metode moraju pozivati (`k.f()`), već se preporučuje oblik pozivanja bez pominjanja konkretnog objekta (`Abc::f()`). Funkcija `f()`, bez obzira na način pozivanja, ne može da koristi sve članove objekta `k`, već samo njegove zajedničke članove. Međutim, ti zajednički članovi ne pripadaju samo objektu `k`!

3.8 Prijateljske funkcije klase

Prijateljske funkcije neke klase su funkcije koje nisu članovi te klase ali imaju pravo pristupa do privatnih članova te klase. Prijateljske funkcije mogu da budu obične (globalne) funkcije ili da budu metode drugih klasa.

Da bi funkcija postala prijateljska funkcija neke klase, potrebno je u definiciji te klase navesti njen prototip ili definiciju sa modifikatorom `friend` na početku:

```
friend tip funkcija ( parametri ) ;
friend tip funkcija ( parametri ) blok
```

ili

Nije bitno da li se funkcija proglašava prijateljskom funkcijom u privatom ili javnom delu klase, pošto ona nije član posmatrane klase.

Ako se navede definicija funkcije, modifikator `inline` se podrazumeva, kao i za članove klase. Uprkos tome, identifikator funkcije neće imati klasni doseg, već će pripasti dosegu identifikatora cele klase. Najčešće je to datotečki doseg.

Dešava se da sve metode neke klase treba da budu prijateljske funkcije date klase. Umesto da budu navedeni prototipovi svih tih metoda sve one mogu da se učine prijateljskim funkcijama naredbom oblika:

```
friend identifikator_klase ;
friend class identifikator_klase ;
```

ili

Druga varijanta je potrebna ako prethodno još nije navedena ni definicija niti deklaracija klase čije metode treba da budu prijateljske funkcije. Naredba se, naravno, stavlja u definiciju klase kojoj te metode treba da budu prijateljske funkcije.

Prijateljske funkcije se definišu kao i bilo koje druge funkcije. Ne poseduju pokazivač `this` za klasu čije su prijateljske funkcije. Zbog toga mogu da obrađuju samo konkretnе objekte te klase. Ti objekti mogu da budu, na primer, parametri, lokalni objekti, itd. Naravno ako je prijateljska funkcija metoda neke druge klase ona poseduje pokazivač `this` za svoju klasu.

Data funkcija može da bude prijateljska funkcija većeg broja klasa istovremeno.

Nema nikakvih formalnih razloga da se da prednost ostvarivanju funkcija u obliku metoda ispred ostvarivanja u obliku prijateljskih funkcija. Međutim, metoda može da bude pozvana samo za „stvaran objekat” dok prijateljska funkcija može da bude pozvana i za podatke koji su stvoreni u toku automatske konverzije tipa (konverzija tipa za slučaj klasnih tipova objašnjena je u odeljku 4.4.1).

Druga situacija kada se koriste prijateljske funkcije je kada neka funkcija treba neposredno da pristupa članovima više klase. Na primer ako uz klase `Matrica` i `Vektor` želi da se realizuje množenje matrice s vektorom. Ako članovi klase `Vektor` nisu prijateljske funkcije klasi `Matrica`, do elemenata matrice mogu da dođu samo pozivanjem odgovarajućih javnih metoda klase `Matrica`. To, međutim, može biti vrlo neefikasno.

Na kraju, neki programeri više vole klasičnu notaciju pozivanja funkcija kao što je `f(x)`, umesto `x.f()`. Prvi način je pozivanje prijateljske funkcije za objekat `x` neke klase, a drugi pozivanje metode iste klase.

Evo primera kojim se povlači paralela između ...

cija:

```
class Alfa {
    int x;
public:
    void p (int n) { x = n; } // Smeštanje vrednosti.
    int q () { return x; } // Uzimanje vrednosti.
    friend void r (int n, Alfa& a) { a.x = n; } // Smeštanje vrednosti.
    friend int s (Alfa a) { return a.x; } // Uzimanje vrednosti.
};

#include <iostream>
using namespace std;
void main () {
    Alfa a;
    a.p (55); // Smeštanje u objekat metodom.
    int i = a.q (); // Uzimanje iz objekta metodom.
    r (55, a); // Smeštanje u objekat prijateljskom funkcijom.
    int j = s (a); // Uzimanje iz objekta prijateljskom funkcijom.
    cout << i << ' ' << j << endl;
}
```

Klasa `Alfa` ima jedno privatno polje `x`. Smeštanje neke vrednosti u to polje ili uzimanje vrednosti tog polja moguće je samo posredstvom funkcija koje su ovlašćene za to. Funkcije `p()` i `q()` te radnje izvode kao metode, a funkcije `r()` i `s()` kao prijateljske funkcije.

Pošto se radi o vrlo jednostavnim funkcijama predviđeno je da budu ugrađene funkcije. To se podrazumeva, pošto su definisane unutar definicije klase. Treba naglasiti da će prijateljske funkcije, ipak, biti globalne funkcije datotečkog dosega identifikatora.

Treba uočiti da funkcije za istu operaciju imaju po jedan parametar više kod prijateljskih funkcija nego kod metoda. To je objekat kome se pristupa. Kod metoda taj objekat se podrazumeva prilikom definicije, a pojavljuje se prilikom pozivanja metode kao prvi operand operatora `.` ili njegova adresa kao prvi operand operatora `->`.

Funkcija `p()` mora da bude tipa `void` jer nema šta da bude njena vrednost funkcije. Funkcija `r()` mogla bi se definisati, a da se postigne isti konačni efekat, i kao funkcija tipa `Alfa` s jednim parametrom tipa `int`. Dva reda, u kojima se u posmatranom primeru koristi ta funkcija, izgledala bi onda ovako:

```
friend Alfa r (int) { Alfa w; w.x = n; return w; }
a = r (55);
```

Treba naglasiti da se u ovom slučaju isti konačni efekat dobija na osetno drugačiji način. Funkcija `r()` je sada neka vrsta funkcije za konverziju tipa koja podatak tipa `int` pretvara u objekat tipa `Alfa`. Pošto su objekat `a` i funkcija `r()` istih tipova, dodela vrednosti u poslednjem redu je dozvoljena.

Ovo drugo rešenje za funkciju `r()` ima još i tu prednost da može da se koristi i u složenijim izrazima, pod uslovom da su operatori jezika C++ sposobljeni za prihvatanje operanada tipa `Alfa`. Kako se to postiže objašnjeno je u poglavljiju 4.

Pokazivači na polja klasa pokazuju na određeno polje primeraka klase. Za razliku od običnih pokazivača, koji pokazuju na neki konkretni podatak, dodelom vrednosti pokazivaču na članove klase samo se označi neki član te klase. Objekat, čijem obeleženom članu će da se pristupi, određuje se kao operand odgovarajućeg operatora u momentu pristupanja.

Pokazivač na polja klasa se definije ubičajenim naredbama za definisanje podataka, s tim da se ispred identifikatora dodaje modifikator `Klasa::*`, gde je `Klasa` identifikator klase na čije članove će da pokazuje definisani pokazivač. Oznaka tipa na početku naredbe određuje tip polja klase na koja može da pokazuje definisani pokazivač.

Za nalaženje adrese polja klase koristi se ubičajeni (unarni) operator `&` čiji operand uz identifikator, naravno, mora da se koristi i operator za razrešenje dosega `(::)`.

Poljima pomoću pokazivača na polja klasa pristupa se pomoću binarnih operatora `* ili ->*. Prvi operand operatora * treba da je objekat, a operatora ->* pokazivač na objekat date klase. Drugi operand oba operatora je identifikator pokazivača na članove klase. Oba operatora su prioriteta 14 i grupišu se sleva udesno.`

Evo primera za definisanje i korišćenje pokazivača na članove klase:

```
class Alfa { ... public: int a, b; ... }
int Alfa::*pp;
Alfa alfa, *beta;
beta = &alfa;
pp = &Alfa::a;
alfa.*pp = 1;
beta->*pp = 1;
// pp je pokazivač na int polja klase Alfa.
// Objekat i pokazivač na objekte klase Alfa.
// beta pokazuje na objekat alfa.
// pp pokazuje na polja a objekata klase Alfa.
// alfa.a = 1;
// beta->a = 1;
// beta->b = 2;
// pp pokazuje na polja b objekata klase Alfa.
// alfa.b = 2;
// beta->b = 2;
```

Pokazivač na polja klasa `pp` je definisan da može da pokazuje na bilo koje polje tipa na objekata klase `Alfa`. Posle toga, definisan je jedan objekat (`alfa`) i jedan pokazivač na objekte (`beta`) klase `Alfa`.

Posle prve dodele vrednosti pokazivaču na polja `pp`, `alfa.*pp` označava član `alfa.a`, a posle druge dodele vrednosti, doslovce isti izraz označava član `alfa.b`. Slično je i kada se za pristup koristi pokazivač na objekte `beta`.

Pokazivač na polja klasa deluje slično kao indeks kod rada s nizovima. Kao što izraz `niz[i]` označava različite elemente niza zavisno od vrednosti indeksa `i`, tako izraz `objekat.*pp` označava različita polja objekta, zavisno od vrednosti pokazivača na polja `pp`. Treba naglasiti, dok indeks može da bude proizvoljan izraz, pokazivač na polja može da bude samo identifikator!

3.10 Unutrašnje klase

Klase mogu da se definisu i unutar definicija drugih klasa. Identifikator tako definisane klase ima klasni doseg i proteže se do kraja definicije spoljašnje klase.

Doseg identifikatora članova unutrašnje klase i doseg identifikatora članova spoljašnje klase čine dva nezavisna dosega. To znači da je pristup iz spoljašnje klase do članova unutrašnje klase, kao i iz unutrašnje klase do članova spoljašnje klase, moguć samo na uobičajene načine, pomoću operatora `.. ->` ili `::`. Izuzetak je kada se u unutrašnjoj klasi koriste identifikatori tipova, nabrojanih konstanti i zajedničkih članova iz spoljašnje klase. Ti identifikatori mogu da se koriste neposredno, bez primene pomenutih operatora.

Iz unutrašnje klase ne mogu da se koriste privatni članovi spoljašnje klase, i obrnuto, iz spoljašnje klase ne može da se pristupa privatnim članovima unutrašnje klase. Izuzev, razume se, ako je spoljašnja klasa prijateljska klasa unutrašnje klase, odnosno ako je unutrašnja klasa prijateljska klasa spoljašnje klase.

Metode unutrašnje klase mogu, a zajednička polja unutrašnje klase moraju da se definisu izvan obe klase. Tada doseg prvo mora da se proširi na spoljašnju i tek posle toga na unutrašnju klasu.

Uklapanje klasa se koristi relativno retko. Unutrašnje klase su obično vrlo male, često bez ijdene metode. Tada, naravno, polja moraju da budu javni.

Evo primera uklapanja klasa:

```
class Spolja {
    class Unutra {
        static int a;
        int b;
    public:
        int c;
        void f (Spolja s);
    };
    int x;
public:
    static int y;
    void g (Unutra u);
};

int Spolja::y = 5;
void Spolja::g (Unutra u) {
    int i = u.c;
    int j = u.b;
    int k = c;
}

int Spolja::Unutra::a = 2;
void Spolja::Unutra::f (Spolja s) { // (metoda)
    int i = s.y;
    int j = y;
    int k = s.x;
    int l = x;
}

// Definicija unutrašnje klase:
// privatno zajedničko polje,
// privatno pojedinačno polje,
// javno pojedinačno polje,
// javna pojedinačna metoda.
// Članovi spoljašnje klase:
// privatno pojedinačno polje,
// javno zajedničko polje,
// javna pojedinačna metoda.
// Definicije za spoljašnju klasu:
// (zajedničko polje)
// (metoda)
// Javno polje objekta tipa Unutra.
// GREŠKA: privatno polje.
// GREŠKA: ne može direktno.

// Definicije za unutrašnju klasu.
// (zajedničko polje)
// Javno polje objekta tipa Spolja.
// Zajedničko polje objekta tipa
// Spolja.
// GREŠKA: privatno polje.
// GREŠKA: ne može direktno.
```

3.11 Lokalne klase

Klase mogu da se definišu unutar funkcija. Takve klase se nazivaju lokalne klase. Identifikator lokalne klase ima blokovski doseg. Unutar lokalne klase dozvoljeno je korišćenje samo identifikatora tipova, trajnih podataka, spoljašnjih podataka i funkcija kao i nabrojanih konstanti iz okružujućeg dosega.

Pristupanje članovima lokalne klase iz funkcije unutar koje je definisana podleže svim uobičajenim pravilima i ograničenjima.

Metode lokalne klase moraju da se definišu unutar definicije klase. Lokalna klasa ne može da ima zajedničke članove (polja i metode). Ova ograničenja treba da obeshrabre pokušaje sastavljanja složenih lokalnih klasa.

Evo primera za definisanje lokalne klase:

```
int x;
void f () {
    static int s;
    int x;
    extern int g ();
    class Lok {                                // Definicija lokalne klase.
        ...
        public:
            int h () { return x; }      // GREŠKA: x nije trajan podatak.
            int j () { return s; }      // Trajan podatak s.
            int k () { return ::x; }    // Globalan podatak x.
            int l () { return g(); }   // Spoljašnja funkcija g.
        };
        ...
    }
    Lok* p = 0;                                // GREŠKA: Lok nije u dosegu.
```

3.12 Strukture i unije

Strukture (**struct**) u jeziku C++ su klase čiji su svi članovi podrazumevano javni. Od toga može da se odstupa umetanjem oznaka **private** i **public**. Nema nikakvih drugih razlika između klasa struktura. Zbog toga termini klasni doseg i strukturni doseg mogu da se koriste kao sinonimi.

Iz prethodno rečenog sledi da i strukture mogu da imaju metode. Dalje, početni deo date strukture (pre prve oznake **private**) je javan. Iz tog razloga, od sledećih definicija prva i druga su međusobno istovetne. Međusobno su istovetne i treća i četvrta definicija.

struct A {	class A {	class A {	struct A {
 ...	 public:	 public:	 private:
private:	private:	

}	}	}	}

Preporučuje se da se za definisanje „inteligentnih“ objekata, s predviđenim radnjama nad njihovim vrednostima, koriste klase. Strukture treba koristiti samo za opisivanje strukturiranih (složenih) podataka koji nemaju osobine „objekata“ u smislu objektno orijentisanih programiranja.

3.12 Strukture i unije

Prilikom uklapanja klasa (videti odeljak 3.10), unutrašnja klasa je vrlo često struktura čiji se primerci koriste kao elementi za izgradnju složenih objekata. Na primer, kod klasa čiji su primerci linearne liste unutrašnja struktura opisuje elemente liste.

Neki programeri više vole da koriste prvu od treće definicije. Kod nje se na početku nalazi javni deo, koji je jedino interesantan za korisnika klase. Bilo bi dobro ako korisnik ne bi mogao ni da vidi sadržaj privatnog dela, pošto već ne može da ga koristi. Tim programerima se preporučuje korišćenje druge od prethodnih definicija (koja, na žalost, sadrži jednu oznaku više!).

Unije (**union**) su specifične po tome što sva polja imaju iste početne adrese. Zbog toga svakog momenta samo jedan od njih može da ima definisanu vrednost.

Unije, kao i klase i ili strukture mogu da imaju i metode uključujući i konstruktore i destruktore. Mogu da imaju privatne i javne delove, s tim da je početni deo (pre prve oznake **private**) javan. Mogućnost postojanja privatnih delova kod unija nema neku praktičnu upotrebnu vrednost, ali nema nikakvog posebnog razloga da se to zabrani.

Konstruktori omogućavaju da se unije inicijalizuju ne samo preko svojih prvih polja, što je slučaj kod unija bez konstruktora.

Problem s destruktorem kod unija je što destruktur na neki način mora da otkrije koje polje unije ima definisanu vrednost u momentu uništavanja objekta da bi ispravno uništio objekat. Za to ne može da mu se daje nikakva pomoć sa strane, kao što je to slučaj kod konstruktora.

Mogućnost postojanja konstruktora, a naročito destruktora kod unija nema neku veliku upotrebnu vrednost.

Zajednička početna adresa svih polja unija nameće određena ograničenja.

Prvo ograničenje je da unije ne mogu imati zajedničke (**static**) članove (polja i metode). Naime, dato zajedničko polje svih primeraka neke klase označava fizički isti deo memorije, tj. smešta se od iste početne adrese. Pošto se kod unija sva polja datog primerka smeštaju od iste početne adrese, postojanje zajedničkog polja dovelo bi do smeštanja svih primeraka te unije na isto mesto u memoriji. A to ne bi imalo nikakvog smisla! Ako ne postoje zajednička polja, onda ni zajedničke metode nisu potrebne.

Unije ne mogu imati polja tipa klasa s konstruktorima i destruktorma.

Kad bi se dozvolilo da neka od polja unije imaju svoje konstruktore, ti konstruktori bi se pozivali u momentima stvaranja primeraka unija. Pošto sva polja unija imaju iste početne adrese sadržaj istog dela memorije bio bi na taj način inicijalizovan više puta. Očigledno, konstruktori polja unija kvarili bi jedan drugome posao.

Sličan je problem ako su jedan ili više polja unija tipa klasa koje imaju destruktore. Bili bi izvršavani destruktori za sva ta polja, a samo jedno polje ima definisanu vrednost i samo ono može, i sme, da se uništava.

Bezimene unije ne mogu da imaju metode. Niti je dozvoljena upotreba oznaka **private** i **public**. Znači svi članovi bezimenih unija su javna polja.

Svrha unija nije sastavljanje nekih složenih tipova podataka, već postizanje uštede u utrošku memorije smeštanjem podataka različitih tipova na isto mesto, kada je to moguće. Prirodno je da unije i njihova polja budu bez metoda i da se unije koriste kao polja drugih klasa. Te klase treba da obezbede mehanizme za kontrolu pristupa i održavanje integriteta sadržaja unija.

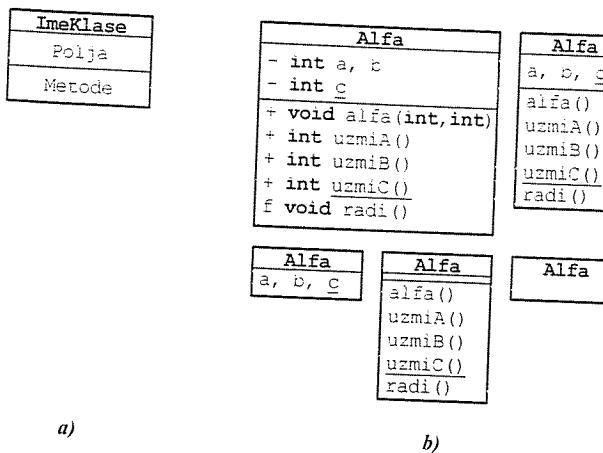
Unije su odraz vremena kada je nastao jezik C. To je bilo vreme skromnih računara kada je ušteda od nekoliko desetina bajtova nešto značila. U današnje vreme ogromnih operativnih memorija čak i u ličnim računarima, ni ušteda od nekoliko desetina kilobajtova ne znači mnogo, pa se unije vrlo retko koriste.

3.13 Dijagrami klasa

Složena obrada u objektno orijentisanom programiranju ostvaruje se interakcijom objekata. To znači da se iz metoda jedne klase koriste (javni) članovi druge klase – pristupa se poljima, pozivaju se metode.

Dijagrami klasa se koriste za grafičko prikazivanje odnosa među klasama. To su grafovi u kojima čvorovi predstavljaju klase, a linije veze među klasama.

Potpuna grafička oznaka za klasu sadrži tri deljaka (slika 3.1.a). U prvom polju se nalazi ime klase, u drugom polja klase, a u trećem metode klase.



Slika 3.1 – Oznaka za klasu u dijagramu klasa

Količina prikazane informacije može da se menja prema potrebama. Za polja mogu da se navode tipovi i imena, samo imena, a mogu i da se polja uopšte ne navode. Za metode mogu da se navode potpuni prototipovi, samo imena i tipovi rezultata, samo imena i da se ne navodi ništa.

Za sve vrste članova može da se navede pravo pristupanja (vidljivost), pri čemu + označava javne, a – privatne članove. Zajednički (**static**) članovi mogu da se označe podvlačenjem. Prijateljske funkcije nisu članovi klase, ali zbog tesne povezanosti i one mogu da se navedu u dijagramu klasa. Ako se za članove navode prava pristupanja, prijateljske funkcije se označavaju slovom *f*.

Na slici 3.1.b prikazano je pet varijanti prikazivanja sledeće klase:

```
class Alfa {
    int a, b;
    static int c;
public:
    void alfa (int, int);
    int uzmiA ();
    int uzmiB ();
    static int uzmiC ();
    friend void radi ();
};
```

3.13 Dijagrami klasa

Najopštiji odnos među klasama naziva se **asocijacija**. To podrazumeva da među poljima jedne klase postoji polje koje je pokazivač ili upućivač na objekte druge klase. Pomoću tog pokazivača ili upućivača može da se pristupa članovima druge klase. Asocijacija može da bude dvosmerna, ako se i iz druge klase može pristupiti članovima prve klase. Tada i u drugoj klasi postoji polje koje je pokazivač na objekte prve klase. Skreće se pažnja na to da u dvosmernom odnosu bar u jednoj klasi mora da bude pokazivač na drugu klasu, tj. ne mogu u obe klase da budu upućivači na drugu klasu. Zašto je to tako, biće jasno posle odeljka 3.4.5 u kome se objašnjava tok stvaranja objekata.

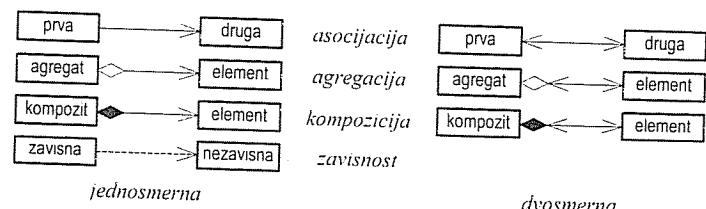
U praksi su česti slučajevi kada je odnos među klasama odnos sklopa i delova. Na primer jedna klasa je niz objekata druge klase.

Ako delovi mogu da postoje izvan sklopa, tj. mogu da se stavljaju u sklop, da se izvade iz sklopa i da se koriste i kada nisu u sklopu, odnos se naziva **agregacija**, a sklop se tada naziva **agregat**. Za uspostavljanje ovog odnosa između klasa mogu da se koriste samo pokazivači u agregatu ili u obe klase, zavisno da li je odnos jednosmeran ili dvosmeran. Samo pomoću njih može uspostavljeni odnos među objektima da se raskida kada deo treba izvaditi iz sklopa. Skreće se pažnja na to da je sadržavanje kod aggregata logičko, a ne fizičko. Delovi aggregata fizički se nalaze izvan aggregata. Njihove veličine ne utiču na veličinu aggregata. U veličinu objekata aggregata ulaze samo veličine pokazivača na delove.

Ako delovi ne mogu da postoje izvan sklopa odnos se naziva **kompozicija**, a sklop se naziva **kompozit**. Delovi mogu da se stvaraju i uništavaju istovremeno sa sklopopom ili da ih sklop stvara i uništava po potrebi, ali ne mogu da se izvade iz sklopa i da se koriste nezavisno od sklopa. Delovi u kompozitu pored pokazivačkih i upućivačkih polja mogu da se predstave i poljima tipa klase delova. Tada se delovi fizički nalaze unutar objekata kompozita, tj. u veličinu objekata kompozita uračunavaju i veličine sadržanih delova.

Ima slučajeva kada na osnovu polja dve klase koje su u odnosu ne može da se zaključi da među njima postoji bilo kakav odnos. To su slučajevi kada se objekti jedne klase koriste kao parametri metoda ili lokalni objekti u metodama druge klase. Takav odnos se naziva **zavisnost**. Kaže se da druga klasa zavisi od prve klase, jer ako se nešto promeni u prvoj klasi, ta promena može da utiče na drugu klasu. Zavisnost je uvek jednosmeran odnos. Desi li se da i prva klasa zavisi od druge, što se u praksi vrlo retko dešava, govori se o dva jednosmerna odnosa umesto jednog dvosmernog odnosa.

Na slici 3.2 prikazane su oznake za iskazivanje odnosa među klasama u dijigramima klasa.



Slika 3.2 – Oznake za odnose među klasama u dijagramu klasa

Kod jednosmernih odnosa postoji strelica koja pokazuje na klasu u kojoj se ne zna koja se klasa nalazi na drugom kraju linije. Kod dvosmerne veze postoji strelica na oba kraja linije. Ako se ne želi da se navede da li je veza jednosmerna ili dvosmerna, strelice mogu da se izostave.

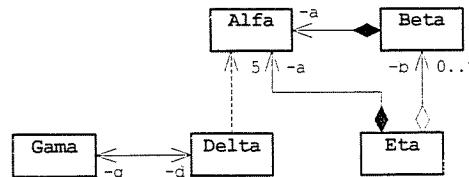
Na krajevima linija mogu da se navedu neki parametri koji bliže određuju odnos među klasama. Oznaka na jednom kraju linije uvek označava na koji način se objekti klase s tog kraja linije (klasa *B*) koriste u klasi sa suprotne strane linije (klasa *A*).

Jedan od mogućih parametara je uloga klase *B* u klasi *A*, tj. u koju svrhu se koristi klasa *B* u klasi *A*. Ta uloga se predstavlja imenom polja u klasi *A*, koji je tipa klase *B* (može biti i pokazivač ili upućivač). Uz ime može da se navede i oznaka za vidljivosti polja (+ ili -) izvan klase *A*. Pošto se oznaka uloge poklapa s imenom odgovarajućeg polja, u odeljku za polja u dijagramima klasa se navode samo polja neklasnih tipova.

Drugi mogući parametar je broj objekata klase *B* s kojima može da bude u odnosu objekat klase *A*. Moguće oznake su: 1 (tačno jedan), 0..1 (nula ili jedan), 0..*, 0..n (nula ili više), 1..*, 1..n (jedan ili više), x (tačno x, gde je x celobrojna konstanta), x..y (od x do y, gde su x i y celobrojne konstante). Ako nije bitno, ne mora da se stavlja nijedna od ovih oznaka. Odnos s više objekata može da se ostvaruje nizovima ili listama. U slučaju nizova najčešće se koriste dinamički nizovi. U nizovima ili listama češće se nalaze pokazivači na objekte nego sami objekti.

Na slici 3.3 prikazan je dijagram klasa za sledeći sistem klasa:

```
class Alfa { };
class Beta { Alfa a; };
class Gama;
class Delta { Gama* g; void m (Alfa&); };
class Gama { Delta* d; };
class Eta { Alfa a[5]; Beta* b; };
```



Slika 3.3 – Primer dijagrama klasa

Klase Beta sadrži privatno polje a tipa Alfa, što je označeno ulogom -a na strani klase Alfa. Pošto je polje tipa klase, nedvosmisleno se radi o kompoziciji. Kada klase sadrži pokazivač na drugu klasu tumačenje odnosa kao asocijaciju, kompozicija ili agregacija zavisi od logičkog smisla odnosa, a ne od formalne deklaracije.

Između klase Gama i Delta uspostavljena je dvosmerna asocijacija. Činjenica da se u obe klase zna za postojanje druge klase, označena je ulogom -g na strani klase Gama i -d na strani klase Delta. Ovo može da se tumači i ovako: klasa Delta se u klasi Gama vidi pod privatnim imenom d. Za uspostavljanje ove veze, u tekstu programa, bilo je neophodno prvo deklarisati klasu Gama, da bi se pri definisanju klase Delta znalo da Gama predstavlja ime klase radi definisanja pokazivača na objekte klase Gama. Posle toga pri definisanju klase Gama već se zna da identifikator Delta predstavlja klasu. Drugo moguće rešenje je da se prvo deklariše klasa Delta i posle toga da se definišu klase Gama pa Delta. Na slici 3.3 nije navedeno koliko objekata klase Gama i Delta učestvuje u asocijaciji.

Pošto u klasi Delta postoji metoda m(Alfa&) s parametrom tipa Alfa&, klasa Delta zavisi od klase Alfa.

3.13 Dijagrami klasa

Na kraju, klasa Eta je u odnosu s više objekata tipa Alfa i Beta. Pošto se u klasi Eta nalazi niz od 5 elemenata tipa Alfa, radi se o kompoziciji od 5 objekata tipa Alfa. To je naznačeno sa 5 na liniji veze na kraju kod klase Alfa. Za odnos s klasom Beta postoji pokazivač b. Pošto pokazivači mogu da pokazuju na pojedinačne objekte, ali i na (dinamičke) nizove objekata, na osnovu postojanja pokazivača ne može da se zaključi da li se radi o odnosu s jednim ili više objekata klase na koje pokazivač može da pokazuje. Ovde je uzeto da pokazivač b pokazuje na dinamički niz koji može da bude i prazan. Zato je za mogući broj objekata tipa Beta u odnosu s jednim objektom tipa Eta stavljena oznaka 0..*.

3.14 Zadaci

3.14.1 Obrada tačaka u ravni

Zadatak:

Napisati na jeziku C++ klasu s elementarnim operacijama nad tačkama u ravni. Napisati na jeziku C++ program za prikazivanje mogućnosti te klase.

Rešenje:

Program 3.2 predstavlja definiciju jedne jednostavne klase za obradu tačaka u ravni.

```
// Definicija klase tačaka u ravni (Tacka).

class Tacka {
    double x, y;
public:
    void postavi (double a, double b) // Postavljanje koordinata.
        { x = a; y = b; }
    double aps () const { return x; } // Apscisa tačke.
    double ord () const { return y; } // Ordinata tačke.
    double potez () const; // Odstojanje od koordinatnog početka.
    double nagib () const; // Nagib poteza u odnosu na x-osu.
    double rastojanje (Tacka) const; // Odstojanje od zadate tačke.
    Tacka najbliža (const Tacka*, int) const;
        // Najблиža tačka u nizu tačaka.
    void citaj (); // Citaj tačku.
    void pisi () const; // Piši tačku.
};
```

Program 3.2 – Definicija klase tačaka u ravni (tacka1.h)

Kao privatna polja postoje Descartesove koordinate tačke x i y.

Od javnih metoda na prvom mestu nalazi se metoda postavi() za postavljanje koordinata koje su tipa double. Ova radnja je neka vrsta konverzije tipa iz osnovnih tipova u nestandardne tipove (koje je definisao programer).

Sličnu ulogu imaju, ali u suprotnom smeru, metode aps() za nalaženje apscise i ord() za nalaženje ordinata zadate tačke. Pomoću njih može da se dolazi do delova sadržaja tačke kao složenog objekta. U širem smislu, tu mogu da se svrstavaju i metode potez() za nalaženje udaljenosti tačke od koordinatnog početka i nagib() za nalaženje

nagiba (ugla) potega u odnosu na x-osu. Ova dva podatka su, u stvari, polарne koordinate posmatrane tačke.

U klasi Tacka postoje dve metode koje među parametrima imaju objekte tog tipa. Metoda rastojanje() nalazi rastojanje između zadate tačke i tekuće tačke, dok metoda najbliza() u zadatom nizu tačaka nalazi najblizu tačku tekućoj tački. Vrednost ove metode je pronađena tačka (tačnije, kopija pronađene tačke), dakle objekat tipa Tacka.

Na kraju, metode citaj() i pisi() predviđene su za ulazni i izlaznu konverziju vrednosti objekata tipa Tacka. Metoda citaj() čita s glavnog ulaza računara koordinate x i y i smešta ih u odgovarajuća polja tekuće tačke. Metoda pisi() ispisuje na glavnom izlazu računara koordinate tekuće tačke između para oblih zagrada, međusobno razdvojenih zarezom.

Od svih metoda samo dve menjaju vrednost tekućeg objekta, postavi() i citaj(). Kod svih ostalih metoda dodat je modifikator const iza spiska parametara da bi se označilo da ne menjaju vrednost svojih tekućih objekata.

Pošto definicija klase mora da stoji na raspolaganju prevodiocu prilikom prevođenja bilo kog programskog modula u kome se koristi data klasa, tekst programa 3.2 treba da se smešta u zaglavje (datoteku .h). U ovom slučaju to je datoteka tackal.h.

Za neke od gore nabrojanih metoda u programu 3.2 navedene su definicije. Za njih se, s toga, podrazumeva da su ugrađene metode. To su one, stvarno, najjednostavnije: postavi(), aps() i ord(). Prevodi njihovog sadržaja sastoje se od svega jedne ili dve mašinske naredbe. Bilo bi vrlo neefikasno uboličavati ih u prave funkcije.

Skreće se pažnja na to da sve metode imaju i jedan skriveni parametar, adresu tekuće tačke. Koordinate x i y predstavljaju polja tekuće tačke, tj. `this->x` i `this->y`.

Definicije preostalih metoda su izdvojene u zasebnu datoteku, čiji sadržaj treba odvojeno prevoditi. Te definicije su prikazane u programu 3.3. Pošto se definicije nalaze izvan dosega klase Tacka, ispred identifikatora svake metode dodato je Tacka::.

Pošto neke od funkcija u programu 3.3 koriste matematičke funkcije, u prevodenje je uključeno zaglavje za matematičke funkcije <cmath>. Zaglavje <iostream> je potrebno zbog ulazno-izlaznih operacija u metodama citaj() i pisi(). Da bi identifikatori definisani u tim zaglavljima bili samostalno dostupni, naredbom using su uvezeni u datotečki doseg. Na kraju, u definiciju klase Tacka trebalo je staviti na raspolaganje prevodiocu iz zaglavja tackal.h.

Sve metode u programu 3.3, izuzev metode najbliza() dovoljno su male da ne bi bilo nerazumno zahtevati da budu ugrađene metode. S druge strane, pošto svaka od njih poziva bar još neku drugu funkciju, priloženo rešenje nije nerazumno neefikasno.

Metoda rastojanje() ima jedan parametar tipa Tacka, naravno pored skrivenog parametra this, koji pokazuje na objekat istog tipa i predstavlja tekuću tačku čije se rastojanje od parametra traži. Izraz x-a.x se, dakle, tumači kao (`this->x`) - (a.x).

Prvi vidljivi parametar metode najbliza() je niz tačaka koji je predstavljen pokazivačem a na objekat tipa Tacka. Kao obično, adresa i-tog elementa niza je a+i, a sam element može da se označi sa a[i]. U toj metodi definiše se i lokalni objekat t tipa Tacka koji se inicijalizuje početnim elementom niza tačka (Tacka t=a[0];). Na kraju, skreće se pažnja na to da se i za metode podrazumeva tekući objekat. To znači da se pozivanje metode rastojanje(...) tumači kao `this->rastojanje(...)`.

Funkcije (metode) definisane u programu 3.3 prevede se nezavisno. Korisniku klase Tacka dovoljno je isporučiti samo dobijeni prevedeni oblik, izvorni tekst mu nije potreban.

3.14.1 Obrada tačaka u ravni

```
// Definicije metoda klase Tacka.

#include <cmath>
#include <iostream>
using namespace std;
#include "tackal.h"

// Odstojanje tekuće tačke od koordinatnog početka.
double Tacka::poteg () const { return sqrt (x*x + y*y); }

// Nagib potega u odnosu na x-osu.
double Tacka::nagib () const { return (x==0 && y==0) ? 0 : atan2(y,x); }

// Rastojanje tekuće tačke od tačke a.
double Tacka::rastojanje (Tacka a) const
{ return sqrt(pow(x-a.x,2) + pow(y-a.y,2)); }

// Najблиža tačka u nizu tačaka a u odnosu na tekuću tačku.
Tacka Tacka::najbliza (const Tacka* a, int n) const {
    Tacka t = a[0]; double r, m = rastojanje (t);
    for (int i=1; i<n; i++) if ((r=rastojanje(a[i]))<m) {m = r; t = a[i];}
    return t;
}

// Čitanje koordinata tekuće tačke s glavnog ulaza.
void Tacka::citaj () { cin >> x >> y; }

// Pisanje koordinata tekuće tačke na glavnom izlazu.
void Tacka::pisi () const { cout << '(' << x << ',' << y << ')'; }
```

Program 3.3 – Definicije metoda klase Tacka (tackal.C)

Potrebni su i dovoljni samo prototipovi koji se nalaze u zaglavlu (program 3.2). Ovo omogućava izmenu ostvarenja date klase bez potrebe za izmenama u programima koji koriste tu klasu, sve dok se ništa ne promeni u javnom delu te klase. U posmatranom primeru, polja x i y mogli bi da se zamene poljima r i fi koja bi predstavljala polarne koordinate tačke.

Korisnik klase ništa od toga ne bi primetio. To znači da ništa ne bi trebalo da se izmeni u izvornim tekstovima programa koji koriste klasu Tacka. Bez obzira na to, svi korisnikovi programi trebalo bi ponovo da budu prevedeni u slučajevima izmena u ostvarenju klase Tacka.

Program 3.4 predstavlja primer za korišćenje klase Tacka.

Posle čitanja broja tačaka n, operatorom new traži se dodela memorije u dinamičkoj zoni za niz od n objekata tipa Tacka. Rezultat operatora dodeljuje se pokazivaču niz na objekte tipa Tacka.

U ciklusu za čitanje podataka o tačkama, pozivanjem metode citaj() izrazom niz[i].citaj(), niz[i] postaje tekuća tačka (objekat) te metode.

Posle toga čitaju se koordinate jedne referentne tačke kao dva realna broja x i y od kojih se pozivom t.postavi(x,y) postavlja sadržaj tačke t. Ta tačka se u nastavku koristi kao tekuća tačka pri pozivanju metoda aps(), ord(), poteg() i nagib(). Svi ti pozivi koriste se kao operandi operatora << za ispisivanje na glavnom izlazu.

```
// Primer korišćenja klase Tacka.

#include <iostream>
using namespace std;
#include "tackal.h"

int main () {
    int n; cout << "\nBroj tacaka? "; cin >> n;
    Tacka* niz = new Tacka [n];
    cout << "Niz tacaka? "; for (int i=0; i<n; i++) niz[i].citaj ();
    double x, y; cout << "\nReferentna tacka? "; cin >> x >> y;
    Tacka t; t.postavi (x, y);
    cout << "Koordinate: (" << t.aps () << ',' << t.ord () << ")\n";
    cout << "Poteg i nagib: " << t.poteg () << ", " << t.nagib () << endl;
    Tacka w = t.najbliza (niz, n);
    cout << "\nNajbliza tacka: "; w.pisi (); cout << endl;
    cout << "Udaljenost od referentne tacke: " << t.rastojanje (w) << endl;
    delete [] niz;
}
```

Program 3.4 – Primer korišćenja klase Tacka (tackal.C)

Tačka *w* inicijalizuje se tačkom koja je u nizu tačaka *niz* najbliža tački *t*. Taj podatak dobija se kao vrednost metode *najbliza()* za tekući tačku *t*. Rezultujući sadržaj tačke *w* ispisuje se na glavnom izlazu pomoću *w.pisi()*. Zatim se korišćenjem izraza *t.rastojanje (w)* kao operand operatoru *<<* ispisuje veličina tog najmanjeg rastojanja.

Na kraju programa, iz principijelnih razloga, operatom *delete* oslobađa se memorija u dinamičkoj zoni, koja je na početku programa bila dodeljena nizu tačaka. Ovo nije neophodno pred sam kraj programa. Pri završetku programa ionako se sva dodeljena memorija u dinamičkoj zoni oslobađa automatski.

Rezultat 3.1 prikazuje primer rada programa 3.4.

```
Broj tacaka? 5
Niz tacaka? 1 2 3 4 5 1 3 2 6 3

Referentna tacka? 3 5
Koordinate: (3,5)
Poteg i nagib: 5.83095, 1.03038

Najbliza tacka: (3,4)
Udaljenost od referentne tacke: 1
```

Rezultat 3.1 – Obrada tačaka programom 3.4

3.14.2 Obrada uređenih skupova

Zadatak:

Napisati na jeziku C++ klasu za obradu uređenih skupova realnih brojeva. Napisati na jeziku C++ program za prikazivanje mogućnosti te klase.

Rešenje:

Skupovi su zbirke podataka koje ne sadrže međusobno jednake vrednosti. Obično se smatra da redosled elemenata u skupu nije bitan. U ovom zadatku je uzeto da su elementi skupa uređeni što omogućava efikasnije izvođenje osnovnih operacija nad skupovima: nalaženje unije, preseka i razlike.

Program 3.5 prikazuje definiciju klase Skup za obradu uređenih skupova realnih brojeva.

```
// Definicija klase za uredene skupove (Skup).
```

```
class Skup {
    int vel; double* niz; // Veličina i elementi skupa.
    void kopiraj (const Skup&); // Kopiranje skupa.
    void brisi () { delete [] niz; niz = 0; vel = 0; } // Pražnjenje skupa.
public:
    Skup () { niz = 0; vel = 0; } // Stvaranje praznog skupa.
    Skup (double a) { niz = new double [vel = 1]; niz[0] = a; } // Konverzija broja u skup.
    Skup (const Skup& s) { kopiraj (s); } // Inicijalizacija skupom.
    ~Skup () { brisi (); } // Uništavanje skupa.
    void presek (const Skup&, const Skup&); // Presek dva skupa.
    void razlika (const Skup&, const Skup&); // Razlika dva skupa.
    void unija (const Skup&, const Skup&); // Unija dva skupa.
    void pisi () const; // Ispisivanje skupa.
    void citaj (); // Čitanje skupa.
    int velicina () const { return vel; } // Veličina skupa.
};
```

Program 3.5 – Definicija klase uređenih skupova (skup.h)

Sadržaj skupa čine privatna polja koja predstavljaju broj elemenata skupa *vel* i pokazivač na *niz* u dinamičkoj zoni memorije za smeštanje samih elemenata skupa.

U privatnom delu klase postoje još dve pomoćne metode za kopiranje sadržaja zadatog skupa u tekući objekat *kopiraj()* i za uništavanje sadržaja tekućeg objekta *brisi()*. Njih će da pozivaju neke od javnih metoda klase. U većini klasa koje koriste dinamičko dodeljivanje memorije za smeštanje delova sadržaja, korisno je da se prave ovakve dve metode.

Treba obratiti pažnju na prenošenje skupa u metodu *kopiraj()* (i u sve kasnije metode klase Skup) pomoću upućivača (*Skup&*). Modifikator *const* označava da metoda ne menja vrednost upućivanog objekta. To je preporučljivi način prenošenja primeraka klase u funkcije umesto prenošenja objekata (*Skup*), jer se time izbegava gubitak vremena i prostora za inicijalizaciju parametra kopijom argumenta uz pomoć konstruktora kopije.

Pošto metoda za pražnjenje skupa *brisi()* može da se koristi na više mesta, a ne samo prilikom konačnog uništavanja objekata, važno je obezbediti da po povratku iz te metode tekući objekat bude ispravan prazan objekat. To je u programu 3.5 učinjeno postavljanjem polja *niz* i *vel* na nulu posle oslobađanja memorije u dinamičkoj zoni operatom *delete*.

U javnom delu klase prvo su navedena tri konstruktora: podrazumevani konstruktor, konstruktor konverzije i konstruktor kopije. Podrazumevani konstruktor samo postavlja rednosti oba polja na nule. Konstruktor konverzije stvara skup od jednog elementa. Konstruktor kopije vrednost svog parametra kopira u tekući objekat pozivanjem već imenovane privatne metode `kopiraj()`. Pošto su sva tri konstruktora vrlo jednostavna, definisana su unutar definicije klase, što znači da će biti ugrađeni u kôd.

Sledeći, praktično obavezni element klase koji koriste dinamičko dodeljivanje memorije je destruktor. Destruktor samo poziva privatnu metodu `brisi()`, pa se i on ugrađuje u kôd.

Iza destruktora slede metode `presek()`, `razlika()` i `unija()` za ostvarivanje odgovarajućih skupovnih operacija nad skupovima koji se navode kao parametri (upućivači na nepromenljive podatke) i stavlju rezulat u tekući objekat. Pored njih postoje još metode `isi()` za ispisivanje sadržaja tekućeg objekta na glavnom izlazu računara (`cout`) i za citanje skupa (broja elemenata i samih elemenata) s glavnog ulaza računara (`cin`). Sve ove metode su relativno složene, pa su njihove definicije odložene za kasnije. Pošto metoda `isi()` ne menja vrednost svog tekućeg objekta, dodat je modifikator `const` na kraju jenog prototipa.

Poslednja metoda u klasi `Skup` je mala uslužna javna metoda `velicina()` za dohvatanje privatnog polja `vel`, za slučaj da nekoga interesuje trenutni broj elemenata tekućeg skupa. Ovakve metode, koje ne rade ništa već samo dohvataju neki privatno polje klase su vrlo česte u praksi. Njihovo ostvarenje u obliku pravih funkcija bilo bi vrlo neefikasno, pa h uvek treba predvideti da budu ugrađene metode.

Pored dohvatanja privatnih polja, u praksi se često sreću i javne metode koje samo postavljaju vrednost po jednog privatnog polja. Pri tome, po potrebi, one mogu i da provjeravaju da li navedena vrednost može da se prihvati kao novi sadržaj polja.

Ostvarenje opisanih metoda, a koje nisu navedene unutar definicije klase `Skup`, prikazano je programom 3.6.

O metodi `kopiraj()` nema mnogo da se kaže. Posle dodele potrebne količine memorije (`s.vel`) u dinamičkoj zoni, prenose se elementi niza iz pravog parametra (`s.niz`) u tekući objekat (na koji pokazuje skriveni parametar `this`).

Metoda `presek()` je već interesantnija. Prvo se definiše lokalni skup `s` koji se, u odsustvu inicijalizatora, inicijalizuje podrazumevanim konstruktorom kao prazan skup. Pošto presek dva skupa sadrži elemente koji se pojavljuju u oba skupa istovremeno, rezultat ne može da ima više elemenata od onoga koliko ima manji od ta dva skupa. Upravo se toliko memorija dodeljuje lokalnom skupu `s`. Pošto su po pretpostavci skupovi uređeni, njihov presek može da se obrazuje u jednom prolasku kroz elemente skupova. Ciklus traje dok se ne stigne do kraja jednog od skupova (nizova u kojima su elementi skupova). U svakom prolasku kroz ciklus upoređuju se tekući elementi oba skupa (`s1.niz[i]` prema `s2.niz[j]`). Ako je element prvog skupa manji od elementa drugog skupa on može da se odbaci jer se u nastavku drugog skupa nalaze samo još veći elementi. Slično, ako je element prvog skupa veći od elementa drugog skupa, element drugog skupa može da se odbaci. Ako su tekući elementi oba skupa međusobno jednak, potrebno je tu zajedničku vrednost smestiti u rezultujući skup (`s.niz[k++]`) i preći na sledeće elemente oba početna skupa istovremenim povećavanjem indeksa `i` i `j`. Po izlasku iz ciklusa, `s` predstavlja veličinu rezultujućeg skupa i to se stavlja u polje `s.vel` lokalnog skupa. Ta veličina će najčešće biti manja od prostora koji je dodeljen na početku metode. Preostaje još da se rezultat ozvaniči.

3.14.2 Obrada uređenih skupova

Definicije metoda klase `Skup`.

```
#include "skup.h"
#include <iostream>
using namespace std;

void Skup::kopiraj (const Skup& s) { // Kopiranje skupa.
    niz = new double [vel = s.vel];
    for (int i=0; i<vel; i++) niz[i] = s.niz[i];
}

void Skup::presek (const Skup& s1, const Skup& s2) { // Presek dva skupa.
    Skup s; s.niz = new double [s1.vel<s2.vel ? s1.vel : s2.vel];
    int i = 0, j = 0, k = 0;
    while (i<s1.vel && j<s2.vel)
        if (s1.niz[i] < s2.niz[j]) i++;
        else if (s1.niz[i] > s2.niz[j]) j++;
        else
            s.niz[k++] = s1.niz[j++, i++];
    s.vel = k; brisi (); kopiraj (s);
}

void Skup::razlika(const Skup& s1, const Skup& s2) { // Razlika dva skupa.
    Skup s; s.niz = new double [s1.vel];
    int i = 0, j = 0, k = 0;
    while (i < s1.vel)
        if (j >= s2.vel) s.niz[k++] = s1.niz[i++];
        else if (s1.niz[i] < s2.niz[j]) s.niz[k++] = s1.niz[i++];
        else if (s1.niz[i] > s2.niz[j]) j++;
        else
            { i++; j++; }
    s.vel = k; brisi (); kopiraj (s);
}

void Skup::unija (const Skup& s1, const Skup& s2) { // Unija dva skupa.
    Skup s; s.niz = new double [s1.vel+s2.vel];
    int i = 0, j = 0, k = 0;
    while (i<s1.vel || j<s2.vel)
        s.niz[k++] = (i<s1.vel && j<s2.vel)
            ? (s1.niz[i]<s2.niz[j] ? s1.niz[i++] :
               s1.niz[i]>s2.niz[j] ? s2.niz[j++] :
               s1.niz[j++, i++])
            : (i < s1.vel ? s1.niz[i++] : s2.niz[j++]);
    s.vel = k; brisi (); kopiraj (s);
}

void Skup::pisi () const { // Ispisivanje skupa.
    cout << '(';
    for (int i=0; i<vel; i++) { cout << niz[i]; if (i < vel-1) cout << ',';
    cout << ')';
}

void Skup::citaj () { // Čitanje skupa.
    brisi ();
    int vel; cin >> vel;
    double broj;
    for (int i=0; i<vel; i++) { cin >> broj; unija (*this, broj); }
```

Pre svega, potrebno je uništiti zatečeni sadržaj u tekućem objektu (na koji pokazuje skriveni parametar) koji je ispravan skup i kao takav koristi i prostor u dinamičkoj zoni memorije da veličinu skupa određuje na osnovu vrednosti polja `vel` svog argumenta (u ovom slučaju to je `s.vel`), kopiji će dodeliti tačno toliko memorije u dinamičkoj zoni koliko je mesta u nizu `s.niz` popunjeno, a ne koliko je prostora dodeljeno na osnovu procene očekivane veličine rezultujućeg skupa. Na kraju, prilikom povratka iz metode `presek()`, pošto se tada napušta doseg lokalnog prolaznog objekta `s`, destruktorni će da oslobodi ceo prostor u dinamičkoj zoni memorije dodeljen objektu `s`, a ne samo `s.vel` elemenata (što inače fizički nije moguce). Na taj način trajno zauzeće memorije biće upravo toliko koliko je potrebno za smeštanje elemenata rezultujućeg skupa, bez obzira što njihov broj unapred nije mogao da se odredi tačno, pa je privremeno zauzet nešto veći prostor. Još jedna završna primedba. Ovako napravljena metoda `presek()` omogućava da se rezultat operacije stavi u treći skup (`c.presek(a,b)`), ali može da se stavlja i u jedan od početnih skupova (`a.presek(a,b)` ili `b.presek(a,b)`), pošto se početni sadržaj odredišnog skupa (tekućeg objekta metode) uništava tek kada je već rezultat napravljen u lokalnom objektu `s`.

Osnovna ideja metoda `razlika()` i `unija()` je slična metodi `presek()` pa se njihova detaljna analiza prepušta čitaocu.

Metoda `pisi()` na uobičajeni način ispisuje elemente skupa, između para vitičastih zagrada međusobno razdvojenih zarezima (na primer: `{1,3,4,6}`). Poslednja metoda `citaj()` prvo uništi stari sadržaj tekućeg objekta i posle pročita broj elemenata u lokalnu promenljivu `vel` (koja sakriva polje `vel!`) i same elemente skupa. Elementi se jedan po jedan čitaju u lokalnu promenljivu `broj` odakle se metodom `unija()` stavljuju u tekući objekat. Argumenti metode su tekući objekat (`*this`) i rezultat konverzije sadržaja promenljive `broj` tipa `double` u skup od jednog elementa automatskim pozivanjem konstruktora konverzije (`Skup(double)`). Na ovaj način rezultat čitanja sigurno će biti uređeni skup, bez obzira da li korisnik unese korektan niz vrednosti. Naravno, ako među unetim vrednostima bude međusobno jednakih, broj elemenata skupa neće biti jednak najavljenom broju na početku metode `citaj()`. Ono što je najvažnije je da je obezbeđeno da ni na koji način ne može da se pojavi objekat tipa `Skup` koji nema sva obeležja tog skupa. Prikazano rešenje je dosta nefikasno imajući na umu šta se sve radi dok se nađe `unija` dva skupa, ali je ovako najlakše za programiranje. Uz malo više programerskog truda moglo bi da se ostvari uređivanje niza čitanih brojeva i preskakanje eventualnih duplikata u samoj metodi `citaj()`, bez pozivanja drugih metoda. To bi svakako dovelo do manjeg utroška procesorskog vremena uz nešto veći utrošak memorije zbog dodatnog koda.

Program 3.7 predstavlja primer za prikazivanje mogućnosti klase `Skup`.

Čitaju se parovi skupova i nalaze se redom njihovi preseci, unije i razlike pozivanjem odgovarajućih metoda. Dobijeni rezultati se ispisuju pozivanjem metode `pisi()`. Skreće se pažnja na to da se svi skupovi stvaraju kao prazni skupovi, pa se njihov sadržaj menja kao rezultat pozivanja metoda klase `Skup`. Sadržaj skupa s se menja čak tri puta. Takođe, važno je uočiti da se na kraju svakog prolaska kroz ciklus napušta doseg identifikatora definisanih skupova, pa se automatskim pozivanjem destruktora njihov sadržaj uništava, tj. oslobađa se memorija koja je u dinamičkoj zoni dodeljena za smeštanje elemenata skupa.

Rezultat 3.2 prikazuje primer rada programa 3.7 s dva kompleta ulaznih podataka.

3.14.2 Obrada uređenih skupova

```
// Program za ispitivanje klase Skup.

#include "skup.h"
#include <iostream>
using namespace std;

int main () {
    char jos;
    do {
        Skup s1; cout << "niz1? "; s1.citaj ();
        Skup s2; cout << "niz2? "; s2.citaj ();
        cout << "s1 ="; s1.pisi (); cout << endl;
        cout << "s2 ="; s2.pisi (); cout << endl;
        Skup s; s.presek (s1, s2); cout << "s1*s2="; s.pisi (); cout << endl;
        s.razlika (s1, s2); cout << "s1-s2="; s.pisi (); cout << endl;
        s.unija (s1, s2); cout << "s1+s2="; s.pisi (); cout << endl;
        cout << "\nJos? "; cin >> jos;
    } while (jos=='d' || jos=='D');
}
```

Program 3.7 – Prikazivanje mogućnosti klase `Skup` (skupt.C)

```
& skupt
niz1? 4 1 2 3 4
niz2? 5 3 4 5 6 7
s1 ={1,2,3,4}
s2 ={3,4,5,6,7}
s1*s2={3,4}
s1-s2={1,2}
s1+s2={1,2,3,4,5,6,7}

Jos? d
niz1? 6 9 3 5 1 7 3
niz2? 4 6 2 8 8
s1 ={1,3,5,7,9}
s2 ={2,6,8}
s1*s2={}
s1-s2={1,3,5,7,9}
s1+s2={1,2,3,5,6,7,8,9}

Jos? n
```

Rezultat 3.2 – Obrada skupova programom 3.7

U prvom kompletu oba uneta niza podataka su stvarno uređeni skupovi. Napominje se da u svakom nizu brojeva prvi predstavlja broj elemenata niza, a tek od drugog broja počinju vrednosti namenjene za umetanje u skup.

U drugom kompletu među podacima se nalaze i brojevi međusobno jednakih vrednosti, a brojevi nisu ni po rastućem redosledu svojih vrednosti. Bez obzira na to, rezultati njihovih čitanja su pravi uređeni skupovi.

3.14.3 Operacije nad jedinstvenim stekom

Zadatak:

Napisati na jeziku C++ klasu za realizaciju jedinstvenog steka za celobrojne podatke koristenjem dinamičkih struktura podataka. Napisati na jeziku C++ program za prikazivanje ispravnosti dobijene klase.

Rešenje:

Stek je niz podataka čijim elementima se prilazi isključivo na jednom kraju niza. Taj kraj se naziva *vrh steka*. Osnovne operacije nad stekovima su dodavanje i uzimanje podataka. Novi podatak dodaje se na vrh steka. Uzimanje podataka se vrši, takođe, s vrha steka.

Kaže se da stek ima ograničen kapacitet ako se u momentu stvaranja određuje najveći broj podataka koji mogu istovremeno da budu stavljeni na stek. Takvi stekovi mogu da se ostvaruju u obliku nizova elemenata odgovarajućih tipova.

Kaže se da stek ima neograničen kapacitet, ako se najveći broj istovremenih podataka na steku ne navodi u momentu stvaranja steka. Takvi stekovi se, obično, ostvaruju u obliku linearnih lista. Elementi liste treba, pored korisnih podataka, da sadrže i pokazivač na prethodni element liste po redosledu stavljanja na stek, tj. umetanja u listu. Razume se, i ovakvi stekovi imaju ograničene kapacitete, ali to ograničenje nije ugrađeno u program već predstavlja ukupan raspoloživi memoriski prostor u dinamičkoj zoni memorije.

Pod „jedinstvenim stekom“ u tekstu zadatka podrazumeva se stvaranje zajedničkog steka od svih primeraka klase za stekove. Naglašava se da ovo nije uobičajeno u praksi. Mnogo su češće klase za stekove kod kojih svaki primerak klase predstavlja kompletan stek. Na taj način, mogu istovremeno da postoje više nezavisnih stekova. To je pokazano u jednom od kasnijih zadataka. Ovaj zadatak treba shvatiti kao primer za korišćenje zajedničkih članova klase, a ne kao uzor za klase stekova.

Program 3.8 predstavlja jednu moguću definiciju klase za ostvarivanje steka neograničenog kapaciteta uz upotrebu zajedničkih članova klase.

```
// Definicija klase za stek celih brojeva (Stek).
class Stek {
    static Stek* vrh; // Zajednički pokazivač na vrh steka.
    Stek* preth; // Pokazivač na prethodni objekat na steku.
    int broj; // Koristan sadržaj objekta na steku.

public:
    static void stavi (int i); // Stavljanje novog podatka na stek.
    static int uzmi (); // Uzimanje podatka sa steka.
    static bool prazan () { return vrh == 0; } // Da li je stek prazan?
    static void prazni (); // Pražnjenje steka.
};
```

Program 3.8 – Definicija klase za stek (stek1.h)

Zajedničko polje *vrh* predstavlja pokazivač na vrh steka, tj. na primerak klase koji je poslednji stvoren i stavljen na stek. Pojedinačna polja su pokazivač na prethodni element steka *preth* i koristan sadržaj tekućeg elementa *broj*. Naglašava se, još jednom, da na ovaj način svi primerci klase *Stek* zajedno čine jedan stek.

3.14.3 Operacije nad jedinstvenim stekom

S obzirom da sve predviđene metode treba da deluju na stek kao celinu, a ne na pojedinačne podatke na steku (primerke klase), sve su deklarisane kao zajedničke metode. Program 3.9 prikazuje njihove definicije.

Definicije metoda klase Stek.

```
#include "stek1.h"
#include <cstdlib>
using namespace std;

Stek* Stek::vrn = 0; // Definicija vrh-a s inicijalizacijom.

void Stek::stavi (int i) { // Stavljanje novog podatka na vrh steka:
    Stek* novi = new Stek; // - dodala memorije,
    novi->broj = i; // - smeštanje podatka u element,
    novi->preth = vrh; // - uključivanje elementa u listu.
    vrh = novi;
}

int Stek::uzmi () { // Uzimanje podatka s vrha steka:
    if (! vrh) exit (2); // - prekid programa ako je stek prazan,
    int i = vrh->broj; // - pamćenje podatka s vrha steka,
    Stek* stari = vrh; // - isključivanje elementa iz liste,
    vrh = vrh->preth; // - oslobadanje dodeljene memorije,
    delete stari; // - vraćanje upamćenog podatka.
    return i;
}

void Stek::prazni () { while (vrh) uzmi (); } // Pražnjenje steka.
```

Program 3.9 – Definicije uklas Stek (stek1.C)

Pre svega, tu se nalazi definicija zajedničkog polja *vrh*. Inicijalizacijom vrednošću nula označava se da je na početku programa stek prazan. Zajednička polja se inicijalizuju pre nego što se bilo šta koristi iz klase, najčešće neposredno pred početak izvršavanja programa (pozivanjem funkcije *main()*).

Metoda *stavi()* prvo treba da stvara novi primerak klase *Stek*. Posle je potrebno u novi objekat smestiti koristan sadržaj i uključiti ga u listu.

Uzimanje podatka je moguće samo ako na steku postoji bar jedan podatak, tj. ako pokazivač *vrh* nije nula. Ako to nije slučaj, u metodi *uzmi()*, predviđen je prekid programa i povratak operativnom sistemu. Ako stek nije prazan prvo treba koristan sadržaj elementa na vrhu sačuvati u nekoj lokalnoj promenljivoj, isključiti element iz liste i tek onda je dozvoljeno osloboditi memoriju koja je ranije bila dodeljena u dinamičkoj zoni. Na kraju, upamćeni koristan sadržaj predstavlja vrednost funkcije.

Da bi korisnik mogao da se zaštiti od nasilnog prekida pri pokušaju uzimanja podatka s praznog steka, predviđena je metoda *prazan()* čija je vrednost logička istina (*true*) ako je stek prazan. Definicija metode nalazi se u samoj definiciji klase *Stek*. U slučaju stekova ograničenih kapaciteta obično se pravi i metoda *pun()* za ispitivanje da li na steku ima još slobodnih mesta.

Na kraju, komplet metoda za rad sa stekom treba da sadrži i metodu za pražnjenje steka. U posmatranom slučaju pražnjenje steka je u nadležnosti metode `prazni()`. Ona to postiže uzimanjem sukcesivnih vrednosti sa steka, sve dok pokazivač na vrh steka ne bude nula.

Skreće se pažnja na način kako zajedničke metode pristupaju pojedinačnim poljima. Pošto metode ne poseduju pokazivač `this`, za pristupanje pojedinačnim članovima (poljima i metodama) uvek moraju da navedu neki konkretni objekat čijem pojedinačnom članu žele da pristupaju. Taj konkretni objekat u metodi `stavi()` je objekat na koji pokazuje pokazivač `novi`, a u metodi `uzmi()` pokazivač vrh.

Program 3.10 služi za prikazivanje ispravnog rada svih metoda u klasi `Stek`.

```
// Program za ispitivanje klase Stek.

#include "stekl.h"
#include <iostream>
using namespace std;

int main () {
    for (;;) {
        int n;
        cout << "\nDuzina niza brojeva? "; cin >> n;
        if (! n) break;
        cout << "Niz od " << n << " brojeva? ";
        for (int i=0, k; i<n; i++) { cin >> k; Stek::stavi (k); }
        cout << "Niz po obrnutom redosledu:";
        while (! Stek::prazan ()) cout << ' ' << Stek::uzmi (); cout << endl;
        Stek::prazni ();
    }
}
```

Program 3.10 – Ispitivanje klase `Stek` (stek1t.C)

Glavna funkcija ima strukturu ciklusa s izlazom u sredini. Program čita nizove brojeva s glavnog ulaza računara stavljajući ih na stek i posle toga, uzimajući ih sa steka, ispisuje na glavnom izlazu računara. Na taj način brojevi u ispisanim nizu treba da su po obrnutom redosledu u odnosu na redosled čitanja. Program se završava kada za dužinu niza brojeva pročita nulu.

Na kraju svakog prolaska kroz glavni ciklus programa poziva se funkcija za pražnjenje steka. U konkretnom primeru to je suvišno, jer tog momenta stek je sigurno prazan. To je posledica ispisivanja uzastopnih brojeva sa steka sve dok stek ne bude ispraznen!

Interesantno je uočiti da u programu 3.10 ne postoji nijedna definicija objekata tipa `Stek`. Za korisnika stek jednostavno postoji od samog početka programa i u vezi s tim korisnik ne treba ništa da preduzima. Metode, naravno, pozivaju se označavanjem klase kojoj pripadaju (`Stek::`).

Rezultat 3.3 prikazuje primer rada programa 3.10.

Duzina niza brojeva? 5
Niz od 5 brojeva? 1 2 3 4 5
Niz po obrnutom redosledu: 5 4 3 2 1

Duzina niza brojeva? 9
Niz od 9 brojeva? 9 8 7 6 5 4 3 2 1
Niz po obrnutom redosledu: 1 2 3 4 5 6 7 8 9

Duzina niza brojeva? 0

Rezultat 3.3 – Obrada stekova programom 3.10

3.14.4 Obrada krugova u ravni

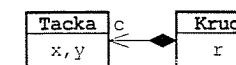
Zadatak:

Napisati na jeziku C++ klasu za obradu krugova u ravni koji ne smeju da se preklapaju. Napisati na jeziku C++ program za prikazivanje mogućnosti te klase.

Rešenje:

Program 3.11 prikazuje definiciju klase `Krug` koja zadovoljava uslove zadatka.

Privatna polja krugova su `centar` c tipa `Tacka` (klasa `Tacka` je ostvarena u zadatu u odeljku 3.14.1) i poluprečnik `r` tipa `double`. Odnos između klase `Krug` i `Tacka` prikazan je dijagramom klasa na slici 3.4. Jednostavnosti radi, navedeni su samo polja u klasama.



Slika 3.4 – Odnos između klase Krug i Tacka

Da bi moglo da se spreči međusobno delimično ili potpuno preklapanje bilo koja dva kruga među primercima klase `Krug`, unutar klase mora da se znaju svi postojeći primerci te klase. To može da se ostvari pomoću zajedničke liste svih krugova. Unutrašnja struktura `Elem` opisuje elemente te liste. Pošto se krugovi stvaraju u raznim delovima programa, neki u dinamičkoj zoni memorije dok drugi ne, lista može da sadrži samo pokazivače na te objekte, a ne i same objekte. Zbog toga kao polja strukture `Elem` pojavljuju se pokazivač na krug (krug tipa `Krug*`) i pokazivač na sledeći element liste (sled tipa `Elem*`). Pored njih postoji još samo konstruktor za stvaranje novog elementa liste. Kao što je rečeno u odeljku 3.12, strukture bi trebalo koristiti samo za „neinteligentne“ podatke bez složenih metoda. Ipak, postojanje konstruktora i destruktora često je vrlo korisno u strukturama. Dalje metode bi trebalo izbegavati. Ako se oseća potreba za njima, treba razmisiliti o projektovanju klase umesto strukture.

Lista svih krugova unutar klase predstavljena je zajedničkim (`static`) poljem `prvi` tipa `Elem*`.

Od metoda na prvom mestu su konstruktori i destruktur, metode koje imaju poseban status u klasama.

Inicijalizacija kruga podrazumevanim konstruktorom u posmatranom slučaju nema smisla, jer bi se na taj način dobijali identični krugovi koji bi se međusobno u potpunosti preklapali. Zbog toga je on pravljen kao privatna metoda. Prevodilac će da prijavljuje grešku

// Definicija klase krugova u ravni (Krug).

```
#include "tacka1.h"
#include <cstdlib>
#include <cmath>
#include <iostream>
using namespace std;

class Krug {
    Tacka c; double r; // Centar i poluprečnik kruga.
    struct Elem {
        Krug* krug; // Element zajedničke liste svih krugova.
        Elem* sled; // - pokazivač na krug,
        Elem (Krug* k, Elem* s=0); // - pokazivač na sledeći element liste,
        - stvaranje elementa liste.
    };
    static Elem* prvi; // Početak zajedničke liste.
    Krug () { r = -1; } // Stvaranje "praznog" kruga:
    // - SAMO za interne potrebe!
    Krug (const Krug&); // Konstruktor kopije:
    // - NE SME da se pravi kopija!

public:
    Krug (double rr, double x, double y); // Stvaranje kruga;
    ~Krug (); // Uništavanje kruga.
    static bool moze (double r, double x, double y); // Može li stati?
    friend double rastojanje (const Krug& k1, const Krug& k2);
    bool postavi (double x, double y); // Postavljanje kruga.
    bool pomeri (double dx, double dy); // Pomeranje kruga.
    void pisi () const; // Pisanje kruga.
    static void pisi_sve (); // Pisanje svih krugova.

    // Definicije ugrađenih funkcija.

    inline double rastojanje (const Krug& k1, const Krug& k2)
        // Rastojanje dva kruga.
    { return k1.c.rastojanje (k2.c) - k1.r - k2.r; }

    inline void Krug::pisi () const // Pisanje kruga.
    { cout << "K[" << r << ','; c.pisi(); cout << ']'; }

    inline Krug::Elem::Elem (Krug* k, Elem* s) // Stvaranje elementa liste.
    { krug = k; sled = s; }
}
```

Program 3.11 – Definicija klase krugova u ravni (krug1.h)

ako se izvan klase pokuša stvaranje objekta tipa Krug bez inicijalizatora. Pošto su unutar nekih metoda posmatrane klase potrebni pomoćni objekti tipa Krug, koji možda ne zadovoljavaju sve osobine klase (posebno, neprekapanje s ostalim krugovima), u telu podrazumevanog konstruktora poluprečnik se postavlja na nemoguću vrednost -1 . To ukazuje na činjenicu da objekat nije „pravi“ krug.

Zbog zahteva neprekapanja krugova, stvaranje kopija postojećih krugova nije dozvoljeno. Da bi se to sprečilo, u klasi Krug je definisan privatni konstruktor kopije praznog tela. Pošto je privatан, izvan klase sigurno neće moći da se pozove. Unutar klase vodilo se strogo računa o tome da se on ne pozove.

3.14.4 Obrada krugova u ravni

Jedini javan konstruktor ima tri parametra: poluprečnik i koordinate centra kruga koji treba da se stvori. Taj konstruktor ima da vodi računa o tome da se novi krug ne preklopi sa već postojećim krugovima.

Mada primerci klase Krug ne poseduju pokazivačka (pojedinačna) polja, destruktur je potreban radi izbacivanja kruga koji se uništava iz liste svih krugova. Time se omogućava eventualno postavljanje drugih krugova u tom delu ravni.

U nastavku definicije klase sledi prototipovi „običnih“ metoda i funkcija.

Zajednička metoda moze() proverava da li neki krug s datim poluprečnikom i datim koordinatama centra može da postoji. Pravljena je kao zajednička metoda jer ne može da joj se navede nijedan konkretan krug kao tekući objekat, jer ako takav krug već postoji, nema šta da se proverava da li može da postoji. Ova metoda, dakle, ispituje klasu Krug u celini s parametrima potencijalnog, ali još nepostojećeg, kruga.

Prijateljska funkcija rastojanje() izračunava najmanje rastojanje između zadata dva kruga, tj. rastojanje između najbliže dve tačke na periferiji tih krugova. Pravljena je kao prijateljska funkcija radi naglašavanja ravноправности njenih parametara u odnosu na datu radnju (rastojanje nije orijentisan pojam). Pozivanje sa rastojanje(k1, k2) ne pravi razliku u statusu krugova, oba kruga se u funkciju prenose na isti način. U slučaju metode pozivanje sa k1.rastojanje(k2) jasno ukazuje na različiti tretman tih dva kruga: prvi je tekući (skriveni) objekat, a drugi pravi argument metode. Skreće se još pažnja na to da su parametri funkcije rastojanje() upućivači (Krug&). U posmatranom slučaju drugačije i ne može biti. Da su parametri tipa Krug (a ne upućivači), to bi podrazumevalo inicijalizaciju svakog parametra kopijom odgovarajućeg argumenta, što bi dovelo do pojave po dva istovetna kruga. Privatan konstruktor kopije sprečava da korisnik klase tako nešto pokuša u slučaju svojih funkcija.

Metode postavi() i pomeri() postavljaju centar postojećeg kruga (tekućeg objekta) u određenu tačku u ravni, odnosno pomeranje centra postojećeg kruga za zadati pomeraj. Vrednosti funkcija predstavljaju indikator uspeha tih radnji.

Poslednja, zajednička metoda pisi_sve() ispisuje podatke o svim trenutno postojećim krugovima. Predviđena je za potrebe ispitivanja klase Krug. Inače nema neku veću upotrebnost vrednost.

Iza definicije klase Krug u programu 3.11 nalaze se definicije jednostavnih metoda i funkcija koje su predviđene da budu ugrađene funkcije. Njihove definicije moraju biti dostupne prevodiocu svuda gde se pozivaju, pa se obično stavljaju u istu datoteku s definicijom klase i u slučajevima kada se definišu izvan klase.

Navođenje definicije malih metoda i funkcija u definiciji klase čini tekst klase manje preglednim. Navođenje definicija izvan klase zahteva navođenje eksplicitnog zahteva za ugradivanjem (**inline**) i ponavljanje zaglavlja metoda i funkcija. Na ovaj način ukupan tekst je obimniji.

U prijateljskoj funkciji rastojanje() rastojanje između centara krugova se računa istoimenom metodom iz klase Tacka. Pošto prijateljska funkcija nije član klase, definiše se kao i bilo koja obična funkcija, bez posebne naznake da je prijatelj nekoj klasi.

Metoda pisi() će podatke o krugu ispisati u obliku $K[r, (x, y)]$, s tim da za ispisivanje koordinata centra poziva istoimenu metodu iz klase Tacka. Pošto je c tipa Tacka, poziv c.pisi() nedvosmisleno se odnosi na metodu iz klase Tacka, a ne iz klase Krug. Ne smeta što su obe metode s jednakim parametrima (u ovom slučaju, bez parametara), jer se ovde ne radi o preklapanju imena funkcija, već o metodama iz različitih klasa. Tehnički

gleđano, moglo bi da se govori i o preklapanju imena funkcija ali koje pored pravih parametara (kojih u ovom slučaju nema, pa su isti) imaju i po jedan skriveni parametar čiji su tipovi različiti (jer su adrese primeraka različitih klasa).

Na kraju treba da se obrati pažnja na definiciju konstruktora strukture `Elem`. Pošto ispred imena bilo koje metode koja se definiše izvan svoje klase treba navesti ime te klase, zaključuje se da bi trebalo da se piše `Elem::Elem()`. Međutim, struktura `Elem` je uklopljena u klasu `Krug`, pa je doseg njenog identifikatora ograničen na tu klasu. Zbog toga izvan klase `Krug` neophodno je naznačiti i da se misli na identifikator strukture `Elem` koja se nalazi u klasi `Krug`, što dovodi do notacije `Krug::Elem::Elem()`. Prva dva identifikatora su identifikatori klase/strukture, a treći je identifikator metode (u ovom slučaju konstruktora koji se poklapa s identifikatorom klase/strukture).

Program 3.12 predstavlja sve definicije potrebne uz klasu `Krug`. To podrazumeva definicije zajedničkih polja i svih pravih metoda i prijateljskih funkcija (ako ih ima).

Pokazivač na početak zajedničke liste je tipa `Elem*`, ali pošto je struktura `Elem` uklopljena u klasu `Krug`, mora da se piše `Krug::Elem*`. Slično tome, i zajednički član `prvi` nalazi se u klasi `Krug`, izvan klase mora da se piše `Krug::prvi`. Pokazivač se inicijalizuje nulom, što obezbeđuje da je na početku izvršavanja programa lista prazna.

Konstruktor `Krug::Krug()` prvo proverava, pozivanjem zajedničke metode `moze()`, da li sme da konstruiše traženi krug. Ako ne, prekida program pozivanjem globalne funkcije `exit()`, jer povratkom u pozivajući program u programu bi postojao neispravan krug. Dužnost je korisnika da pre pokušaja stvaranja novog kruga proveri (takođe pozivanjem javne zajedničke metode `moze()`), da li sme to da uradi. Propusti li korisnik tu proveru, rizikuje nasilni prekid programa. Ako je stvaranje traženog kruga dozvoljeno, posmatrani konstruktor popunjava polja novog objekta i uključi ga u zajedničku listu svih krugova. Pošto redosled krugova u listi nije bitan novi krug se stavlja na početak liste, jer je to najjednostavnije. Skreće se pažnja na jednostavnost tog zahvata, zahvaljujući postojanju konstruktora u strukturi `Elem`. Pokazivač `this` predstavlja pokazivač na krug koji se stvara i koga treba staviti u listu.

Zadatak destruktora je da krug koji se uništava izbaci iz liste svih krugova. To sme da uradi samo za „ispravne“ krugove, koji su prilikom stvaranja uključeni u listu. Oni se prepoznaju po tome da im je poluprečnik $r > 0$. Krugove, privremenog karaktera, sa $r < 0$ može da stvara samo privatni podrazumevani konstruktor i oni se ne nalaze u listi. Element za izbacivanje iz liste se traži na uobičajeni način, krećući se duž liste sa pokazivačima na tekući (`tek`) i prethodni (`preth`) element liste. Traženi element u listi se prepoznaže po tome što njegovo polje `tek->krug` pokazuje na tekući objekat destruktora, na koga pokazuje i pokazivač (skriveni argument) `this`. Ciklus u posmatranom rešenju pretpostavlja da će se traženi element sigurno pronaći u listi, što će se stvarno i desiti pod uslovom da je klasa `Krug` ispravno programirana (oprezniji programer će, ipak, predviđeti proveru neuspešnog pretraživanja i prijaviti neočekivanu grešku i prekinuti program ako se desi da se traženi krug ne pronađe u listi). Po izlasku iz ciklusa pronađeni element liste treba uključiti iz liste i posle operatorom `delete` oslobođiti prostor koji je zauzimao u dinamičkoj zoni memorije. Skreće se pažnja na korišćenje uslovnog izraza (`?:`) kao *lrednosti* (na levoj strani operatora `=`). Na element iza onoga koji se briše (`tek->sled`) treba da pokazuje pokazivač `prvi` ako se briše prvi element u listi (tada je `preth==0`), odnosno `preth->sled` inače.

```
// Definicije metoda i zajedničkih polja klase Krug.

#include "krugl.h"

Krug::Elem* Krug::prvi = 0; // Početak zajedničke liste.

Krug::Krug (double rr, double x, double y) { // Stvaranje kruga.
    if (! moze (rr, x, y)) exit (1);
    r = rr; c.postavi (x, y);
    prvi = new Elem (this, prvi);
}

Krug::~Krug () { // Uništavanje kruga (izbacivanje iz liste).
    if (r > 0) {
        Elem *tek = prvi, *preth = 0;
        while (tek->krug != this) { preth = tek; tek = tek->sled; }
        (preth==0 ? prvi : preth->sled) = tek->sled;
        delete tek;
    }
}

bool Krug::moze (double r, double x, double y) { // Može li stati?
    Krug k; k.r = r; k.c. postavi (x, y);
    for (Elem* tek = prvi; tek; tek = tek -> sled) {
        if (rastojanje (k, *tek->krug) < 0) { k.r = -1; return false; }
    }
    k.r = -1; return true;
}

bool Krug::postavi (double x, double y) { // Postavljanje kruga.
    if (! moze (r, x, y)) return false;
    c. postavi (x, y);
    return true;
}

bool Krug::pomeri (double dx, double dy) { // Pomeranje kruga.
    if (! moze (r, c.aps ()+dx, c.ord ()+dy)) return false;
    c. postavi (c.aps ()+dx, c.ord ()+dy);
    return true;
}

void Krug::pisi_sve () {
    cout << "\nSvi krugovi u memoriji:\n\n";
    for (Elem* tek=prvi; tek; tek=tek->sled)
        { tek->krug->pisi(); cout << endl; }
}
```

Program 3.12 – Definicije metoda i zajedničkih polja uz klasu `Krug` (`krugl.C`)

U zajedničkoj metodi `moze()` prvo se stvara lokalni krug `k` privatnim podrazumevanim konstruktorom. Na taj način krug `k` neće da se uključi u listu svih krugova. Posle toga se popuni polje `k.r` dodelom vrednosti, a polje `k.c` pozivanjem metode `postavi()` iz klase `Tacka` (u klasi `Tacka` ne postoji konstruktor jer je ona projektovana u poglavljiju pre ovog poglavlja o konstruktorima i destruktorma). Ovako veštački napravljen krug `k` (koji možda i ne bi smeo da postoji zbog uslova neprekapanja krugova) omogućava korišćenje

prijateljske funkcije `rastojanje()` za ispitivanje da li se isti preklapa s nekim od već postojećih krugova. Krugovi se preklapaju ako je njihovo rastojanje negativno. Ako se u listi nađe bar jedan krug s negativnim rastojanjem u odnosu na krug `k`, vrednost funkcije biće `false` (logička neistina), a u suprotnom `true` (logička istina). Bez obzira na rezultat, pre naredbe `return` poluprečnik `k.r` obavezno se postavlja na `-1`. To će sprečiti da destruktur, koji će se pozivati za krug `k` prilikom povratka iz funkcije `moze()`, pokuša da pronađe krug `k` u listi svih krugova. On se tamo sigurno ne nalazi!

Metode `postavi()` i `pomeri()` za menjanje položaja kruga u ravni su vrlo jednostavne. Prvo proveravaju da li krug nepromjenjene vrednosti poluprečnika smještaju na novom mestu. Ako ne, vraćaju vrednost logičke neistine (`false`) i ostavljaju krug na starom mestu. Inače, premeštaju krug na novo mesto i vraćaju vrednost logičke istine (`true`). Kod ovih operacija neuspeh nije fatalan (za razliku od neuspeha pri stvaranju kruga). Ispravan krug postoji i pre i posle operacije, jedino je pitanje da li je došlo do promene položaja. Korisnik to može utvrditi na osnovu vrednosti funkcije. Skreće se još pažnja, u metodi `pomeri()`, na način dohvatanja koordinata centra kruga `c` koje su privatna polja klase `Tacka`. To se postiže pomoću javnih metoda `aps()` i `ord()` te klase. Pošto su one ugradene metode, program nije ništa manje efikasan od neposrednog pristupanja poljima. Jedino je tekst programa na ovakav način nešto duži, ali vredi uložiti taj napor radi bezbednosti korišćenja klase!

Na kraju, zajednička metoda `pisi_sve()`, prolazeći kroz listu svih krugova, ispisuje njihov sadržaj pomoću metode `pisi()`.

Program 3.13 je primer korišćenja klase `Krug` u kome se pročita niz krugova i posle se isti ispisuju.

```
// Program za ispitivanje klase Krug.

#include "krugl.h"
#include <iostream>
using namespace std;

int main () {
    Krug* krugovi [100]; int n = 0;
    while (true) {
        cout << "Poluprecnik? "; double r; cin >> r;
        if (r <= 0) break;
        cout << "Koordinate centra? "; double x, y; cin >> x >> y;
        if (Krug::moze (r, x, y)) {
            krugovi [n++] = new Krug (r, x, y);
        } else cout << "*** Ne može da se smesti! ***\n";
    }
    Krug::pisi_sve ();
}
```

Program 3.13 – Ispitivanje klase Krug (krugl.C)

Za smeštanje niza krugova predviđen je niz pokazivača na krugove (krugovi). Nije mogao da se definije niz krugova, jer se elementi nizova inicijalizuju podrazumevanim konstruktorom, a u klasi `Krug` taj konstruktor je privatан.

U ciklusu se prvo pročita poluprečnik sledećeg kruga i ako je negativan ciklus se završi. Posle poluprečnika pročitaju se koordinate centra i proverava se zajedničkom metodom `moze()` da li sme da se stvori takav krug. Time se štiti od nasilnog prekidanja programa u slučaju kada bi se novi krug preklopio s nekim od već postojećih krugova. U slučaju potvrđnog odgovora, novi krug se stvori u dinamičkoj zoni memorije, a pokazivač na njega se stavlja na sledeće mesto u nizu krugova. Skreće se pažnja na način pozivanja zajedničke metode sa `Krug::moze()`.

Po izlasku iz ciklusa svi uspešno stvoreni krugovi se ispisuju pozivanjem zajedničke metode `Krug::pisi_sve()`.

U nedostatku eksplicitnog uništavanja objekata u dinamičkoj zoni memorije, oni se uništavaju prilikom završetka programa. To će se desiti i u ovom primeru, ali bez pozivanja destruktora za pojedine objekte.

Rezultat 3.4 prikazuje primer rada programa 3.13. Zbog umetanja novih krugova na početak liste svih krugova, oni se metodom `pisi_sve()` ispisuju po suprotnom redsledu u odnosu na njihovo stvaranje.

```
krugl
Poluprecnik? 1
Koordinate centra? 1 1
Poluprecnik? 2
Koordinate centra? 3 3
*** Ne može da se smesti!
Poluprecnik? 2
Koordinate centra? 4 4

Poluprecnik? 3
Koordinate centra? -1 5
Poluprecnik? -1

Svi krugovi u memoriji:

K[3, (-1,5)]
K[2, (4,4)]
K[1, (1,1)]
```

Rezultat 3.4 – Obrada krugova programom 3.13

4 Preklapanje operatora

Za većinu standardnih operatora jezika C++ mogu da budu definisana tumačenja za slučajevе kada su operandi klasnih tipova. To omogućava pisanje izraza na uobičajeni način i za tipove podataka koje je definisao programer. Na primer, da može da se napiše $A = (B+C) * D$, gde su operandi matrice odgovarajućih dimenzija.

Tumačenja za klasne tipove podataka mogu da budu definisana za sve operatore izuzev operatora za pristup članu klase (. i .*), za razrešenje dosega (: :), za uslovni izraz (? :), za veličinu objekata (**sizeof**) i za prijavljivanje izuzetka (**throw**).

Definisanje novih tumačenja operatora podleže sledećim ograničenjima:

- ne mogu da budu redefinisana tumačenja operatora za standardne (numeričke i pokazivačke) tipove podataka,
- ne mogu da budu uvedeni novi simboli za operatore, i
- ne mogu da budu promjenjeni prioriteti ni smerovi grupisanja pojedinih operatora.

Zajednički cilj svih ovih ograničenja je onemogućavanje promene značenja programa koji koristi samo standardne tipove podataka. Drugim rečima, jezik C++ može da se prošire, a ne i da se menja od strane programera.

Prvi od prethodna tri uslova je ispunjen ako je bar jedan od operanada klasnog tipa.

Za sve novouvedene tipove podataka automatski postoji tumačenje samo za operatore dodelje vrednosti jednog objekta drugom (=), za obrazovanje adrese objekta (&) i za obrazovanje niza izraza (,).

Dodela vrednosti jednog objekta drugom podrazumeva kopiranje sadržaja jednog objekta u drugi objekat istog tipa, polje po polje. Ako su neka od polja pokazivači, pokazivani podobjekti se ne kopiraju. Dakle, ne dobija se kopija celog izvorišnog objekta!

4.1 Operatorske funkcije

Smatra se da se operatori u jeziku C++ ostvaruju operatorskim funkcijama čija imena imaju sledeći opšti oblik:

operator op

gde je **op** simbol odgovarajućeg operatora (na primer: **operator+**, **operator--**). Simbol operatora može da bude napisan spojeno ili odvojeno od reči **operator**, osim ako se

Simbol operatora sastoji od slova. Tada simbol mora da bude napisan odvojeno (na primer: **operator new**).

Imena operatorskih funkcija mogu da se preklapaju kao i kod drugih funkcija (videti deljak 2.4.3). To znači da je za dati operator moguće definisati proizvoljan broj tumačenja. Jedino je neophodno da se tipovi operanada dovoljno razlikuju, da bi prevodilac može jednoznačno da odabere odgovarajuće tumačenje.

Operatorske funkcije za klasne tipove mogu da budu metode klase ili „obične“ globalne funkcije (najčešće prijateljske funkcije, mada to nije neophodno).

Unarni operatori imaju jedan operand pa odgovarajuća operatorska funkcija treba da ma tačno jedan parametar. Zbog toga mogu da se ostvaruju metodama bez parametara ili globalnim funkcijama s jednim parametrom. Tip tog jedinog parametra mora da bude klasa a koju se data funkcija pravi. Metoda, naravno, poseduje skriveni parametar koji predstavlja adresu operanda datog unarnog operatora.

Binarni operatori imaju dva operanda pa odgovarajuća operatorska funkcija mora da ma tačno dva parametra. Zbog toga mogu da se ostvaruju metodama s jednim parametrom ili globalnim funkcijama s dva parametra. Prvi operand u slučaju metode je tekući objekat, i drugi operand je jedini pravi parametar. Drugi operand može da bude proizvoljnog tipa, uključujući i standardne. U slučaju globalne funkcije, bar jedan od parametara treba da je ipa klase za koju se data funkcija pravi. Preostali parametar može da bude proizvoljnog tipa, uključujući i standardne.

Operatorske funkcije ne mogu da imaju parametre s podrazumevanim vrednostima i ne mogu da budu zajedničke (**static**) metode. Operatori (*tip*), **=**, **()**, **[]** i **->** mogu da budu preklopljeni samo pojedinačnim metodama, ne i globalnim funkcijama. Od prethodnih pravila izuzeci su operatori **new** i **delete** koji mogu da budu preklopljeni samo zajedničkim metodama.

Evo primera klase s preklopljenim operatorima:

```
class T {
    ...
public:
    T1 operator- (); // Unarni - kao metoda.
    friend T2 operator! (T a); // Unarni ! kao prijateljska fun.
    T3 operator+ (T4 b); // Binarni + kao metoda.
    friend T5 operator* (T a, T6 b); // Binarni * kao prijateljska fun.
    friend T7 operator/ (T8 a, T b); // Binarni / kao prijateljska fun.
    ...
    T t; T4 x; T6 y; T8 z;
    ...
    T1 a = - t;
    T2 b = ! t;
    T3 c = t + x;
    T5 d = t * y;
    T7 e = z / t;
    ...
    // a = t.operator- ();
    // b = operator! (t);
    // c = t.operator+ (x);
    // d = operator* (t, y);
    // e = operator/ (z, t);
}
```

Vrednosti operatorskih funkcija mogu da budu potpuno proizvoljnih tipova. Operatorske funkcije mogu i da ne daju nikakve vrednosti funkcije, tj. da su tipa **void**. U posmatranom primeru to su tipovi T1, T2, T3, T5 i T7. Nema, naravno, prepreke da neki od tih tipova budu baš tip T, tj. klasa koja se ovde posmatra. U slučaju prijateljske (globalne)

funkcije parametar za unarne operatore i bar jedan parametar za binarne operatore mora da bude tipa T.

Operatorske funkcije mogu da se pozivaju i na način kako se pozivaju i sve ostale funkcije (videti komentare u prethodnom primeru). To se, međutim, radi samo vrlo retko. Obično se pozivaju implicitno, kao rezultat tumačenja operatora u izrazima.

Funkcije kojima se preklapaju operatori s bočnim efektima (to su operatori **++**, **--** i operatori dodele vrednosti) ne moraju da proizvode bočne efekte. Iz tog razloga operandi koji za standardne tipove podataka moraju da budu *lvrednosti*, ne moraju i za preklopljene operatore. Podsećanja radi, *lvrednost* je izraz koji obeležava neki podatak (objekat ili običan podatak) u memoriji. Ako operator ne menja vrednost svog operanda taj operand ne mora da bude *lvrednost*. Na primer, izrazom **a+=b** ne mora da se promeni vrednost operanda a.

S druge strane, operatorske funkcije mogu da proizvode bočne efekte čak i ako odgovarajući operatori za standardne tipove ne proizvode bočne efekte. Parametar za operand koji podleže bočnom efektu, naravno, mora da bude *lvrednost*, tj. upućivač. Na primer, izrazom **a+=b** mogu da se promene vrednosti čak oba operanda i da rezultat bude neka treća vrednost. Ipak, treba izbegavati ponašanje operatorskih funkcija neuobičajeno u odnosu na standardne tipove podataka.

Ukoliko je potrebno da se rezultat neke operatorske funkcije koristi kao prvi operand operatora za dodelu vrednosti (**=**), ili na drugim mestima gde se očekuje *lvrednost*, tip vrednosti funkcije treba da je upućivač na objekte odgovarajućeg tipa. Na primer, da bi moglo da se napiše **a[i]=b**, rezultat operatorske funkcije **operator[]()**, kojom se preklapa operator za indeksiranje **[]**, mora da je *lvrednost*. Podsećanja radi, za standardne tipove podataka, poređ indeksiranja, svi operatori za dodelu vrednosti i prefiksni oblici operatora **++** i **--** daju *lvrednost* kao rezultat. Vrlo često je korisno da se ta semantika očuva i u slučajevima preklapanja tih operatora. Naravno, ne postoji formalna obaveza za to, niti prepreka da operatorske funkcije za ostale operatore imaju *lvrednosti* za rezultate.

Formalno gledano, u većini slučajeva, svejedno je da li se operatori preklapaju metodama ili globalnim funkcijama. Kao jedan praktičan savet predlaže se ostvarivanje binarnih operatora bez bočnih efekata u obliku globalnih funkcija. Nema razloga da se operandi operatora bez bočnih efekata tretiraju međusobno različito. Tekući objekat i pravi parametar metode ne tretiraju se na isti način. Na primer, automatska konverzija tipa argumenta u tip parametra prilikom pozivanja metoda primenjuje se samo za prave parametre, a ne i za tekući objekat metode. S druge strane, deluje prirodno da operand koji podleže bočnom efektu bude tekući objekat operatorske funkcije koja je član svoje klase. Pošto se kod unarnih operatora nikada ne primenjuje automatska konverzija tipa operanda preporučuje se njihovo ostvarivanje u obliku metoda. Na taj način se više ističe pripadnost matičnoj klasi.

Funkcionalne veze koje važe među nekim operatorima za standardne tipove podataka (na primer **+**, **++** i **+=**) ne uspostavljaju se automatski preklapanjem jednog od tih operatora. Neophodno je da se svaki od tih operatora, nezavisno, preklopni na odgovarajući način. Preporučljivo je da se to i uradi za tipove podataka za koje to ima smisla, na primer za klasu kompleksnih brojeva.

Na kraju, jedno važno upozorenje. Po svaku cenu treba izbegavati neočekivana tumačenja preklopljenih operatora. Na primer, sabiranje matrica ne treba ostvarivati preklapanjem operatora minus (**-**). Na žalost, nema formalnog načina kojim bi se to sprečilo.

4.2 Preklapanje operatora ++ i --

Unarni operatori ++ i -- su specifični po tome što imaju prefiksni (++a) i postfiksni (a++) oblik.

Uobičajene operatorske funkcije za tip (klasu) T u obliku metode bez parametara (na primer, T1 **operator++()**), odnosno globalne funkcije s jednim parametrom (na primer, T1 **operator++(T)**) koriste se za preklapanje prefiksnih oblika tih operatora.

Postfiksni oblici se preklapaju metodama s jednim parametrom tipa **int** (na primer, T1 **operator++(int)**), odnosno globalnim funkcijama s dva parametra od kojih je prvi tipa klase za koju se vrši preklapanje a drugi je tipa **int** (na primer, T1 **operator++(T, int)**). Vrednost tog celobrojnog parametra je nula ako se te funkcije pozivaju kao posledica primene operatora ++ odnosno -- (na primer, t++). Ako se, međutim, koristi notacija za pozivanje funkcija (na primer, **t.operator++(k)**) odnosno **operator++(t, k)**, vrednost celobrojnog argumenta k prenosi se u funkciju. Svrha celobrojnog parametra nije da se on koristi, već da za dva oblika istog operatora postoje dve funkcije s različitim prototipovima.

Evo primera preklapanja operatora ++ i -- za klasu kompleksnih brojeva:

```
class Kompl {
    float re, im;
public:
    Kompl (float r, float i)           // Stvaranje kompleksnog broja.
        { re = r; im = i; }
    Kompl& operator++ ()              // Prefiksni oblik.
        { ++re; return *this; }
    Kompl operator++ (int k)          // Postfiksni oblik.
        { Kompl z = *this; re+=k; im+=k; return z; }
    friend Kompl& operator-- (Kompl& z) // Prefiksni oblik.
        { --z.re; return z; }
    friend Kompl operator-- (Kompl& z, int k) // Postfiksni oblik.
        { Kompl x = z; z.re-=k; z.im-=k; return x; }
    ;
    Kompl x (1,3);                  // x = (1,3)
    Kompl a = ++ x;                // x = (2,3),   a = (2,3)
    Kompl b = x ++;                // b = (2,3),   x = (3,3)
    Kompl c = x.operator++ (2);    // c = (3,3),   x = (4,5)
    Kompl d = -- x;                // x = (3,5),   d = (3,5)
    Kompl e = x --;                // e = (3,5),   x = (2,5)
    Kompl f = operator-- (x, 2);    // f = (2,5),   x = (1,3)
```

Kompleksan broj sastoji se od realnog (re) i imaginarnog dela (im). Konstruktor **Kompl()** stvara kompleksne brojeve od dva realna broja.

Preklapanje operatora ++ je dato u obliku metoda, a operatora -- u obliku prijateljskih funkcija.

Za prefiksne oblike oba operatora odmah može da se izvrši promena realnog dela parametra (iskrivenog ili pravog) za jedan (bočni efekat) i rezultat je promenjeni parametar. Pošto je za standardne tipove podataka rezultat ovih operatora *vrednost*, tipovi funkcija su definisani kao upućivači (**Kompl&**).

Kod postfiksnih oblika, prvo treba, u lokalnoj promenljivoj z, sačuvati početnu vrednost parametra, zatim promeniti vrednost parametra (bočni efekat) i sačuvanu vrednost s početka vratiti kao rezultat funkcije. Mogućnost postojanja dodatnog celobrojnog

parametra iskorišćena je za menjanje vrednosti imaginarnog dela kompleksnog broja. Taj parametar dolazi do izražaja samo kada se operatorske funkcije pozivaju kao obične funkcije. Ovakva kombinacija se u praksi se praktično nikad ne koristi. Ovde je navedena samo kao školski primer.

4.3 Operatori za ulaz i izlaz podataka (>>, <<)

U jeziku C++, slično jeziku C, ne postoje naredbe i/ili operatori za ulaz i izlaz podataka, već se oni ostvaruju odgovarajućim klasama. Svakom izvoru (ulaznom toku) i odredištu (izlaznom toku) podataka pridružuje se jedan primerak jedne od tih klasa. Prenos podataka se ostvaruje pristupanjem tim primercima, isključivo pomoću metoda ili prijateljskih funkcija tih klasa.

Sve potrebne deklaracije vezane za klase za ulazne i izlazne tokove podataka nalaze se u zaglavljku **<iostream>**. Klasa za ulazne tokove naziva se **istream**, a za izlazne tokove **ostream**.

Objekat klase **istream** za pristup glavnom ulazu (tipično tastaturi) računara zove se **cin**. Objekat klase **ostream** za pristup glavnom izlazu računara (tipično ekranu) zove se **cout**. Ova dva objekta automatski se stvaraju na početku izvršavanja svakog programa.

Operatori za čitanje i pisanje podataka uz primenu ulazno-izlaznih konverzija definisani su za sve standardne tipove podataka preklapanjem operatora >> odnosno << (videti i odeljak 2.3.3). Prototipovi odgovarajućih operatorskih funkcija su:

```
istream& operator>>(istream& ut, T& t);                                odnosno
ostream& operator<<(ostream& it, const T& t);
```

ut je objekat koji predstavlja ulazni tok iz kojeg se čita, a *it* izlazni tok u koji se piše. Za sada u obzir dolaze samo **cin** i **cout**. Stvaranje drugih objekata za tokove (na primer za rad s datotekama), kao i ostale radnje s tokovima, objašnjene su u poglavljju 9.

Objekat *t* tipa *T* predstavlja odredište za podatke čitane iz ulaznih tokova, odnosno izvoru podataka za pisanje u izlazne tokove.

Vrednost obe funkcije je upućivač na tok *ut*, odnosno *it* koji je prvi operand tih operatorskih funkcija. Ovo omogućava lančanje operatora >> i << u izrazima, tj. prenošenje više podataka samo jednim izrazom. Pošto se ovi operatori grupišu sleva udesno, redosled prenosa podataka je po redosledu navođenja, sleva udesno.

Programer, daljim preklapanjem operatora >> i <<, može da obezbedi operatore za ulazno-izlazne konverzije za svoje tipove (klase) podataka. Ove funkcije ne mogu da budu metode klase za koju se prave, pošto prvi parametar (operand) nije tipa te klase. Moraju biti globalne, vrlo često prijateljske, funkcije.

Naglašava se da operatori >> i << mogu da se preklapaju i na druge načine bez ikakvog ograničenja u neke druge svrhe. Ovde opisani način treba ispoštovati ako se želi da se obezbedi čitanje i pisanje podataka klasnih tipova na isti način kao što se čitaju i pišu podaci standardnih tipova.

Evo primera definisanja operatora za izlaz kompleksnih brojeva:

```
#include <iostream>
using namespace std;
class Kompl {
    double re, im;
public:
    friend ostream& operator<< (ostream& it, const Kompl& z)
    { return it << '(' << z.re << ',' << z.im << ')'; }
};
int main ()
{
    Kompl z = {1, 0};
    cout << "z = " << z << endl;
}
```

U prijateljskoj funkciji `operator<<()` sadržaj kompleksnog broja se ispisuje kao dva realna broja unutar para oblih zagrada međusobno razdvojenih zarezom. Operatori `<<` se zvode pozivanjem operatorskih funkcija `ostream& operator<<(ostream&, char)`, odnosno `ostream& operator<<(ostream&, double)` iz klase `ostream`, zavisno od toga da li je desni operand tipa `char` ili `double`.

U glavnoj funkciji se tekst `"z = "` ispisuje pozivanjem operatorske funkcije `stream& operator<<(ostream&, char*)` iz klase `ostream`, a kompleksan broj `z` pozivanjem (globalne) prijateljske funkcije `ostream& operator<<(ostream&, Kompl&)` iz klase `Kompl`. Skreće se pažnja na to da se u izrazu za ispisivanje ne vidi formalna razlika između standardne i za klasne tipove podataka.

4.4 Preklapanje operatora (tip)

Preklapanje unarnog operatora za konverziju tipa (`tip`) je, pored konverzije pomoću konstruktora (videti odeljak 3.4.4.3), druga mogućnost definisanja konverzija tipova za klase i tipove podataka.

Operatorska funkcija za konverziju u tip `T` naziva se `operator T()`. Tip `T` može da bude standardni tip, klasni tip ili iz njih izvedeni tip (na primer: pokazivač na neki tip). Funkcija za konverziju mora da bude član klase iz koje konvertuje. Tip vrednosti funkcije treba da bude naveden u definiciji, nego se podrazumeva na osnovu imena funkcije. Potrebno je da se radi o preklapanju unarnog operatora metodom funkcija nema parametra.

Treba uočiti da je preklapanjem operatora za konverziju tipa moguće ostvariti konverziju tipa iz tipa klase kojoj pripada ta operatorska funkcija u bilo koji drugi, klasni ili standardni tip. S druge strane, konstruktorom konverzije (videti odeljak 3.4.4.3) moguća je konverzija iz bilo kog (klasnog ili standardnog) tipa u tip klase kojoj pripada konstruktor.

Operatorska funkcija `T::operator U()`, radi konverzije objekta `t` tipa `T` u tip `U`, može eksplisitno da se poziva kao i druge funkcije članovi (`t.operator U()`), kao operator `cast(U t)`, kao obična funkcija (`U(t)`) ili izrazom oblika `static_cast<U>(t)`.

Kada se u odeljak 3.4.4.3 pogledaju mogući načini pozivanja konstruktora konverzije, idući se da su zadnja tri oblika izraza potpuno ista kao i izrazi za pozivanje konstruktora konverzije. Prema tome, na osnovu oblika izraza ne može da se zaključi da li se data konverzija ostvaruje konstruktorom ili operatorskom funkcijom. Što i nije bitno, bitno je da se vrši konverzija tipa. Samo zbog lakoće pisanja i jasnoće izražavanja, može da se dà prednost funkcionalnoj notaciji (`U(t)`).

4.4.1 Automatska konverzija tipa

Ako se u pozivima funkcija tipovi argumenta ne slažu s tipovima odgovarajućih parametara obavlja se automatska konverzija tipa argumenta u tip parametra.

Automatska konverzija tipova između standardnih tipova je deo jezika C++. Pravila za konverziju tipova operanada za operatore čiji su operandi različitih standardnih tipova, objašnjena su u odeljku 1.3.3.

Konverziju između tipova, od kojih je bar jedan klasa, mora da definiše programer pomoću konstruktora konverzije (videti odeljak 3.4.4.3) ili preklapanjem operatora (`tip`) (videti odeljak 4.4). Po potrebi, te konverzije će da se koriste automatski.

Ne treba ispuštiti izvida da se operatori ostvaruju operatorskim funkcijama, pa se automatska konverzija koristi i u izrazima kada su operandi nekog operatora različitih tipova.

Konverzija za nestandardne tipove primenjuje se automatski samo ako je neposredna i jednoznačna.

Ako postoji konverzija `T(U&)` iz klase `U` u klasu `T` i konverzija `U(V&)` iz klase `V` u klasu `U`, neće se obaviti automatska konverzija iz tipa `V` u tip `T` pomoću `T(U(v))`, gde je `v` objekat tipa `V`. Naravno, takav izraz može da se napiše eksplisitno da bi se postigla konverzija iz tipa `V` u tip `T`. Nije bitno da li su te konverzije definisane konstruktorma (`T::T(U&)` i `U::U(V&)`), operatorskim funkcijama (`U::operator T()` i `V::operator U()`) ili jednim konstruktorom i jednom operatorskom funkcijom.

Višežnačnost konverzije može da nastane ako u klasi `T` postoji konstruktor konverzije `T(U&)`, a u klasi `U` operatorska funkcija `operator T()`. Obe konverzije su za konverziju iz tipa `U` u tip `T` i obe se eksplisitno pozivaju sa `T(u)`. Zbog toga izraz `t=u` ne može da se razreši, jer se ne zna koju od dve konverzije treba koristiti za konverziju objekta u u tip `T` pre dodelje vrednosti. Šta više, u ovakvim slučajevima konstruktor `T::T(U&)` ni na koji način ne može da se pozove eksplisitno, dok operatorska funkcija može da se poziva sa `u.operator T()`. Treba još naglasiti da prilikom stvaranja objekata (na primer: `T t(u)` ili `T* pt=new T(u)`) bez dvoumljenja se koristi konstruktor `T::T(U&)`, čak i ako postoji i operatorska funkcija `U::operator T()`.

Drugi potencijalni izvor problema je definisanje konverzije iz tipa `T` u tip `U` i obrnuto iz tipa `U` u tip `T`. Tada u nekim situacijama ne može da se odredi koju od te dve konverzije treba primeniti. Na primer, ako je `t` objekat tipa `T` i u tipa `U` i obe klase poseduju operatorsku funkciju `operator+(U)` za objekte svojih tipova, nije jasno da li izraz `t+u` treba da se tumači kao `t+T(u)` ili `U(t)+u`? Da je definisana samo jedna od konverzija, na primer `T(u)`, posmatrani izraz bi se jednoznačno tumačio kao `t+T(u)`. S druge strane, da je operatorska funkcija `operator+(U)` definisana samo u klasi `U`, posmatrani izraz bi se tumačio kao `U(t)+u`, bez obzira na postojanje konverzije u oba smera.

Upućivači kao parametri funkcija omogućavaju menjanje vrednosti upućivanih objekata iz funkcija. Ako se na odgovarajući argument primenjuje konverzija tipa pre pozivanja funkcije, u funkciju će se prenositi upućivač na privremen objekat koji je rezultat konverzije. Menjanjem vrednosti parametra unutar funkcije menjajuće se vrednost privremenog objekta, a ne objekta koji je naveden u pozivu funkcije. To sigurno nije ono što je programer zamislio! Zbog toga, prevodioci obično šalju opomenu programeru ako su generisali privremen objekat za neki upućivački parametar. Ukoliko se upućivač koristi samo radi uštete kopiranja argumenta u parametar, a ne s ciljem menjanja vrednosti argumenta, preporučuje se da se za takve parametra koristi modifikator `const`. Time se daje do znanja

prevodiocu da vrednost parametra neće biti promenjena unutar funkcije. Prema tome, nema potrebe da prevodilac šalje opomenu ako se u funkciju prenosi privremen objekat a ne onaj koji je naveo korisnik.

Program 4.1, na primeru klase kompleksnih brojeva, prikazuje definisanje konverzija za nestandardne tipove i njihovo korišćenje.

```
#include <iostream>
#include <cmath>
using namespace std;

class Kompl {
    double re, im;
public:
    Kompl (double r=0, double i=0) { re=r; im=i; } // Inic. ili konverzija.
    operator double() { return sqrt(re*re+im*im); } // Konverzija u double.
    friend double real (Kompl z) { return z.re; } // Realni deo.
    friend double imag (Kompl z) { return z.im; } // Imaginarni deo.
    Kompl operator+ (Kompl z) // Sabiranje.
    { z.re += re; z.im += im; return z; }
    friend ostream& operator<< (ostream& it, const Kompl& z)
    { return it << '(' << z.re << ',' << z.im << ')'; }
};

int main ()
{
    Kompl a (1, 2); cout << "a      = " << a          << endl;
    Kompl b = a;     cout << "b      = " << b          << endl;
    Kompl c = 5;    cout << "c      = " << c          << endl;
    cout << "a+b   = " << a + b        << endl;
    cout << "a+3   = " << a + (Kompl)3 << endl;
    cout << "|a|+3 = " << (double)a + 3 << endl;
    cout << "a+(3,4) = " << a + Kompl (3,4) << endl;
    cout << "dble(a) = " << (double)a << endl;
    double d=Kompl(3,4); cout << "d      = " << d          << endl;
}
```

Program 4.1 – Definisanje konverzija tipova (kompl.C)

Jedini konstruktor `Kompl(double=0,double=0)`, pošto može da se poziva i bez argumenata i s jednim argumentom, predstavlja i podrazumevani konstruktor i konstruktor konverzije iz tipa `double` u tip `Kompl`. Rezultat konverzije je kompleksan broj čiji je realni deo jednak konvertovanom podatku tipa `double`, a imaginarni deo jednak nuli.

Za konverziju kompleksnih brojeva (tip `Kompl`) u realne brojeve (tip `double`) postoji operatorska funkcija (metoda) `operator double()`. Ova konverzija nema opšteprihvaćenu definiciju u matematici. Ovde je uzeto da rezultat konverzije bude apsolutna vrednost konvertovanog kompleksnog broja.

Treba uočiti da pošto objekti tipa `Kompl` imaju vrlo jednostavna polja, za ovu klasu nisu potrebni konstruktor kopije, destruktur, ni preklapanje operatora = (videti odeljak 4.5).

Prijateljske funkcije `real()` i `imag()` dohvataju realni i imaginarni deo svojih kompleksnih parametara. U širem smislu, i oni mogu da se ubrajaju u funkcije za konverziju tipova, jer daju rezultat različitog tipa od tipa parametra. Međutim, mogu da budu pozvane samo eksplicitno, i zato se ne smatraju konverzijama u smislu izlaganja u ovom odeljku.

Operatorska funkcija `operator+()` izračunava zbir dva kompleksna broja. Pošto se parametar z inicijalizuje kopijom argumenta (drugog operanda operatora +), iskorišćen je za obrazovanje rezultata, a da se pri tom ne stvori neočekivani bočni efekat.

Za ispisivanje vrednosti kompleksnog broja postoji operatorska funkcija `operator<<()`.

Glavna funkcija (funkcija `main()` u programu 4.1) prikazuje neke mogućnosti korišćenja klase `Kompl`.

Na početku glavne funkcije definišu se tri promenljive. Promenljiva a se inicijalizuje pozivanjem konstruktora `Kompl()` s dva argumenta. Ti argumenti su rezultati automatskih standardnih konverzija iz tipa `int` u tip `double`. Promenljiva b se inicijalizuje (automatski generisanim) konstruktorom kopije vrednošću promenljive a. Za inicijalizaciju promenljive c konstruktor klase `Kompl` koristi se kao konverzija tipa iz `double` u tip `Kompl`. Naravno, prethodno se celobrojna konstanta automatskom primenom standardne konverzije pretvorila u realan broj dvostrukе tačnosti.

Pošto su definisane konverzije tipa između kompleksnih i realnih brojeva u oba smera, za izraz `a+3` ne bi bilo jasno da li rezultat treba da bude kompleksan ili realan. Ne bi pomoglo ni ako bi taj izraz bio napisan u naredbi za dodelu vrednosti bilo kog tipa (na primer: `Kompl z=a+3;`), jer bi se prvo izračunala vrednost izraza, pa tek posle toga bi se gledalo za šta je rezultat potreban. Ako treba, tada bi se primenila odgovarajuća konverzija. U programu 4.1 prikazane su obe mogućnosti. Ako `a+3` treba da bude kompleksan, treba da se piše kao `a+(Kompl)3`, ili funkcijском notacijom `a+Kompl(3)`. Za dobijanje realnog rezultata potrebno je pisati (`double`)`a+3` ili `double(a)+3`. U naredbi `Kompl z=double(a)+3;` taj realan rezultat bi se tek pri dodeli vrednosti pretvarao u ekvivalentni kompleksan broj, korišćenjem konstruktora `Kompl()`.

Kada u klasi `Kompl` ne bi postojala operatorska funkcija za konverziju `operator double()`, u izrazu `a+3` drugi sabirak bi se automatski pretvorio u ekvivalentni kompleksan broj i rezultat sabiranja bio bi kompleksan. To, međutim, nije slučaj s izrazom `3+a`, zato što se automatska konverzija tipa primeniće samo na prave parametre, a ne i na tekući objekat, predstavljen skrivenim parametrom operatorskih funkcija. Pošto je u posmatranom primeru funkcija `operator+()` metoda klase `Kompl`, prvi sabirak je tekući objekat! Za sabiranje, kao i za sve ostale binarne operatore bez bočnih efekata, obično nema razloga da se prvi i drugi operand tretiraju na različite načine. To se ne dešava ako se operatorska funkcija definiše kao prijateljska funkcija a ne metoda.

U poslednjem redu posmatrane glavne funkcije definiše se promenljiva tipa `double` koja se inicijalizuje kompleksnim brojem. Jasno je da će se taj kompleksan broj prethodno pretvoriti u realan broj operatorskom funkcijom `operator double()`.

4.5 Preklapanje operatora =

Operator = ima automatsko tumačenje za sve novouvedene klase podataka. To je kopiranje izvorišnog objekta u određeni objekat polje po polje. Ovo tumačenje je zadovoljavajuće samo za „jednostavne“ objekte kod se kojih sve informacije nalaze samo u poljima koja se vide u definiciji klase. Kod „složenih“ objekata neka od polja mogu da budu pokazivači koji pokazuju na delove memorije u dinamičkoj zoni. Informacije sadržane u dinamičkoj zoni neće biti prekopirane operatorom = koji se obezbeđuje automatski.

Svrha preklapanja operatora = je upravo da se obezbedi kopiranje celog složenog objekta, zajedno sa svim delovima koji se nalaze u dinamičkoj zoni memorije.

Operator = može da bude preklopljen samo pojedinačnim (ne **static**) metodama klase. Funkcija **operator=()** ne može da bude globalna funkcija izvan klase. To je jedino graničenje u odnosu na ostale preklopljene operatore. Da li ta funkcija stvarno dodeljuje rednost jednog objekta drugom, stvar je programera. Ne preporučuje se da to ne bude tako.

Prilikom sastavljanja funkcije **operator=()** ne treba ispustiti izvida da je u izrazu $=\rightarrow$ prvi operand a objekat koji postoji. Ako se delovi tog objekta nalaze u dinamičkoj zoni memorije, te delove treba osloboditi pre početka kopiranja delova objekta b. Izuvez, aravno, ako delovi objekta b mogu tačno da se uklape u odgovarajuće delove objekta a. O se retko dešava ako se radi o objektima koji su, na primer, ostvareni u obliku lančanih struktura. Pa čak i u slučaju relativno jednostavnih linearnih listi, retko se dešava da liste ove učestvuju u dodeli vrednosti imaju isti broj elemenata. Obično je jednostavnije, s primerske tačke gledišta, prvo u potpunosti uništiti stari sadržaj objekta a, i onda graditi novu listu koja je kopija sadržaja objekta b. Nije potrebno upuštati se u probleme skraćivanja ili produžavanja stare liste. Mada, ovo drugo može da ubrza rad programa. Cena kojom je to ubrzanje plaća je više uloženog programerskog npora, složeniji program i, verovatno, veći utrošak memorije za sam program.

Posebno treba обратити pažnju na, doduše retke, slučajeve izraza oblika a=a, tj. kada su evi i desni operand isti objekat (pažnja: ne dva objekta jednog sadržaja!). U ovakvoj situaciji uništavanje starog sadržaja odredišnog objekta uništilo bi i sadržaj izorišnog objekta, pošto su ta dva objekta, u stvari, jedan te isti objekat. Posle toga operacija bi se završila „praznim“ objektom a! Da je došlo do te situacije, može da se utvrdi upoređivanjem adrese izorišnog objekta (pravog parametra metode) i pokazivača **this** (skrivenog parametra) koji je adresa odredišnog objekta (tekućeg objekta). Ako su te dve adrese jednakе jednostavno ne treba raditi ništa.

Pošto je vrednost operatora za dodelu vrednosti za standardne tipove podataka *lvertnost*, taj operator se obično preklapa metodom čija je vrednost upućivač na primerke sopstvene klase. Naravno, to nije obavezno.

Ukoliko za neku klasu nije definisana metoda **operator=()** ona se automatski generiše za potrebe dodeljivanja kopije. Tako generisana operatorska funkcija kopira (samo) vrednosti svih polja date klase. Ovo, obično, zadovoljava kada među poljima nema pokazivača.

Ako su neka od polja klasnih tipova za njihovo kopiranje koriste se operatorske funkcije za dodelu vrednosti iz njihovih klasa.

Ako su neka od polja pokazivači, automatski generisana operatorska funkcija neće da kopira pokazivane podobjekte. Smatra se da dobro opremljena klasa koja koristi dinamičko dodeljivanje memorije mora obavezno da ima i operatorsku funkciju **operator=()** za dodelu kopije.

4.5.1 Inicijalizacija i dodela vrednosti

Inicijalizacija i dodela vrednosti su dve kvalitativno različite radnje, mada se i jednom i drugom radnjom dodeljuje vrednost jednog objekta drugom.

Inicijalizacija se vrši prilikom stvaranja novih objekata. Pri tome se ne prepostavlja ništa o prethodnom sadržaju memorijskog prostora u kome se vrši inicijalizacija. Rezultat inicijalizacije treba da je ispravan objekat koji posluje sve osobine objekata date klase.

4.5.1 Inicijalizacija i dodela vrednosti

Inicijalizacija se obavlja automatskim pozivanjem odgovarajućeg konstruktora. Konstruktor se bira na osnovu broja i tipova izraza u inicijalizatoru. U odsustvu inicijalizatora pozivaće se podrazumevani konstruktor. Ako je inicijalizator pri stvaranju objekta tipa T izraz istog tipa, pozivaće se konstruktor kopije **T::T(T&)**. Taj konstruktor treba da obezbedi kopiranje celog (složenog) objekta, a ne samo polja klase. Zbog toga se i naziva konstruktor kopije.

Dodela vrednosti se vrši prilikom izvršavanja operatora = pozivanjem operatorske funkcije **operator=()**. Podrazumeva se da je objekat s leve strane operatora inicijalizovan, tj. da poseduje sve osobine svoje klase. Zbog toga, u slučaju složenih objekata, dodela vrednosti može zahtevati uništavanje starog sadržaja objekta i stvaranje novog, uz kopiranje celog složenog objekta s desne strane operatora za dodelu vrednosti. Kao što je već naloži isti objekat.

Program 4.2 prikazuje primer inicijalizacije i preklapanja operatora = za objekte koji su niske.

```
#include <cstring>
using namespace std;

class Tekst {
    char* txt;
public:
    Tekst (const char* niz) { // Pokazivač na sam tekst.
        txt = new char [strlen(niz)+1];
        strcpy (txt, niz);
    }
    Tekst (const Tekst& tks) { // Konverzija iz char* u Tekst.
        txt = new char [strlen(tks.txt)+1];
        strcpy (txt, tks.txt);
    }
    Tekst& operator= (const Tekst& tks) { // Dodela vrednosti.
        if (this != &tks) {
            delete [] txt;
            txt = new char [strlen(tks.txt)+1];
            strcpy (txt, tks.txt);
        }
        return *this;
    }
    ~Tekst () { delete [] txt; }

int main () { // Stvaranje i inicijalizacija.
    Tekst a ("Dobar dan.");
    Tekst b = Tekst ("Zdravo.");
    Tekst c (a), d = b;
    a = b; // Dodela vrednosti.
}
```

Program 4.2 – Inicijalizacija i dodela vrednosti (tekst2.C)

Klasa **Tekst** je opremljena minimalnim brojem metoda koje su neophodne za ilustraciju upravo izložene materije, a da pri tome bude funkcionalno bezbedna.

Konstruktor konverzije **Tekst (char*)** posle dodelje memorije u dinamičkoj zoni (**new**) prekopira tekst (**strcpy()**) iz svog parametra u taj dodeljeni prostor. Konstruktor kopije **Tekst (Tekst&)** na sličan način inicijalizuje upravo stvarani objekat sadržajem svog parametra. Razlika je u tome što je parametar sada objekat tipa **Tekst**, a ne obična niska (**char***, odnosno **char[]**). Modifikator **const** uz tip parametra u oba slučaja označava da funkcija (konstruktor) ne menja vrednost parametra. To može da ima određene, povoljne, posledice na mestima pozivanja konstruktora. Skreće se još pažnja na to da nijedan od konstruktora ne prepostavlja ništa o prethodnom sadržaju objekta koji se inicijalizuje (stvara).

Operatorska funkcija **operator=()** za preklapanje operatora = za objekte tipa **Tekst** ima jedan pravi parametar, upućivač na objekat **tks** tipa **Tekst**. U slučaju da parametar funkcije nije tekući objekat (**this!=&tks**), uništi se stari sadržaj tekućeg objekta (**delete**), dodeli se memorija za novi sadržaj (**new**) i prekopira se tekst izvorišnog objekta (**strcpy()**). Novi, ili neizmenjeni sadržaj tekućeg objekta (***this**) je vrednost funkcije.

Na kraju, destruktur **~Tekst ()** oslobađa memoriju koja je dinamički dodeljena objektu. Ovaj destruktur ne bi smeо eksplicitno da se poziva, jer objekat ne ostavlja u ispravnom praznom stanju (ne postavlja vrednost pokazivača **txt** na nulu).

U glavnoj funkciji na kraju programa 4.2 prvo se definišu četiri objekta tipa **Tekst** koji se inicijalizuju nekim tekstovima, a u poslednjem redu se dodeljuje vrednosti jednog objekta drugom.

U prvoj naredbi objekat a inicijalizuje se pozivanjem konstruktora konverzije.

U drugoj naredbi prvo se konstruktorom konverzije stvara privremen objekat koji se posle konstruktorom kopije kopira u objekat b. Na kraju, privremen objekat se uništava destruktorem. Kao što se vidi, konačan rezultat je isti kao i u prvom redu, ali na znatno složeniji način. Zbog toga inicijalizaciju korišćenjem operatora = treba izbegavati (ta notacija može sugerisati i da se koristi operatorska funkcija za dodelu vrednosti, što nije slučaj!).

U trećoj naredbi oba objekta c i d se inicijalizuju korišćenjem samo konstruktora kopije, bez obzira na razlike u navođenju inicijalizatora.

Operatorska funkcija za dodelu vrednosti poziva se samo u poslednjoj naredbi, kada se na levoj strani nalazi već postojeći objekat koji ima izgrađeni sadržaj.

4.6 Preklapanje operatora []

Operator za indeksiranje [] može da se preklopi samo pojedinačnim (ne **static**) metodama klase. Prototip odgovarajućih operatorskih funkcija je:

```
tip_rezultata operator [] ( indeks ) ;
```

Preklapanje operatora [] u nekoj klasi omogućava korišćenje objekata izrazima oblika **obj [ind]**. Za razliku od standardnog indeksiranja, gde indeksni izraz **ind** mora da bude nekog celobrojnog tipa, kod preklapanja operatora [] parametar **ind** može da bude proizvodljnog tipa. Moguće primene su projektovanje klase čiji su objekti nizovi zadatih granica indeksa uz proveru vrednosti indeksa prilikom svakog pristupa ili za projektovanje „asocijativnih“ nizova kod kojih se elementima ne pristupa pomoću rednog broja već pomoću vrednosti

(ključa) u nekom od delova tih elemenata. Uopšteno rečeno, u klasama koje predstavljaju zbirke podataka među kojima treba odabrat jedan podatak na osnovu nekog kriterijuma.

Da bi izraz s preklopljenim operatorom [] mogao da se koristi kao prvi operand operatora za dodelu vrednosti (**obj [ind]=izr**), tip vrednosti operatorske funkcije **operator [] ()** treba da bude upućivač na podatke koje sadrži posmatrana klasa. Pošto je tada rezultat upućivač na deo objekta kome pripada operatorska funkcija **operator [] ()**, za bezbedno korišćenje klase potrebno je napraviti dve varijante te operatorske funkcije, jednu za promenljive i jednu za nepromenljive objekte (videti odeljke 2.4.4 i 3.3).

Naročito treba naglasiti da izraz s preklopljenim operatorom [] nije indeksiranje nego samo tako izgleda. Operatorska funkcija **operator [] ()** dejstvuje na objekat svoje klase, a ne na niz objekata. Zbog toga izraz **obj [i]** ne može da se zameni izrazom ***(obj+i)** po

uzorak **izraz * (niz+i)** umesto **izraz niz [i]** kod nizova podataka. Prirodno je, mislim, da interna struktura objekta bude niz elemenata nekog tipa, čiji se jedan element odabira kao vrednost te funkcije.

Program 4.3 prikazuje preklapanje operatora [] na primeru klase nizova zadatih opsega indeksa.

Privatna polja u klasi **Niz** su početna (poc) i krajnja (kraj) vrednost indeksa i upućivač (a) na elemente niza.

Na početku javnog dela nalaze se obavezni delovi klase, konstruktori, destruktur i operator za dodelu vrednosti. Ne postoji podrazumevani konstruktor, pa objekti tipa **Niz** ne mogu da se definišu bez inicijalizatora (pošto postoje drugi konstruktori, podrazumevani konstruktor se ne generiše automatski, pa će prevodilac prijaviti grešku ako se bilo gde u programu pojavi potreba za njim). Ako je pri stvaranju novog niza početni indeks veći od krajnjeg indeksa (**min>max**), program se prekida završnim statusom 1 (**exit(1)**). Za kopiranje izvorišnog niza konstruktor kopije i preklopljeni operator = oslanjaju se na usluge privatne metode **kopiraj ()**.

Poslednje dve metode u klasi **Niz** su dve varijante operatorske funkcije **operator [] ()**. Prvi će se pozivati za promenljive, a drugi za nepromenljive objekte tipa **Niz**. Tip rezultata obe varijante je upućivač na podatke tipa elemenata niza. Kod prve varijante to je upućivač na promenljive, a kod druge na nepromenljive podatke. Inače, sadržaj obe varijante je istovetan. Ako je indeks i izvan dozvoljenog opsega program se prekida završnim statusom 2 (**exit(2)**). Ako je indeks u redu on se izrazom **i-poc** presliku u opseg od 0 do **kraj-poc (≥0)** i standardnim indeksiranjem se dohvati odgovarajući element niza na koji pokazuje upućivač a.

U glavnoj funkciji prvo se definije objekat **niz** tipa **Niz** opsega indeksa od -5 do +5. Posle se elementima dodeljuju neke vrednosti uz pomoć operatorske funkcije za indeksiranje (**niz [i]**).

Pokušaj pisanja adresnog izraza obično dovodi do formalno neispravnog izraza. Ali ne uvek, pošto ako se operatoru binarni + i unarna * u klasi **Niz** dodelje neka tumačenja, izraz može da bude formalno ispravan. Naravno, značenje izraza zavisi od projektanta klase.

U drugom delu glavne funkcije definisan je nepromenljiv objekat **cniz** tipa **const Niz**. Inicijalizovan je kopijom sadržaja objekta **niz**, korišćenjem konstruktora kopije (bez obzira na korišćenje operatora = u inicijalizatoru). Kasnije pokušaje menjanja vrednosti tog niza, bilo u celini (**cniz=niz**) bilo u delovima (**cniz [2]=55**), prevodilac će prijaviti kao greške.

```

#include <iostream>
#include <cstdlib>
using namespace std;

class Niz {
    int poc, kraj;
    double* a;
    void kopiraj (const Niz&); // Opseg indeksa.
                                // Elementi niza.
                                // Kopiranje.
public:
    Niz (int min, int max) { // Konstruktori.
        if (min > max) exit (1);
        a = new double [(kraj=max)-(poc=min)+1];
    }
    ~Niz () { delete [] a; } // Destruktor.
    operator= (const Niz& niz) { // Dodela vrednosti.
        if (this != &niz) { delete [] a; kopiraj (niz); }
        return *this;
    }
    double& operator[] (int i) { // DOHVATANJE ELEMENTA.
        if (i<poc || i>kraj) exit (2);
        return a[i-poc];
    }
    const double& operator[] (int i) const {
        if (i<poc || i>kraj) exit (2);
        return a[i-poc];
    }
};

void Niz::kopiraj (const Niz& niz) { // Kopiranje.
    a = new double [(kraj=niz.kraj)-(poc=niz.poc)+1];
    for (int i=0; i<kraj-poc+1; i++) a[i] = niz.a[i];
}

int main () {
    Niz niz (-5, 5);
    for (int i=-5; i<=5; i++) niz[i]=i;
    *(niz+4) = 4; // GREŠKA: ne važi adresna aritmetika!
    const Niz cniz = niz;
    cniz = niz; // GREŠKA: menjanje celog nepromenljivog niza!
    cniz[2] = 55; // GREŠKA: menjanje dela nepromenljivog niza!
    double a = cniz[2]; // U redu: uzimanje je dozvoljeno.
}

```

Program 4.3 – Preklapanje operatora [] (niz1.C)

4.7 Preklapanje operatora ()

Operator za pozivanje funkcija () može da se preklopi samo pojedinačnim (ne static) metodama klase. Prototip odgovarajućih operatorskih funkcija je:

```
tip_rezultata operator () (arg1, ..., argN);
```

Skreće se pažnja na to da, za razliku od ostalih operatorskih funkcija, može da postoji proizvoljan broj, uključujući i nulu, parametara.

Preklapanje operatora () u nekoj klasi omogućava korišćenje objekata u izrazima oblika obj(arg1, ..., argN). Na primer, ako se radi o klasi polinoma ovo omogućava da se vrednost polinoma p iz te klase izračunava izrazom oblika p(x). Objekti u sebi nose koeficijente polinoma, pa je svaki objekat neka druga funkcija.

Program 4.4 prikazuje preklapanje operatora () na primeru jednostavne klase čiji objekti predstavljaju trigonometrijske funkcije oblika $a \cdot \sin(\omega \cdot x + \phi)$.

```

#include <cmath>
#include <iostream>
#include <iomanip>
using namespace std;

class Sin {
    double a, omega, fi;
public:
    explicit Sin (double a=1, double omega=1, double fi=0)
        : this->a = a, this->omega = omega, this->fi = fi {}
    double operator() (double x) { return a * sin (omega*x+fi); }
};

const double PI = 3.141592;

int main () {
    Sin sin (2, 0.5, PI/6);
    cout << fixed << setprecision(5);
    for (double x=0; x<=2*PI; x+=PI/12)
        cout << setw(10) << x
            << setw(10) << sin(x)
            << setw(10) << std::sin(x) << endl;
}

```

Program 4.4 – Preklapanje operatora () (sin1.C)

Privatna polja a, omega i fi u klasi Sin predstavljaju parametre sinusoide i postavljaju se jedinim konstruktorom. Podrazumevane vrednosti su birane tako da objekat predstavlja funkciju $\sin x$. Pošto se imena parametara konstruktora poklapaju s imenima polja do polja se može doći jedino pomoću skrivenog parametra this (this->a je polje a, a samo a je parametar a). Neki programeri koriste ovakvo rešenje kada se parametar metode koristi samo za postavljanje vrednosti polja.

Operatorska funkcija operator () (double) za datu vrednost parametra x izračunava vrednost sinusoide, koja zavisi i od vrednosti parametara u objektu. Skreće se pažnja na to da se vrednost nezavisne promenljive x ne smešta u objekat.

U glavnoj funkciji se tabeliraju vrednosti funkcija $2 \sin(x/2 + \pi/6)$ i $\sin x$ za sve vrednosti nezavisne promenljive x od 0 do 2π s korakom $\pi/12$. Za računanje vrednosti prve funkcije koristi se objekat sin tipa Sin, a druge funkcije standardnom bibliotečkom funkcijom. Zbog lokalne definicije objekta sin izraz sin(x) dovodi do pozivanja operatorske funkcije kojom je u klasi Sin preklopljen operator (). Za pozivanje bibliotečke funkcije mora da se koristi operator za razrešenje dosega (std::sin(x), ili samo ::sin(x)), pošto ne postoji drugi globalni identifikator sin, a std::sin je naredbom using uvezan u datotečki doseg).

4.8 Preklapanje operatora -> Δ

Operator za posredan pristup članovima klase -> može da se preklopi samo pojedinačnim (ne static) metodama klasa. Prototip odgovarajućih operatorskih funkcija je:

```
tip_rezultata operator -> ( ) ;
```

Bez obzira što je standardni operator -> binarni operator, operatorska funkcija kojom se preklapa ima jedan operand, objekat na koji se primenjuje. Štaviše operator se piše iza operanda, što je karakteristika postfiksnih operatora.

Ako se operator -> preklopi u nekoj klasi, izraz *obj->član* se tumači kao *(obj.operator->())->član*. Zbog toga rezultat te operatorske funkcije mora da bude pokazivač na objekat klase koja sadrži navedeni član. Kao krajnji rezultat izvršiće se pristup navedenom članu odabranog objekta.

Preklapanje operatora -> omogućava obrazovanje klase čiji se objekti ponašaju kao „pametni” pokazivači, pokazivači koji pored pristupanja objektu urade i neki dodatni posao. Na primer, broje pristupe objektima, ili proveravaju ispravnost pokazivača pre samog pristupanja objektu.

Program 4.5 prikazuje preklapanje operatora -> na primeru „pametnog” pokazivača koji broji pristupe njemu pridruženom objektu.

```
#include <iostream>
using namespace std;

struct Obj { double x, y; }; // Korišćeni objekat.

class PokObj { // Klasa „pametnih” pokazivača.
    Obj* p; // Pokazivač na pridruženi objekat.
    int n; // Broj pristupa priaruženom objektu.

public:
    PokObj (Obj* obj) { n = 0; p = obj; } // Inicijalizacija.
    Obj* operator-> () { n++; return p; } // Pristup članu.
    Obj& operator* () { n++; return *p; } // Pristup objektu.
    int operator+ () const { return n; } // Broj pristupa.
};

int main () {
    Obj a;
    PokObj pa = &a; // „Pametan” pokazivač.
    pa->x = pa->y = 0;
    cout << +pa << endl; // Piše se: 2
    Obj b = *pa;
    cout << +pa << endl; // Piše se: 3
    Obj* pb = &b;
    pb->x = pb->y = 5;
    *pa = *pb;
    cout << +pa << endl; // Piše se: 4
}
```

Program 4.5 – Preklapanje operatora -> (pametan.C)

„Pametni” pokazivači tipa PokObj mogu da pokazuju na objekte tipa Obj, pošto u sebi imaju pokazivač na takve podatke. Konstruktor klase PokObj samo postavlja brojač pri-

stupa n na nulu i uskladišti pokazivač na pridruženi objekat čija adresa se zadaje kao parametar konstruktora u polje p tipa Obj*.

Operatorska funkcija **operator->()** samo vraća adresu pridruženog objeka, uz povećavanja brojača pristupa za 1. Na osnovu rezultata funkcije moći će da se pristupi pojedinim članovima pridruženog objekta.

Da bi se omogućio pristup pridruženom objektu u celini, preklopljen je i unarni operator * tako da vraća upućivač (Obj&) na pridruženi objekat. Naravno, i ta operatorska funkcija povećava brojač pristupa za jedan. Napominje se da preklapanje unarnog operatora * podleže opštim uslovima za preklapanje operatora. Drugim rečima, može da se preklopi pojedinačnim (ne static) metodama bez parametara ili globalnim (verovatno prijateljskim) funkcijama s jednim parametrom.

Poslednja uslužna metoda, kojom je preklopljen unarni operator +, samo vraća trenutnu vrednost brojača pristupa.

Na početku glavne funkcije definiše se jedan objekat a tipa Obj, i „pametan” pokazivač pa koji se inicijalizuje adresom objekta a. Posle toga, izrazom pa->x pristupa se polju x tog objekta. Oblik izraza uopšte ne razlikuje od pristupanja istom polju pomoću „običnog” pokazivača (videti izraz pb->x u drugom delu glavne funkcije). Isto važi i za izraz kojim se pristupa celom objektu „pametnim” pokazivačem (*pa), odnosno „običnim” pokazivačem (*pb).

4.9 Preklapanje operatora new i delete Δ

Upotreba operatora **new** i **delete** prouzrokuje pozivanje globalnih operatorskih funkcija **::operator new()** i **::operator delete()** za pojedinačne podatke, odnosno **::operator new[]()** i **::operator delete[]()** za nizove podataka. Njihovo dejstvo je opisano u odeljku 2.3.4. Dok se drugače ne kaže, te funkcije koriste se ne samo za podatke standardnih tipova, već i za objekte klasnih tipova.

Globalne operatorske funkcije za ostvarivanje operatora **new** i **delete** mogu da se preklapaju radi omogućavanja programeru da projektuje svoj sistem za upravljanje dinamičkom dodelom memorije. Ove operatorske funkcije su obavezno zajedničke (static) metode svojih klasa. Štaviše, modifikator **static** i ne treba da bude stavljen na početak definicije, on se za njih podrazumeva.

Operatorske funkcije **operator new()** i **operator new[]()** mogu da imaju proizvoljan broj parametara od kojih je prvi tipa **size_t** (celobrojan tip definisan u zaglavlju **<cstddef>**, odnosno **<stddef.h>**) i predstavlja veličinu tražene memorije u dinamičkoj zoni memorije, izraženu u bajtovima. Značenje ostalih parametara može da bira programer po želji. Prototip operatorskih funkcija **operator new()** i **operator new[]()** je:

```
void * operator new (size_t veličina, T2 par2, ..., TN parN); ili
void * operator new [] (size_t veličina, T2 par2, ..., TN parN);
```

Opšti oblici izraza kojima se pozivaju te funkcije su:

```
new ( arg2 , ... , argN ) naziv_tipa ( izraz , ... , izraz ) odrhosno
new ( arg2 , ... , argN ) naziv_tipa [ dužina ]
```

Prvi argument, koji će zamjeniti parametar **veličina**, je **sizeof (naziv_tipa)**. Odnosno **dužina*sizeof(naziv_tipa)**. Argumenti **arg2, ..., argN** navedeni u

ethodnim izrazima zameniče, redom, parametre *par2*, ..., *parN* koji su tipa *T2*, ..., *TN*. vrednost funkcije treba da je pokazivač na dodeljeni memoriski prostor.

Niz *izraza* unutar para oblih zagrada (ako postoji) predstavlja inicijalizator za izbor instruktora kojim će se inicijalizovati sadržaj dodeljenog memoriskog prostora za jedinačne objekte (videti i odeljak 3.4.2).

Izraz *dužina* predstavlja broj elemenata niza objekata za koje se dodeljuje memorija, a vektore ne mogu da se navedu inicijalizatori, već se koristi podrazumevani konstruktor.

U klasi može da postoji više operatorskih funkcija ***new***, naravno različitih parametara, rugim rečima, dozvoljeno je preklapanje imena operatorske funkcije ***new***.

Prototipovi operatorskih funkcija **operator delete()**, odnosno **operator delete[]()** su:

```
void operator delete (void * pokazivač);           ili
void operator delete (void * pokazivač, size_t veličina); odnosno
void operator delete[] (void * pokazivač);          ili
void operator delete[] (void * pokazivač, size_t veličina);
```

Funkcije treba da oslobode memoriju od *veličine* bajtova na mestu u dinamičkoj zoni koje je određeno *pokazivačem*. U slučaju varijanti s jednim parametrom, operatorske funkcije **operator delete()**, odnosno **operator delete[]()** moraju da su u stanju da odrede veličinu memorije koju treba da oslobode. Globalne funkcije **::operator delete()** i **::operator delete[]()** su s jednim parametrom.

U klasi sme da postoji samo po jedna operatorska funkcija **delete** za pojedinačne potake i jedna za nizove podataka.

Neka je u klasi *T* izvršeno preklapanje operatora ***new*** i ***delete***. Standardni (globalni) operatori ***new*** i ***delete*** mogu da se pozivaju unutar dosega klase *T* ili eksplisitno, primenom unarnog oblika operatora za razrešenje dosega (**::new T** ili **::delete pt** za pojedinačne podatke, odnosno **::new T [duž]** ili **::delete [] pt** za nizove podataka) ili kada se memorija dodeljuje ili oslobođa za objekte koji nisu tipa *T*.

4.10 Nabranja i preklapanje operatora △

Nabranja su nestandardni prosti tipovi koji spadaju u grupu celobrojnih tipova (videti odeljak 2.2.6). Njihova osnovna namena je definisanje skupova simboličkih konstanti čiom upotreblom teksta programu postaje čitkiji, ali za koje pridružene vrednosti nisu bitne. Na primer, kada takve konstante predstavljaju boje, najčešće nije važno koja je vrednost pridružena konstanti BELA. Zbog toga za njih nisu definisane aritmetičke operacije, već samo međusobna dodela vrednosti podataka istog tipa nabranja, konverzija u tip **int** (koja se po potrebi radi i automatski) i konverzija iz celobrojnih tipova u nabranje (koja se ne radi automatski).

Ako za neki nabrojan tip aritmetičke operacije imaju smisla, programer može te radnje da definiše preklapanjem operatora. Ovo je dozvoljeno zato što su nabranja nestandardni tipovi koje definiše programer.

Operatorske funkcije za nabranja su, naravno, uvek globalne funkcije i ne mogu da se preklapaju operatori za koje odgovarajuće operatorske funkcije moraju da budu metode klase (videti odeljak 4.1). To su operatori (*tip*)**=**, **()**, **[]**, **->**, ***new*** i ***delete***. Konverzija

tipa i dodata vrednosti su automatski definisane na zadovoljavajući način. Preostali navedeni operatori su takve prirode da njihov uobičajeni smisao nije primeren nabranjima. (**==**, **!=**, **<**, **<=**, **>** i **>=**) najčešće nema potrebe preklapati, jer oni upoređuju celobrojne vrednosti pridružene pojednim nabrojanim konstantama. Za logično definisana nabranja to obično zadovoljava. Naravno, u nekim neuobičajenim slučajevima mogu čak i ti operatori da se preklapaju (na primer, kada su konstante A, B, C i D nabrojane po navedenom redosledu, ali potrebno je da bude **B<D<A<C**!).

Program 4.6 prikazuje preklapanje operatora za nabrojane tipove na primeru tipa koji predstavlja dane u nedelji.

```
#include <iostream>
using namespace std;

enum Dan { PO, UT, SR, CE, PE, SU, NE};

inline Dan operator+ (Dan d, int k)
{
    k = (int(d) + k) % 7; if (k < 0) k += 7; return Dan (k); }

inline Dan operator- (Dan d, int k) { return d + -k; }

inline Dan& operator+= (Dan& d, int k) { return d = d + k; }

inline Dan& operator-= (Dan& d, int k) { return d = d - k; }

inline Dan& operator++ (Dan& d) { return d = Dan (d<NE ? int(d)+1 : PO); }

inline Dan& operator-- (Dan& d) { return d = Dan (d>PO ? int(d)-1 : NE); }

inline Dan operator++ (Dan& d, int) { Dan e(d); ++d; return e; }

inline Dan operator-- (Dan& d, int) { Dan e(d); --d; return e; }

ostream& operator<< (ostream& it, Dan dan) {
    char* dani[] = {"pon", "uto", "sre", "cet", "pet", "sub", "ned"};
    return it << dani[dan]; }

int main () {
    for (Dan d=PO; d<NE; ++d)
        cout << d << ' ' << d+2 << ' ' << d-2 << endl;
}
```

Program 4.6 – Nabranja i preklapanje operatora (*dani.C*)

Za dane u nedelji ima smisla nalaženje narednog ili prethodnog dana, kao i određivanje dana koji je za nekoliko dana posle ili pre zadatog dana. Treba pri tome voditi računa o tome da se dani u nedelji ciklički ponavljaju. Pošto su u nabranju Dan u programu 4.6 pojedinim konstantama od PO do NE pridružene vrednosti od 0 do 6, cikličko ponavljanje pri sabiranju i oduzimanju najlakše se obezbeđuje kongruencijom po modulu 7. Za pozitivne rezultate računanjem ostatka (operatorom **%**) dobija se ispravna vrednost. U slučaju negativnih brojeva dobijaju se ostaci od -1 do -6, pa je potrebno na izračunatu vrednost dodati 7 (na primer, -1 je dan ispred ponedeljka, a to je nedelja, kome je pridružena vrednost $-1+7=6$).

Operatorska funkcija **operator+()** ostvaruje opisani postupak za dodavanje celog broja *k* na broj dana *d*. Skreće se pažnja na to da je broj dana *d* morao eksplisitnom kon-

verzijom (`int(d)`) da se pretvorí u tip `int`, jer da je napisano samo `d+k`, pozivala bi se operatorska funkcija koja se upravo definiše, a to bi dovelo do beskonačne rekurzije!

Treba još napomenuti da se ova operatorska funkcija neće pozivati za slučaj izraza `k+d`, već će se obaviti celobrojno sabiranje i rezultat će biti tipa `int`. Da bi se i u tom slučaju dobio rezultat tipa `Dan`, trebalo bi definisati i operatorsku funkciju

```
inline Dan operator+ (int k, Dan d) { return d + k; }
```

S druge strane, čak i bez dodavanja ove funkcije, može da se piše izraz `d+d`, kada će se, posle automatske konverzije drugog operanda u tip `int`, pozivati operatorska funkcija `operator+(Dan, int)` i dobice se rezultat tipa `Dan`. Ova besmislica bi mogla da se izbegne kada bi se pravila klasa, a ne nabranjanje za dane.

Oduzimanje celog broja `k` od broja dana `d` u programu 4.6 u operatorskoj funkciji `operator-()` je ostvareno dodavanjem `-k` uz pomoć već razmatrane operatorske funkcije za sabiranje. Slično tome, operatorske funkcije za sabiranje i oduzimanje s bočnim efektima (`+=` i `-=`) ostvarene su pozivanjem odgovarajućih operatorskih funkcija bez bočnih efekata.

Nalaženje narednog i prethodnog dana ostvareni su preklapanjem operatora `++` i `--`. Cikličko ređanje dana je obezbeđeno uslovnim dodavanjem ili oduzimanjem jedan i time je izbegnuto računanje ostatka pri deljenju sa 7, što traje relativno dugo u odnosu na samo sabiranje. To je razlog zašto, na primer, u operatorskoj funkciji za prefiksni oblik operatora `++(operator++(Dan))` nije napisano jednostavnije samo `return d+=1`. U operatorskim funkcijama za postfiksni oblik operatora `++` i `--` (s fiktivnim dodatnim parametrom tipa `int`) pozivaju se operatorske funkcije za prefiksne oblike istih operatora. Da bi rezultat mogao da bude vrednost operanda pre promene početna vrednost se privremeno čuva u lokalnoj promenljivoj `e`.

Poslednja operatorska funkcija u programu 4.6 preklapa operator `<<` radi ispisivanja podataka tipa `Dan` u obliku trostolovnih oznaka pojedinih dana. Pošto sadrži niz s inicijalizatorom, nije mogla biti ugrađena metoda (nije `inline`).

Na kraju, glavna funkcija prikazuje upotrebu definisanih operatorskih funkcija. Skreće se pažnja na to da, zbog cikličkog ređanja dana, u naredbi `for` ne bi smeo da se piše uslov `d<=NE`, jer je taj uslov uvek ispunjen, pa bi se dobio beskonačan ciklus. Dakle, najveća dužina ciklusa je 6. Ako to ne odgovara, prolaska kroz ciklus treba brojati zasebnom celobrojnom promenljivom.

4.11 Zadaci

4.11.1 Obrada vremenskih intervala

Zadatak:

Napisati na jeziku C++ klasu za obradu podataka koji predstavljaju vremenske intervale s rezolucijom od jedne sekunde. Napisati na jeziku C++ program za prikazivanje mogućnosti te klase.

Rešenje:

Uobičajeni način prikazivanja vremenskih intervala je u obliku broja sata, minuta i sekundi. Ovaj oblik može da se usvoji i za interni oblik predstavljanja podataka o vremenskim

4.11.1 Obrada vremenskih intervala

intervalima. To međutim uvodi odredene probleme pri izvođenju aritmetičkih operacija (sabiranje, oduzimanje i slično). Mnogo je jednostavnije da se interni oblik sastoji samo od ukupnog broja sekundi u datom vremenskom intervalu, i da se predvide odgovarajuće funkcije za konverzije između tog oblika i oblika sastavljenog od sata, minuta i sekundi. To je upravo i učinjeno u ovde opisanoj klasi `Vreme`.

Program 4.7 predstavlja definiciju klase `Vreme`.

Klasa ima samo jedno polje, t tipa `long`, koje predstavlja vrednost vremenskog intervala u sekundama.

Prvu grupu metoda u javnom delu klase čine konstruktori klase `Vreme`. Konstruktor `Vreme(int, int, int)` stvara primerke klase `Vreme` na osnovu broja sati, minuta i sekundi. Drugi konstruktor `Vreme(long)` objekte tipa `Vreme` stvara na osnovu ukupnog broja sekundi. On je konstruktor konverzije, a istovremeno i podrazumevani konstruktor pošto njegov parametar ima podrazumevanu vrednost. Pošto klasa `Vreme` za smeštanje sadržaja svojih primeraka ne koristi dinamičko dodeljivanje memorije, nisu joj potrebni konstruktor kopije ni destruktorni.

Drugu grupu funkcija čine funkcije za dohvatanje delova ili celog sadržaja objekata tipa `Vreme` koji se navodi kao parametar. U širem smislu i one predstavljaju neku vrstu konverzije tipa. Takve funkcije se najčešće ostvaruju kao prijateljske funkcije, da bi mogle da se pozivaju funkcijom notacijom. Tu su funkcije za dohvatanje broja sati, minuta i sekundi (`sat()`, `min()` i `sek()`), odnosno ukupnog broja sekundi (`Sek()`) u njihovom parametru.

Skreće se još pažnja na to da se u slučaju funkcija `sek()` i `Sek()` ne radi o preklapanju imena funkcija, jer su `sek` i `Sek` dva različita identifikatora (vodi se računa o malim i velikim slovima!). Preklapanje imena ne bi ni bilo moguće, jer obe funkcije imaju po jedan parametar međusobno jednakog tipa. Obrazovanje funkcija čija se imena razlikuju samo po korišćenju malih i velikih slova treba izbegavati, naročito za funkcije čija imena imaju spoljašnje povezivanje. Neki (danas već vrlo retki) programi za povezivanje (*linker*) nisu u stanju da razlikuju mala i velika slova!

Treću grupu funkcija čine operatorske funkcije za preklapanje unarnih operatora. Pošto za njih ostvarivanje u obliku prijateljskih funkcija nema nikakvu prednost nad ostvarivanjem u obliku metoda, sve su ostvarene kao metode.

Što se samih operatora tiče preklapanje je vršeno operacijama koje se prirodno nameću. Unarni `+` ne menja tekući objekat, već samo vraća njegovu vrednost (`*this`). Unarni `-` vraća ukupan broj sekundi u tekućem objektu s izmenjenim predznakom (`-t`) kao privremen objekat tipa `Vreme`, stvoren konstruktorom konverzije (`Vreme(-t)`).

Operatori za povećavanje `(++)` i smanjivanje `(--)` menjaju vrednost svog operanda za jednu sekundu, a vrednosti izraza su vrednost operanda (tekućeg objekta, `*this`) posle ili pre promene, zavisno od toga da li se radi o prefiksnom ili postfiksnom obliku. Prefiksni oblici svoje rezultate daju kao *referenci* (`Vrem&`), kao što to rade ti operatori za standardne tipove podataka.

Narednu grupu čine operatorske funkcije za binarne aritmetičke operatore bez bočnih efekata. Ostvarene su kao prijateljske funkcije, jer to omogućava automatsku konverziju tipa oba operanda po potrebi (videti odeljak 4.4.1). Usvojeno je da parametri funkcija budu upućivači iz principijelnih razloga. Objekti klasnih tipova mogu da budu vrlo složene strukture, pa njihovo prenošenje po vrednosti može da bude skupo. Modifikator `const` sprečava neočekivane promene vrednosti parametara od strane funkcija. Prenošenje pomocu

Definicija klase vremenskih intervala (Vreme).

```

#include <iostream>
using namespace std;

class Vreme {
public:
    long t; // Vreme u sekundama.
    Vreme (int h, int m, int s) { t = ((h * 60L + m) * 60 + s); } // Sastavljanje vremena.
    Vreme (long s=0) { t = s; } // Sekunde u vreme.

    friend int sat (Vreme v) { return v.t / 3600; } // Sati u vremenu.
    friend int min (Vreme v) { return (v.t/60)*60; } // Minuti u vremenu.
    friend int sek (Vreme v) { return v.t % 60; } // Sekunde u vremenu.
    friend long Sek (Vreme v) { return v.t; } // Vreme u sekundama.

    Vreme operator+ () const { return *this; } // +t
    Vreme operator- () const { return Vreme (-t); } // -t
    Vreme operator++ () { ++t; return *this; } // ++t
    Vreme operator++ (int) { Vreme w (*this); t++; return w; } // t++
    Vreme operator-- () { --t; return *this; } // --t
    Vreme operator-- (int) { Vreme w (*this); t--; return w; } // t--

    friend Vreme operator+ (const Vreme& v, const Vreme& w) // v + w
    { return Vreme (v.t + w.t); }
    friend Vreme operator- (const Vreme& v, const Vreme& w) // v - w
    { return Vreme (v.t - w.t); }
    friend Vreme operator* (const Vreme& v, int k) // v * k
    { return Vreme (v.t * k); }
    friend Vreme operator/ (const Vreme& v, int k) // v / k
    { return Vreme (v.t / k); }

    typedef const Vreme CV;
    Vreme& operator+= (CV& v) { t += v.t; return *this; } // t += v
    Vreme& operator-= (CV& v) { t -= v.t; return *this; } // t -= v
    Vreme& operator*= (int k) { t *= k; return *this; } // t *= k
    Vreme& operator/= (int k) { t /= k; return *this; } // t /= k

    friend int operator< (CV& v, CV& w) { return v.t < w.t; } // v < w
    friend int operator<= (CV& v, CV& w) { return v.t <= w.t; } // v <= w
    friend int operator> (CV& v, CV& w) { return v.t > w.t; } // v > w
    friend int operator>= (CV& v, CV& w) { return v.t >= w.t; } // v >= w
    friend int operator== (CV& v, CV& w) { return v.t == w.t; } // v == w
    friend int operator!= (CV& v, CV& w) { return v.t != w.t; } // v != w

    friend ostream& operator<< (ostream& it, CV& v) { // Ispisivanje.
        Vreme w (v);
        if (w.t < 0) { it << '-'; w = -w; }
        return it << sat (w) << ':' << min (w) << ':' << sek (w);
    }

    friend istream& operator>> (istream& ut, Vreme& v) { // Čitanje.
        int h, m, s; char z; ut >> h >> z >> m >> z >> s;
        v = Vreme (h, m, s); return ut;
    }
}

```

Program 4.7 – Definicija klase vremenskih intervala (vreme.h)

4.11.1 Obrada vremenskih intervala

vrednosti, što je korišćeno u ranijoj grupi funkcija za dohvatanje delova vremena, prihvativljivo je samo za klase čiji primerci zauzimaju mali memorijski prostor. Posmatrana klasa Vreme spada u tu grupu, jer sadrži samo jedno polje tipa long.

Binari + i - izračunaju zbir i razliku dva vremena. Pošto proizvod i količnik dva vremencima brojem, koji se smatra veličinom bez fizičke dimenzije. Skreće se pažnja na stvaranje privremenog objekta koji se vraća kao vrednost funkcije na osnovu celobrojnog izraza korišćenjem konstruktora Vreme (long).

Na početku sledeće grupe funkcija, naredbom **typedef**, uveden je identifikator CV kao skraćenica za tip „nepromenljiv objekat tipa Vreme“ (**const Vreme**). Pošto je većina parametara funkcija u klasi Vreme tipa **const Vreme&**, na ovaj način se dobija izvesna ušteda programerskog truda za navođenje tipa parametra, jer dovoljno je da se piše samo CV.

U ovoj grupi se nalaze operatorske funkcije za aritmetičke operatore s bočnim efektima. I ovde se držalo uobičajenih tumačenja tih operatorka za standardne tipove podataka. Složeni operatori za dodelu vrednosti (+=, -=, *= i /=) vrše analogne radnje kao i odgovarajući binarni aritmetički operatori uz smeštanje rezultata, kao bočni efekat, u prvi operand tih operatorka. Nije predviđeno preklapanje operatorka za običnu dodelu vrednosti (=), jer automatsko tumačenje tog operatorka za novouvedene klase je zadovoljavajuće za ovde posmatranu klasu Vreme.

Kao što je to uobičajeno za operatore s bočnim efektima, posmatrane operatorske funkcije ostvarene su kao metode. Operand koji podleže bočnom efektu je skriveni operand (*this) tih funkcija. Vrednosti ovih metoda su upućivači (Vreme&) na primerke svojih klasa, po uzoru na osobine odgovarajućih operatorka za podatke standardnih tipova.

Kako upoređivanje vremenskih intervala ima smisla, u preposlednjoj grupi u programu 4.7 izvršeno je preklapanje i relacijskih operatorka (<, <=, >, >= i !=) odgovarajućim operatorskim funkcijama. Pošto ne menjaju vrednosti svojih operanada, ostvarene su kao prijateljske funkcije. Skreće se pažnja na to da svih šest operatorka moraju da se preklope odvojeno. Iz tumačenja jednog operatorka automatski se ne izvodi tumačenje ostalih. Na programeru je da to preklapanje obavi na logički usaglašeni način.

Na kraju, izvršeno je preklapanje i operatorka za izlaz (<<) i ulaz (>>) podataka s ulazno-izlaznom konverzijom za potrebe čitanja i pisanja podataka tipa Vreme.

Ispisivanje vremenskog intervala (objekta tipa Vreme) podrazumeva prikazivanje broja sati, minuta i sekundi međusobno razdvojenih dvema tačkama (:). U slučaju negativne vrednosti negativan predznak se stavlja samo ispred broja sati, tj. ispred celokupne ispisane vrednosti (na primer: -3:30:0).

Citanje vrednosti vremenskog intervala podrazumeva čitanje tri cela broja (tip **int**). Da bi format čitanja bio usaglašen s formatom ispisivanja iz prethodnog pasusa, između svaka dva broja pročita se i po jedan znak (tip **char**). Jednostavnosti radi, ne proverava se da li je taj znak stvarno dve tačke (:). Od pročitanih brojeva se konstruktorom Vreme() stvara objekat tipa Vreme koji se, kao rezultat, smešta u drugi operand operatorka >>. U slučaju negativne vrednosti (na primer: -3.5 h), potrebno je ispred sve tri komponente vremena staviti negativan predznak (na primer: -3:-30:0). Treba još naglasiti da unutar funkcije operator>>() drugi operand operatorka >> na tri mesta je tipa **int** i na dva mesta tipa **char**, pa se na tim mestima koristi tumačenje tog operatorka za čitanje celobrojnih, odnosno znakovnih podataka.

Sve gore pomenute funkcije su vrlo jednostavne, pa je prirodno da budu ugrađene funkcije. To se za sve njih podrazumeva pošto se definicije nalaze unutar definicije klase vreme. Naglašava se da, uprkos definisanja prijateljskih funkcija unutar definicije klase, one nisu članovi klase i njihovi identifikatori nemaju klasni već datotečki doseg.

Možda bi bilo bolje da su operatorske funkcije za ulaz i izlaz ostvarene kao prave funkcije, ali to ovde nije učinjeno. Procenjeno je da su one po složnosti na granici prihvataljivosti za ugrađene funkcije. U svakom slučaju, treba biti oprezan sa zahtevima za ugrađenim funkcijama. Šteta može da bude veća od koristi.

Pošto su sve funkcije u potpunosti definisane unutar definicije klase, za klasu vreme nije potrebna zasebna datoteka (tipa .C) s definicijama pravih funkcija.

Program 4.8 predstavlja primer glavne funkcije za prikazivanje nekih mogućnosti gore opisane klase Vreme.

```
// Program za ispitivanje klase Vreme.

#include "vreme.h"
#include <iostream>
using namespace std;

int main () {
    for (;;) {
        int k; cin >> k;
        if (! k) break;
        Vreme t1, t2; cin >> t1 >> t2;
        long s1 = Sek (t1), s2 = Sek (t2);
        cout << "k      = " << k << endl;
        cout << "t1, t2 = " << t1 << ", " << t2 << endl;
        cout << "s1      = " << s1 << " (" << Vreme (s1) << ") \n";
        cout << "s2      = " << s2 << " (" << Vreme (s2) << ") \n";
        cout << "- t1   = " << - t1 << endl;
        cout << "+t1   = " << +t1 << endl;
        cout << "t2 -- = " << t2 -- << endl;
        cout << "t1      = " << t1 << " (" << Sek (t1) << ") \n";
        cout << "t2      = " << t2 << " (" << Sek (t2) << ") \n";
        cout << "t1 + t2 = " << t1 + t2 << endl;
        cout << "t1 - t2 = " << t1 - t2 << endl;
        cout << "t1 == t2 = " << (t1 == t2) << endl;
        cout << "t1 > t2 = " << (t1 > t2) << endl;
        cout << "t1 * k   = " << t1 * k << endl;
        cout << "t2 / k   = " << t2 / k << endl;
        cout << "s1+t2*k = " << Vreme (s1) + t2*k << " (" << s1 + Sek (t2*k) << ") \n";
    }
}
```

Program 4.8 – Primer korišćenja klase Vreme (vremet.C)

Posebno se skreće pažnja na način korišćenja operatora, koji se ni najmanje ne razlikuje od načina korišćenja operatora pri radu sa standardnim tipovima podataka. Pošto se radi o programu koji je samo primer i koji nema nekog posebnog smisla kao celina, izrazi su korišćeni neposredno kao operandi u izrazima za ispisivanje. Treba uočiti, da relacijski

4.11.1 Obrada vremenskih intervala

operatori imaju niže prioritete od operatora << i >>, pa izrazi koji ih sadrže morali su da se stave u zagrade.

Rezultat 4.1 prikazuje primer rada programa 4.8.

```
% vremet <vremet.pod
k      = 3
t1, t2 = 2:30:40, -1:20:30
s1      = 9040  (2:30:40)
s2      = -4830  (-1:20:30)
- t1   = -2:30:40
++ t1   = 2:30:41
t2 -- = -1:20:30
t1      = 2:30:41  (9041)
t2      = -1:20:31  (-4831)
t1 + t2 = 1:10:10
t1 - t2 = 3:51:12
t1 == t2 = 0
t1 > t2 = 1
t1 * k  = 7:32:3
t2 / k  = -0:26:50
s1+t2*k = -1:30:53  (-5453)
```

Rezultat 4.1 – Obrada vremenskih intervala programom 4.8

4.11.2 Obrada polinoma

Zadatak:

Napisati na jeziku C++ klasu za obradu polinoma s realnim koeficijentima i realnim argumentom. Napisati na jeziku C++ program za prikazivanje mogućnosti te klase.

Rešenje:

Polinomi mogu da se predstavljaju redom polinoma i nizom koeficijenata. Radi što manjeg utroška memorije dužina niza koeficijenata treba da se menja od polinoma do polinoma. Dužina niza je određena redom polinoma, tako da element niza s najvećim indeksom sigurno nije nula.

Program 4.9 predstavlja definiciju klase Poli za obradu polinoma.

Klasa ima dva polja. Red polinoma n i pokazivač na niz koeficijenata a od n+1 elemenata. Pošto je polinom nultog reda, u stvari, konstanta čija je vrednost jednaka a[0]. usvojeno je da se „prazan“ polinom, bez ijednog koeficijenta, obeležava sa n==−1. Sami koeficijenti se smeštaju u dinamičku zonu memorije. Prostor za njih se zauzima u momentu dodeljivanja vrednosti, kada se zna kolika je potrebna veličina tog prostora.

Pošto se očekuje da će veći broj metoda imati za parametre upućivače na nepromenljiv objekat tipa Poli naredbom **typedef** uveden je identifikator CP kao skraćenica za tip „nepromenljiv polinom“ (**const Poli**).

Kod klase koje za smeštanje svojih primeraka koriste dinamičko dodeljivanje memorije korisno je da postoje privatne metode za kopiranje objekta posmatrane klase u tekući objekat (predstavljen skrivenim argumentom metode **this**) i za pražnjenje sadržaja objekta (pretvaranje u ispravan „prazan“ objekat). Te metode će se pozivati iz više javnih metoda

Definicija klase polinoma (Poli).

```
#include <iostream>
using namespace std;

class Poli {
    int n; double* a; // Red polinoma i pokazivač na koeficijente.
    typedef const Poli CP; // Nepromenljiv polinom.
public:
    void kopiraj (const double* a, int n); // Kopiranje u tekući objekat.
    void brisi () { delete[] a; a=0; n=-1; } // Pražnjenje polinoma.
    Poli () { a = 0; n = -1; } // Konstruktor: - prazan polinom
    Poli (const double* a, int n) { kopiraj (a, n); } // - od niza
    Poli (CP& p) { kopiraj (p.a, p.n); } // - od polinoma
    ~Poli () { brisi (); } // Destruktor.
    void operator= (CP& p) { // Dodela vrednosti.
        if (this != &p) { brisi(); kopiraj(p.a, p.n); }
        return *this;
    }

    void niz (double* a, int n) const; // Smeštanje polinoma u niz.
    int operator+ () const { return n; } // Red polinoma.
    double operator() (double x) const; // Vrednost polinoma.
    double& operator[] (int ind); // Dohvatjanje koeficijenta.
    const double& operator[] (int ind) const;
    friend Poli operator+ (CP& p1, CP& p2); // p1 + p2
    friend Poli operator- (CP& p1, CP& p2); // p1 - p2
    friend Poli operator* (CP& p1, CP& p2); // p1 * p2
    Poli& operator+= (CP& p2) { return *this = *this + p2; } // p1 += p2
    Poli& operator-= (CP& p2) { return *this = *this - p2; } // p1 -= p2
    Poli& operator*= (CP& p2) { return *this = *this * p2; } // p1 *= p2
    friend istream& operator>> (istream& ut, Poli& p); // Čitanje polinoma.
    friend ostream& operator<< (ostream& it, CP& p); // Pisanje polinoma.
private:
    enum Greska { G_RED, G_IND };
    void greska (Greska g) const; // Šifre grešaka.
};
```

Program 4.9 – Definicija klase polinoma (poli1.h)

posmatranom slučaju to su metode `kopiraj()` i `brisi()`. Metoda `brisi()` je dovoljno jednostavna da može biti ugradena metoda. Za metodu `kopiraj()` treba primetiti da niz parametar nije tipa `Poli&`, kako je gore nagovušeno, već niz koeficijenata i red polinoma, ali to se svodi otprilike na isto. Metoda je na ovaj način mogla bolje da se iskoristi.

U javnom delu klase na prvom mestu su obavezni elementi klase koje za smeštanje držaju svojih primeraka koriste dinamičku zonu memorije: konstruktor, destruktur i operatorska funkcija za preklapanje operatara za dodelu vrednosti `=`.

Podrazumevani konstruktor samo postavlja vrednosti polja na prihvatljive i bezbedne vrednosti. Naročito je važno da se pokazivač a postavi na nulu da destruktur kasnije ne bi pravio štetu ako se u stvoreni objekat nikada ne stavi nijedan polinom.

Drugi konstruktor služi za inicijalizaciju novog objekta običnim nizom koeficijenata i redom polinoma, za slučaj da su ti podaci nastali izvan klase. Treći konstruktor je kons-

4.11.2 Obrada polinoma

truktur kopije. Oba konstruktora se svode na pozivanje gore pomenute privatne metode `kopiraj()`.

Destruktor prosto poziva privatnu metodu `brisi()`.

Dodela vrednosti (metoda `operator=()`) je jedna od suptilnijih radnji kod klase koje koriste memoriju u dinamičkoj zoni. Ako su izvorišni i odredišni objekti isti objekti, što se prepoznaće po tome da imaju iste adrese (`this==&p`), nema šta da se radi. Inače, prvo treba uništiti stari sadržaj odredišnog objekta (`brisi()`) i posle toga izgraditi novi sadržaj od kopije desnog operanda (`kopiraj(p.a, p.n)`). Vrednost metode je izmenjeni tekući objekat. Korišćenjem privatnih metoda `brisi()` i `kopiraj()` sadržaj operatorske funkcije `operator=()` postao je dovoljno jednostavan da ima smisla ugraditi ga u kôd. Ovde prikazano rešenje može da se shvati kao opšti oblik preklapanja operatara `=`. Od slučaja do slučaja menjaće se samo sadržaj pozivanih pomoćnih privatnih metoda.

U drugoj grupi javnih metoda nalaze se operacije za dobijanje podataka o polinomu u tekućem objektu. Nijedna od njih ne menja vrednost (stanje) tekućeg objekta, što je označeno modifikatorom `const` iz zatvorene oble zagrade na kraju niza parametara.

Metoda `niz()` dohvata niz koeficijenata i red polinoma, objekta tipa `Poli`. To omogućava da se, po potrebi, polinom obrađuje van klase `Poli`. Naravno, to je onda sve bez ikakve zaštite i nametanja ispravnog korišćenja tih podataka.

Unarni operator `+ u uobičajenoj aritmetici ne radi ništa`. Ovde je usvojeno da rezultat primene tog operatara na objekat tipa `Poli` daje red polinoma sadržanog u tom objektu. To je, naravno, postignuto preklapanjem operatora odgovarajućom operatorskom funkcijom.

Izračunavanje vrednosti polinoma za neku vrednost nezavisno promenljive ostvareno je preklapanjem operatora za pozivanje funkcija `()`. To omogućava uobičajenu notaciju za pisanje izraza za izračunavanje vrednosti polinoma `p(x)`, gde objekat `p` tipa `Poli` sadrži polinom čija se vrednost traži u tački `x` koja treba da je tipa `double`.

U sledećoj grupi metoda pristup nekom koeficijentu polinoma omogućen je preklapanjem operatora za indeksiranje (`operator[]()`). Pošto je vrednost te metode upućivač na koeficijent polinoma, koji je deo objekta na koji se primenjuje metoda, treba voditi računa da se spreči menjanje dohvaćenog koeficijenta, ako je to koeficijent nepromenljivog polinoma. Zbog toga postoje dve varijante metode za preklapanje operatora `[]`. U prvoj varijanti nije rečeno da metoda ne menja vrednost tekućeg objekta (bez obzira što ne menja taj objekat) i vraća se upućivač na promenljiv podatak (`double&`). U drugoj varijanti rečeno je da metoda ne menja vrednost tekućeg objekta i vraća se upućivač na nepromenljiv podatak (`const double&`). U slučaju promenljivog objekta `p` (tip `Poli`), izrazom `p[i]` pozivaće se prva varijanta operatorske funkcije za indeksiranje. Dobijeni rezultat će moći da se koristi u izrazu oblike `p[i]=x`. Time će vrednost polinoma da se promeni (menjanjem vrednosti odabranog koeficijenta) i tu promenu je posredno omogućila baš pozivana operatorska funkcija. U slučaju nepromenljivog objekta `p` (tip `const Poli`), izrazom `p[i]` pozivaće se druga varijanta operatorske funkcije za indeksiranje. Pošto se kao rezultat dobija upućivač na nepromenljiv podatak, prevodilac će da prijaví grešku za izraze oblike `p[i]=x`.

U nastavku programa 4.9 od aritmetičkih operacija s polinomima predviđeno je sabiranje, oduzimanje i množenje polinoma. Ostvarene su, prirodno, preklapanjem binarnih operatara `+`, `-` i `*` i to kao prijateljske funkcije, pošto ne stvaraju bočne efekte. Njihove vrednosti su privremeni objekti tipa `Poli`.

Varijante aritmetičkih operatora s bočnim efektima, analogno standardnim tipovima podataka, ostvarene su preklapanjem operatora `+=`, `==` i `*=` metodama koje kao rezultat vraćaju upućivač (`Poli&`) na promjenjeni tekući objekat. Za njih su iskorišćene operatorske funkcije za binarne operacije bez bočnih efekata i operatorska funkcija za dodelu vrednosti. Na primer, za operator `+=` prvo se sabiju sadržaj tekućeg objekta i desnog operanda (`*this+p2`) pozivanjem prijateljske funkcije `operator+()` i dobijeni privremeni objekat (`*this=...`) pozivanjem metode `operator=()`. se dodeljuje tekućem objektu (`*this=...`) pozivanjem metode `operator=()`.

Dobro opremljena klasa, obično, sadrži i operatore za ulaz i izlaz podataka. Zadatak operatora za ulaz (`>>`) je da, uz primenu odgovarajućih ulaznih konverzija, pročita sve informacije i od njih obrazuje objekat u ispravnom stanju. Zadatak operatora za potrebne informacije i od njih obrazuje objekat u datom objektu, primenom izlaznih konverzija, izlaz (`<<`) je da informacije sadržane u datom objektu, primenom izlaznih konverzija, prikazuje na neki pogodan način. U konkretnom slučaju radi se o redu polinoma i koeficijentima polinoma.

U toku obrade polinoma mogu da se javi neke greške. Za njihovu obradu u drugom privatnom delu, na samom kraju definicije klase `Poli`, predviđena je metoda `greska()`. Radi veće čitljivosti teksta programa mogućim greškama dodeljene su šifre u obliku simboličkih konstanti. Moguće greške su: nedozvoljeni red polinoma (`G_RED`) i nedozvoljeni indeks za pristup koeficijentu polinoma (`G_IND`). Te simboličke konstante predstavljaju moguće vrednosti za nabrojani tip `Greska`. To je ujedno i tip parametra metode `greska()`.

Definicije pravih funkcija (koje nisu ugradene funkcije), kako metoda tako i prijateljskih funkcija, nalaze se u zasebnoj datoteci `poli.C`, a prikazane su programima 4.10 i 4.11.

U privatnoj metodi `greska()` nalazi se niz tekstova poruka (`poruke`) koji se ispisuju na glavnom izlazu računara (`cout`). Sama poruka se bira na osnovu vrednosti parametra `g` indeksiranjem. Važno je uočiti da se posle ispisivanja poruke, program prekida pozivanjem globalne funkcije `exit()`. Dakle, iz metode `greska()` nikad se ne vraća na mesto odakle je pozvana. Kao završni status programa uzima se vrednost parametra posmatrane metode uvećana za jedan, da završni status ne bude jednak nuli (nula kao završni status smatra se za uspešan završetak programa, što ovde nije slučaj).

Privatna metoda `kopiraj()`, pošto se koristi i za kopiranje podataka koji potiču iz dečova programa izvan klase, prvo proverava ispravnost reda polinoma `nn`, uz istovremeno smeštanje te vrednosti u polje `n`. Ako je red neispravan, pozivanjem metode `greska()`, prekida se program. Zbog toga nastavak metode `kopiraj` ne mora da se stavlja u deo `else` prve naredbe `if`. U nastavku, prvo se dodeli potreban prostor za koeficijente polinoma i posle se prekopira sadržaj niza na koji pokazuje parametar `aa` u niz na koji pokazuje polje `a`.

Kod uzimanja elemenata polinoma (metoda `niz()`) nema nikakvih problema, bar što se tiče klase `Poli`. Dužnost je korisnika klase da pokazivač `a` pokazuje na dovoljno veliki memorijski prostor za prihvatanje svih koeficijenata polinoma. Po potrebi u to može da se uveri pre pozivanja funkcije `niz()` posmatranjem vrednosti izraza `+p` (red polinoma `p`).

Kod izračunavanja vrednosti polinoma (metoda `operator()()`), ne može da bude nikakvih problema, jer osim parametra `x` (koji ne može da ima nedozvoljenu vrednost) sve ostale informacije uzimaju se iz tekućeg objekta, a one su proverene prilikom stvaranja

4.11.2 Obrada polinoma

// Definicije metoda i prijateljskih funkcija uz klasu Poli.

```
#include "poli1.h"
#include <cstdlib>
using namespace std;

void Poli::greska (Greska g) const { // Obrada greške.
    const char* poruke[] = { "Nedozvoljen red polinoma",
        "Nedozvoljen indeks koeficijenta polinoma" };
    cout << "*** " << poruke[g] << "! ***\n";
    exit (g+1);
}

void Poli::kopiraj (const double* aa, int nn) // Kopiranje u tekuci obj.
{
    if ((n = nn) < -1) greska (G_RED);
    a = new double [nn+1];
    for (int i=0; i<=nn; i++) a[i] = aa[i];
}

void Poli::niz (double* aa, int& nn) const // Smeštanje polinoma u niz.
{
    for (int i=0; i<=n; i++) aa[i] = a[i]; nn = n;
}

double Poli::operator() (double x) const // Vrednost polinoma.
{
    double s = 0; for (int i=n; i>=0; (s*=x)+=a[i--]); return s;
}

double& Poli::operator[] (int ind) // Dohvatanje koeficijenta.
{
    if (ind < 0 || ind > n) greska (G_IND);
    return a[ind];
}

const double& Poli::operator[] (int ind) const
{
    if (ind < 0 || ind > n) greska (G_IND);
    return a[ind];
}

Poli operator+ (const Poli& p1, const Poli& p2) // Zbir dva polinoma.
{
    int n;
    if (p1.n == p2.n) for (n=p1.n; n>=0 && p1.a[n]+p2.a[n]==0; n--);
    else n = (p1.n > p2.n) ? p1.n : p2.n;
    Poli p; p.n = n; p.a = new double [n+1];
    for (int i=0; i<=n; i++)
        p.a[i] = (i > p2.n) ? p1.a[i] :
            (i > p1.n) ? p2.a[i] :
                p1.a[i] + p2.a[i];
    return p;
}
```

Program 4.10 – Definicije funkcija uz klasu `Poli` (`poli1.C`, prvi deo)

objekta. Posebno se skreće pažnja na treći izraz u naredbi `for`. Pošto je u jeziku C++ rezultat operatora `*=` (tačnije, svih operatora za dodelu vrednosti) `lvrednost`, vrednost izraza `s*=x` je promenljiva s izmenjenog sadržaja. Posle na to može da se primeni operator `+=` radi dodavanja vrednosti sledećeg koeficijenta polinoma `a[i]`. Na jeziku C isti izraz morao bi da se piše kao `s=s*x+a[i--]`. Oble zgrade u izrazu `(s*=x)+= a[i--]` su neophodne, jer se operatori za dodelu grupišu zdesna uлево. U odsustvu zgrade izraz bi se

iačio kao $s^* = (x += a[i--])$, a to je nešto sasvim drugo (a promenila bi se i vrednost menljive $x!$).

Obe metode za pristup određenom koeficijentu polinoma (**operator[]()**) za promeni za nepromenljiv tekući objekat), prvo proveravaju da li je indeks i unutar dozvoljenih granica, i ako jeste vraćaju odgovarajući element iz niza koeficijenata ($a[i]$) kao menljivu ili kao nepromenljvu *lvrednost* (ne kopiju sadržaja tog elementa već sam ment!). I u ovim metodama otkrivanje greške dovešće do prekidanja celog programa.

Prilikom sabiranja polinoma (prijateljska funkcija **operator+()**) prvo treba odrediti rezultujućeg polinoma n . U slučaju kada su sabirci polinomi istih redova, rezultujući linom može da bude i nižeg reda, ako su vodeći koeficijenti suprotnog predznaka i iste solutne vrednosti. Inače red rezultujućeg polinoma je viši red među redovima sabiraka. privremeno smeštanje rezultata definisan je lokaln polinom p . Posle dodele memorije smeštanje rezultujućih koeficijenata mogu da se izračunaju ti koeficijenti. Pošto je red funkcije tipa **Poli**, a ne **Poli&** (*lvrednost*), kopija lokalnog objekta p , stvorena konstruktorom kopije, vratiće se pozivajućem programu, a p će da se uništi destruktorm u moment povratka iz funkcije **operator+()**.

Oduzimanje polinoma (funkcija **operator-()** u programu 4.11) je potpuno analogno biranju, a i množenje (funkcija **operator*()**) je, principijelno gledano, vrlo slično njemu. Čak je i jednostavnije po pitanju utvrđivanja reda rezultujućeg polinoma. Ako je jedan faktora prazan polinom, rezultat je prazan, inače red rezultata je jednak zbiru redova kторa.

Skreće se još pažnja na to da pri navođenju tipa parametara funkcija za aritmetičke operacije (**const Poli&**) nije mogao da se koristi identifikator **CP** (u obliku **CP&**), koji je u asi **Poli** uveden kao skraćenica upravo za tu svrhu, jer se naredba **typedef** nalazi u privatnom delu klase **Poli**. Mada se radi o prijateljskim funkcijama, privatni identifikatori nase mogu da se koriste samo unutar tela funkcije, a ne i u zaglavlju funkcije. Da je nadeba **typedef** smeštena u javni deo klase, tip parametara posmatranih funkcija mogao bi i se naznači kao **Poli::CP&**.

Funkcija za čitanje polinoma uz primenu ulaznih konverzija (**operator>>()**), prvo ništi stari sadržaj polinoma pozivanjem metode **brisi()**. Pošto se radi o prijateljskoj funkciji, za njeno pozivanje mora da se navede neki objekat tipa **Poli** (**p.brisi()**), samo ime metode nije dovoljno. Slično tome, ako se pročita nedozvoljena vrednost za red polinoma (<-1), za pozivanje metode **greska()** treba da se piše **p.grseka(...)**. Ni argument te metode ne može da se piše samo kao **G_RED**. Mora da se naznači da taj identifikator pripada klasi **Poli**. Mada bi moglo da se piše **p.G_RED**, prirodnije je da se piše **Poli::G_RED**, jer su simboličke konstante uvedene naredbom **enum** u definiciji klase karakteristike klase celini a ne pojedinačnih primeraka klase. Slično važi i za identifikatore uvedene naredbama **typedef**, bez obzira što se ne koristi modifikator **static** za eksplisitno označavanje da se radi o zajedničkim elementima klase.

Na kraju, posle dodele memorije potrebne veličine u dinamičkoj zoni, operatorska funkcija za operator **>>** čita koeficijente polinoma i smešta ih u upravo dodeljenu memoriju. Koeficijenti se čitaju po opadajućim vrednostima stepena nezavisno promenljive (poslednji je čita slobodan član). Da bi promene vrednosti parametra **p** mogle da se prenose u pozivajući program, on je upućivač na objekte tipa **Poli**. Sama vrednost funkcije mora da bude upućivač na ulazni tok (koji je istovremeno i prvi parametar funkcije) odakle se podaci čitaju. Vrednost tog parametra (ulaznog toka) ne sme eksplisitno da se menja unutar

4.11.2 Obrada polinoma

```

Poli operator- (const Poli& pl, const Poli& p2) // Razlika dva polinoma.
    int n;
    if (pl.n == p2.n) for (n=pl.n; n>=0 && pl.a[n]-p2.a[n]==0; n--);
    else n = (pl.n > p2.n) ? pl.n : p2.n;
    Poli p; p.n = n; p.a = new double [n+1];
    for (int i=0; i<=n; i++)
        p.a[i] = (i > p2.n) ? pl.a[i]
                               (i > pl.n) ? -p2.a[i] :
                                         pl.a[i] - p2.a[i];
    return p;
}

Poli operator* (const Poli& pl, const Poli& p2) // Proizvod dva polinoma.
    int n = (pl.n>=0 && p2.n>=0) ? pl.n+p2.n : -1;
    Poli p; p.n = n; p.a = new double [n+1];
    for (int i=0; i<=n; p.a[i++]=0);
    for (int i=0; i<=pl.n; i++)
        for (int j=0; j<=p2.n; j++)
            p.a[i+j] += pl.a[i] * p2.a[j];
    return p;
}

istream& operator>> (istream& ut, Poli& p) // Čitanje polinoma.
    p.brisi ();
    ut >> p.n; if (p.n < -1) p.greska (Poli::G_RED);
    p.a = new double [p.n+1];
    for (int i=p.n; i>=0; ut>>p.a[i--]);
    return ut;
}

ostream& operator<< (ostream& it, const Poli& p) // Pisanje polinoma.
    it << "p[";
    for (int i=p.n; i>=0; i--) it << p.a[i] << (i ? "," : "");
    it << ']';
    return it;
}

```

Program 4.11 – Definicije funkcija u klasu **Poli** (poli1.C, drugi deo)

programerovih funkcija. Do implicitnih izmena vrednosti može doći u toku primene operatora **>>** na podatke standardnih tipova (**int, double** itd.).

Ispisivanje polinoma (funkcija **operator<<()**) predviđeno je u obliku niza vrednosti koeficijenata unutar uglastih zagrada uz ispisivanje slova **p** na početku, na primer **p[3,-2,5]**. Koeficijenti se prikazuju po opadajućem redosledu vrednosti stepena nezavisne promenljive. Navedeni primer, dakle, predstavlja $3x^2 - 2x + 5$.

Program 4.12 prikazuje neke mogućnosti klase **Poli**.

Na početku se definisu tri primerka klase **Poli** i inicijalizuju se da predstavljaju prazne polinome. U nastavku se pročitaju s glavnog ulaz računara dva polinoma da bi mogle da se isprobaju aritmetičke operacije i dodela vrednosti. Bez obzira na složenost objekata tipa **Poli**, identifikatori primeraka te klase mogu da se koriste kao i identifikatori bilo koje promenljive standardnog prostog tipa.

Sledeći blok naredbi prikazuje konverziju polinoma u običan niz koeficijenata i obrnuto, stvaranje polinoma od dobijenog niza. Sam niz koeficijenata smešta se u dinamičku zonu memorije uz pomoć pokazivača **a**. Veličina potrebne memorije određuje se

```
// Program za ispitivanje klase Poli.

#include "poli.h"
#include <iostream>
using namespace std;

int main () {
    // Definisanje tri polinoma i inicijalizacija kao prazni polinomi.
    Poli p1, p2, p3;

    // Čitanje polinoma, aritmetičke operacije i dodela vrednosti.
    cin >> p1; cout << "p1 = " << p1 << endl;
    cin >> p2; cout << "p2 = " << p2 << endl;
    p3 = p1 + p2; cout << "p1+p2 = " << p3 << endl;
    p3 = p1 - p2; cout << "p1-p2 = " << p3 << endl;
    p3 = p2 - p1; cout << "p2-p1 = " << p3 << endl;
    p3 = p1 - p1; cout << "p1-p1 = " << p3 << endl;
    p3 = p1 * p2; cout << "p1*p2 = " << p3 << endl;

    // Konverzija polinoma u niz i obrnuto.
    double* a = new double [+p1 + 1]; int n; p1.niz (a, n);
    cout << "a = ";
    for (int i=n; i>=0; i--) cout << a[i] << ' '; cout << endl;
    p3 = Poli (a, n); cout << "p(a,n)= ";
    for (i=+p3; i>=0; i--) cout << p3[i] << ' '; cout << endl;
    delete [] a;

    // Korišćenje pokazivača na polinom.
    Poli* pp4 = new Poli (p1);
    cout << "*pp4 = " << *pp4 << endl;
    delete pp4;

    // Tabeliranje polinoma.
    double xmin, xmax, dx; cin >> xmin >> xmax >> dx;
    cout << "\nxmin, xmax, dx = "
        << xmin << ", " << xmax << ", " << dx << endl;
    cout << "\n x   p1(x)\n=====\n" << fixed;
    for (double x=xmin; x<=xmax; x+=dx)
        cout << setprecision(2) << setw(6) << x
            << setprecision(6) << setw(12) << p1 (x) << endl;
}

```

Program 4.12 – Primer korišćenja klase Poli (poli1t.C)

kao $+p1+1$, pošto je $+p1$ red polinoma $p1$. Polinom od niza koeficijenata stvara se eksplicitnim pozivanjem konstruktora ($Poli(a, n)$) i dobijeni rezultat se smešta u već postojeći polinom $p3$. Tom prilikom koristi se operatorska funkcija za dodelu vrednosti, a ne konstruktor kopije. Koeficijenti rezultujućeg polinoma $p3$, izuzetno, ispisuju se u ciklusu da bi se isprobala i operatorska funkcija za indeksiranje ($p3[i]$).

Naredni blok naredbi prikazuje korišćenje pokazivača na primerke klase Poli. Pokazivač $pp4$ inicijalizuje se rezultatom operatora `new` koji se koristi za dodeljivanje memorije u dinamičkoj zoni jednom primerku klase Poli. Sam objekat se inicijalizuje kopijom polinoma $p1$ uz automatsko pozivanje konstruktora kopije. Posle toga $*pp4$ je objekat tipa `Poli` i može da se koristi kao operand operatora `<<` radi ispisivanja na glavnom izlazu računara. Prilikom uništavanja dinamičkog objekta pomoću operatora `delete`, automatski

će se pozvati destruktor klase `Poli` koji će da oslobodi memorijski prostor u dinamičkoj zoni memorije u kome se nalaze koeficijenti polinoma.

U poslednjem delu programa 4.12 tabeliraju se vrednosti polinoma unutar datog intervala nezavisne promenljive. Ovo predstavlja primer korišćenja preklopjenog operatora za pozivanje funkcija `()`, u posmatranom slučaju to je izraz $p1(x)$. Za ubolicavanje tabele korišćeni su manipulatori kao specijalni operandi operatora `<<`. Još pri ispisivanju zaglavljena tabela je podešeno ispisivanje realnih brojeva bez eksponenta (fixed). Vrednosti promenljive x se ispisuju sa dve decimale (`setprecision(2)`) u ukupno 6 slovnih mesta (`setw(6)`). Pošto se koriste i manipulatori s parametrima na početku programa 4.12 nalazi se i direktiva preprocesora `#include <iomanip>`.

Rezultat 4.2 prikazuje primer rada programa 4.12.

```
% polit <polit.pod
p1      = p[1,2,3,4,5,6]
p2      = p[5,4,3,2,1]
p1+p2  = p[1,7,7,7,7,7]
p1-p2  = p[1,-3,-1,1,3,5]
p2-p1  = p[-1,3,1,-1,3,-5]
p1-p1  = p[]
p1*p2  = p[5,14,26,40,55,70,50,32,17,6]
a      = 1 2 3 4 5 6
p(a,n)= 1 2 3 4 5 6
*pp4   = p[1,2,3,4,5,6]

xmin, xmax, dx = -1, 1, 0.25

      x      p1(x)
=====
-1.00  3.000000
-0.75  3.629883
-0.50  4.218750
-0.25  4.959961
  0.00  6.000000
  0.25  7.555664
  0.50  10.031250
  0.75  14.135742
  1.00  21.000000
```

Rezultat 4.2 – Obrada polinoma programom 4.12

4.11.3 Obrada redova

Zadatak:

Napisati na jeziku C++ klasu za obradu redova neograničenih kapaciteta koji sadrže celobrojne podatke. Napisati na jeziku C++ program za prikazivanje mogućnosti te klase.

Rešenje:

Redovi su strukture podataka kod kojih se podaci koji pristižu uvek dodaju na kraj reda, a koji odlaze uzmimaju s početka reda. Koriste se za simulaciju redova, na primer ispred neke blagajne, s principom opsluživanja „prvi ušao, prvi izašao“ (*first in, first out – FIFO*).

Redovi mogu da budu ograničenih i neograničenih kapaciteta. Redovi teorijski ne- ničenih kapaciteta mogu da se ostvaruju u obliku jednostruko spregnutih listi, čiji se elementi smeštaju u dinamičku zonu memorije. Pošto se elementi dodaju na jednom, a uključuju u drugog kraja liste, za efikasnu obradu potrebna su dva pokazivača. Jedan pokazuje početak i jedan na kraj liste.

Program 4.13 prikazuje definiciju klase `Red` za obradu redova celobrojnih podataka.

Definicija klase redova (Red).

```
class Red {
    struct Eleml {
        int pod; // Elementi reda;
        Eleml* sled; // sadržaj elementa,
        Eleml* sred; // sledeći element;
    } Eleml (int p, Eleml* s=0) // konstruktor.
    { pod = p; sred = s; }

    Eleml *prvi, *posl; // Početak i kraj reda.
    int duz; // Dužina reda.
    void kopiraj (const Red& r); // Kopiranje reda.
    void brisi (); // Brisanje svih elemenata.
public:
    Red () { prvi = posl = 0; duz = 0; } // Konstruktor praznog reda.
    Red (int a) // Konverzija int u Red.
    { prvi = posl = new Eleml (a); duz = 1; }
    Red (const Red& r) { kopiraj (r); } // Konstruktor kopije.
    ~Red () { brisi (); } // Destruktor.
    Red& operator= (const Red& r) { // Dodela vrednosti.
        if (this != &r) { brisi (); kopiraj (r); }
        return *this;
    }

    int operator+ () const { return duz; } // Dužina reda.
    Red& operator+= (const Red& r); // Dodavanje reda.
    Red operator+ (const Red& r) const; // Spoj dva reda.
    int operator-- (); // Brisanje prvog elementa.
    int operator- () const // Prvi element.
    { return duz ? prvi->pod : 0; }
    Red operator- (int k) const; // Red bez elemenata jednakih k.
    Red& operator-= (int k) // Brisanje svih elemenata jednakih k.
    { return *this = *this - k; }
    Red& operator~ () // Brisanje svih elemenata.
    { brisi (); return *this; }
};
```

Program 4.13 – Definicija klase redova (red.h)

Elementi liste su strukture s dva polja: celobrojan koristan sadržaj elementa i pokazivač sledeći element liste. Struktura je uklopljena u klasu `Red`, pa je doseg njenog identifikatora upravo ta klasa. Članovi strukture su javni, ali pošto se struktura vidi samo unutar klase `Red`, niko izvan klase ne može da im pristupa.

Svaki primerak klase `Red` je jedan potpun red. Polja su privatna i obuhvataju pokazivače na prvi i poslednji element reda (liste) i dužinu reda (broj elemenata u listi). Nije neopodno da informacija o dužini reda bude zasebno polje klase. Do nje uvek može da se dode rebrojavanjem elemenata liste. Prolazak kroz celu listu, međutim, dosta dug je.

4.11.3 Obrada redova

odlučeno da se taj podatak drži stalno u pripravnosti. Naravno, svako dodavanje ili brisanje elemenata, na ovaj način, iziskuje podešavanje trenutne dužine reda. To podešavanje traje vrlo kratko.

U privatnom delu klase `Red` nalaze se još dve pomoćne metode: `kopiraj()` i `brisi()`. Prva obrazuje duplikat svog pravog parametra u tekućem objektu (čija se adresa nalazi u skrivenom parametru). Druga metoda uništava sve elemente reda u tekućem objektu. Neke od javnih metoda pozivaju ove dve metode, a uvedene su da se njihovi sadržaji ne bi pisali više puta.

U javnom delu klase `Red` na prvom mestu nalaze se neizbežni konstruktori i destruktori.

Podrazumevani konstruktor stvara prazan red od nula elemenata. Konstruktor konverzije s jednim parametrom tipa `int` stvara red od jednog elementa. Zbog postojanja konstruktora u strukturi `Eleml`, njegov sadržaj je vrlo jednostavan. Konstruktor kopije poziva privatnu metodu `kopiraj()` za inicijalizaciju tekućeg objekta kopijom pravog parametra. Svi konstruktori su dovoljno jednostavnvi da budu ugrađene metode.

Na kraju, destruktorni uništava sve elemente reda u odgovarajućim trenucima. Sastoje se samo od pozivanja pomoćne metode `brisi()`, pa je i za njega predviđeno da bude ugrađena metoda.

Preostale radnje nad redom su ostvarene preklapanjem operatora. Pokušalo se da tumačenja operatora budu što prirodnija. To nije uspeo najbolje za svaku radnju. U takvim slučajevima upotreba običnih metoda bila bi bolja, jer bi korisnik klase mogao lakše da shvati koja radnja šta znači.

Zbog svog značaja, na prvom mestu je operator za dodelu vrednosti (`=`). Kao što je već u prethodnom zadatku pokazano, uz postojanje pomoćnih privatnih metoda `brisi()` i `kopiraj()` metoda kojom se operator `=` preklapa ima vrlo karakterističan sadržaj, nezavisno od vrste obradivanih podataka. Pored toga, dovoljno je jednostavna da može biti ugrađena metoda. Skreće se još pažnja na to da, zbog postojanja konstruktora konverzije iz tipa `int` u tip `Red`, može da se piše izraz oblika `r=k`, gde je `r` promenljiva tipa `Red`, a `k` celobrojan podatak. Novi sadržaj reda `r` biće lista čiji jedini element sadrži broj `k`.

Unarni operator `+`, koji za standardne podatke nema nikakvog efekta, iskoriscen je za dohvatanje dužine reda. Kao i u prethodnom zadatku, i ovde daje informaciju o „veličini“ sadržaja svog operanda.

Operator `+=` dodaje ceo red s desne strane na kraj reda s leve strane. U slučaju izraza `r+=k`, ceo broj `k` dodaće se na kraj reda s leve strane. Razume se, opet uz primenu konverzije celog broja u red. Ovim operatorom može da se započinje čak i obrazovanje reda polazeći od praznog reda.

Binarni operator `+` radi isto što i `+=`, jedino što ne menja vrednost svog levog operanda, tj. ne stvara bočni efekat. Rezultat se dobija u vidu novog privremenog objekta. Izraz `r+k` je i sada dozvoljen s već objašnjениm rezultatom. Ali izraz `k+r`, koji je za standardne tipove dozvoljen, ovde nije dozvoljen zato što je funkcija `operator+()` član klase. Levi operand je tekući objekat na koji se ne primenjuje automatska konverzija tipa. Za automatsku konverziju tipa levog operanda funkcija `operator+()` treba da je prijateljska funkcija. Ovako u slučaju potrebe mora da se piše `Red(k)+r`. Uzgred, ovim izrazom se novi podatak dodaje na početak reda, što nije baš fair, pa i nije velika šteta što to ne može jednostavno da se uradi.

Interesantno je uočiti da uz navedena preklapanja operatora `=`, `+ i += izrazi r+=a i r=r+a imaju isti efekat, pri čemu je svejedno da li je a tipa int ili Red. Razlika može da`

bude samo u efikasnosti. Drugi izraz koristi dve funkcije, pa zato možda traje duže. Međutim, može da se desi da metoda `opretator+=()` poziva metode `operator+()` i `operator=()`!

Prefiksni oblik operatora `--` je iskorišćen za uzimanje prvog elementa iz reda uz njegovo brisanje iz reda. Dakle, kao što je uobičajeno, operator `--` i sada stvara bočni efekat, „smanjujući” svoj operand za jediničnu veličinu u izvesnom smislu. Neuobičajeno je, što vrednost rezultata nije operand u izmenjenom stanju, već ceo broj koji je bio na početku reda.

Zbog sličnog izgleda operatora `- i --`, operator unarni `-` je iskorišćen za dohvatanje podatka koji je na početku reda, ali bez brisanja podatka iz reda. Dakle, kao što je uobičajeno, ovaj operator ne stvara bočni efekat. Inače, asocijacija između korišćenog operatora i radnje koja se izvodi prilično je slaba. Zbog jednostavnog sadržaja ostvarenje ovog operatora je u obliku ugrađene metode. Treba uočiti da u slučaju praznog reda unarni operator `-` daje nulu kao rezultat. To se smatralo najjednostavnijim „bezbednim” rešenjem. Neko će možda prigovarati ovom rešenju, smatrajući pokušaj dohvatanja prvog elementa iz praznog reda greškom koju treba kažnjavati prekidanjem programa! Na kraju krajeva, unarnim operatorom `+ uvek` može da se proveri ima li bilo čega u redu pre pokušaja dohvatanja prvog elementa.

Binarnom operatoru `-` dato je tumačenje da obrazuje red od onih elemenata reda s leve strane koji nisu jednaki celom broju s desne strane. Ne stvara bočni efekat, a rezultat je „umanjeni” levi operand. Ovde, očigledno, izraz `k-r` nema smisla, pa ne smeta što nije dozvoljen.

Sasvim je logično da operator `--` stvara bočni efekat i da stvarno izostavlja iz reda s leve strane sve elemente koji su jednaki celom broju s desne strane. Operatorska funkcija `operator-=()` je ostvarena pozivanjem operatorskih funkcija `operator-()` za binarni `-` i `operator=()`. Na ovakav način je dovoljno jednostavna da se ugradi neposredno u kôd.

Ovde prikazano rešenje za ostvarenje istovetne binarne operacije bez i s bočnim efektom je ono što se preporučuje: ostvariti binarni operator bez bočnog efekta, a operator s bočnim efektom upotrebom prvog operatora (naravno, i operatora za dodelu vrednosti). Ostvarenje operatora s bočnim efektom tada uvek ima ovde prikazani oblik, menja se samo simbol za operator. Rešenje u kome se prvo ostvaruje operator s bočnim efektom i pomoću njega operator bez bočnog efekta, ne deluje dovoljno prirodno, pa ga treba izbegavati. Ovo drugo rešenje je pokazano u nastavku, na primeru para operatora `i +=`.

Poslednje definisani operator u klasi `Red` je unarni operator `~`, logička negacija po bitovima u „običnom životu”. Iskorišćen je za izostavljanje svih elemenata reda. Ovaj operator je izabran jer se taj simboli koristi i u imenima destruktora koji rade sličnu stvar. Jedino što se destruktur koristi automatski u momentima kada dati objekat više nije potreban, i neće više postojati posle primene destruktora. Posle primene operatorske funkcije `operator~()` objekat i dalje postoji, jedino što je prazan. Nema nikakve prepreke da se kasnije u njega ponovo stave neki podaci. Pošto posmatrana operatorska funkcija samo poziva privatnu metodu `bris()`, predviđena je da bude ugrađena metoda.

Mada destruktori mogu i eksplisitno da se pozivaju i kao takvi mogli bi da se iskoriste za „pražnjenje” objekata, to treba izbegavati. Za tu svrhu bolje je napraviti posebnu metodu, a ulogu destruktora ograničiti samo za trenutke konačnog uništavanja objekata.

Ostvarenje gore opisanih metoda, a koje nisu definisane unutar definicije klase `Red`, prikazano je u programu 4.14.

```
// Definicije metoda uz klasu Red.

#include "red.h"

void Red::kopiraj (const Red& r) { // Kopiranje reda.
    prvi = posl = 0; duz = r.duz;
    for (Elem* tek=r.prvi; tek; tek=tek->sled)
        posl = (!prvi ? prvi : posl->sled) = new Elem (tek->pod);
}

void Red::bris () { // Brisanje svih elemenata reda.
    while (prvi) { Elem* stari=prvi; prvi = prvi->sled; delete stari; }
    posl = 0; duz = 0;
}

Red& Red::operator+= (const Red& r) { // Dodavanje reda.
    Red s (r);
    (!prvi ? prvi : posl->sled) = s.prvi;
    posl = s.posl; duz += s.duz;
    s.prvi = s.posl = 0; s.duz = 0;
    return *this;
}

Red Red::operator+ (const Red& r) const { // Spoj dva reda.
    // Ne može "inline" zbog vraćanja kopije lokalnog objekta!
    Red s (*this); s += r; return s;
}

int Red::operator-- () { // Brisanje prvog elementa iz reda.
    if (duz == 0) return 0;
    int pod = prvi->pod;
    Elem* stari = prvi;
    if ((prvi = prvi->sled) == 0) posl = 0;
    delete stari;
    duz--;
    return pod;
}

Red Red::operator- (int k) const { // Red bez elemenata jednakih k.
    Red r;
    for (Elem* tek=prvi; tek; tek=tek->sled) if (tek->pod != k) r+=tek->pod;
    return r;
}
```

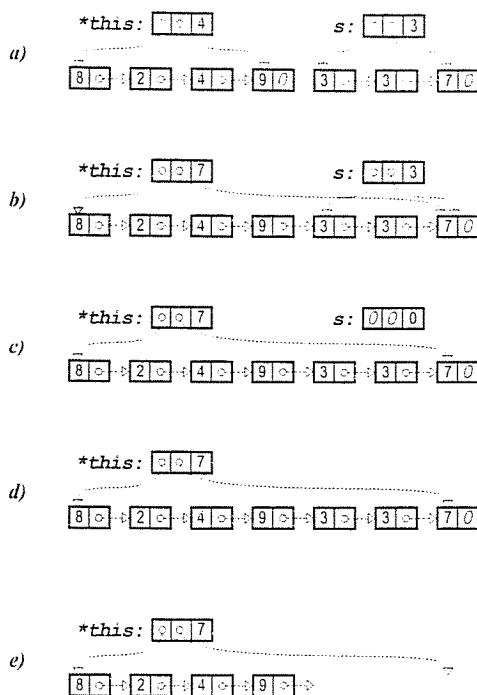
Program 4.14 – Definicije metoda uz klasu Red (red.C)

Metoda `kopiraj()` podrazumeva da tekući objekat nema nikakav koristan sadržaj. To omogućava njeno pozivanje iz konstruktora. Ako se poziva za već postojeći objekat, taj objekat mora prethodno da se isprazni. Metoda `kopiraj()` prvo postavi pokazivače na početak i kraj liste na nulu, što je potrebno ako se poziva iz konstruktora. Posle preuzimanja dužine reda `r`, uzima uzastopne elemente reda (liste) `r`. Za svaki od njih stvara novi element tipa `Elem` u dinamičkoj zoni inicirajući ga kopijom podatka iz tekućeg elementa originalne liste (`tek->pod`). Novostvoreni element se uključuje, jednom naredbom, na kraj liste koja čini sardžaj tekućeg objekta. Skreće se pažnja na to da se operator za dodelu vrednosti grupiše zdesna uлево. Zbog toga se prvo sadržaj pokazivača `=` stavila u

zivač prvi ili u pokazivač posl->sled, zavisno od toga da li je lista prazna ili ne. Toga se ista vrednost stavlja u pokazivač posl koji stalno pokazuje na poslednji element liste.

Izmetoda `bris()` je vrlo jednostavna. Prolazi duž reda u tekućem objektu i oslobođava oriju u dinamičkoj zoni, koja je ranije dodeljena elementima reda. Na kraju, tekući red je ispravan prazan objekat.

Ispitujmo sada učinkovitost operatora `operator+=()`. U skladu sa prethodnim razgovorom, u funkciji `operator+=()` se lokalni katalog s inicijalizuje kopijom parametra `x` koji predstavlja red koji treba dodati na kraj reda objekta `*this` (slika 4.1.a).



Slika 4.1 – Dodavanje reda na kraj drugog reda

Pošto je red s kopijama parametra funkcije, njegovi elementi mogu neposredno da se ljuče na kraj liste u tekućem objektu. To se postiže prepravljanjem svega dva pokazivača ika 4.1.b). Ti elementi sada pripadaju dvama objektima, `*this` i `s`. Pošto to smeta, objekat `s` se prepravi tako da postane prazan red (slika 4.1.c). Na taj način kada se na kraju mene, zbog napuštanja dosega identifikatora `s`, pozove destruktorni kod za uništavanje objekta `s`, ištiće se samo polja tog objekta (slika 4.1.d).

Da je situacija sa slike 4.1.b ostavljena do završetka metode, destruktor bi uništio i elemente koji su dostupni polazeći od polja objekta `s` (slika 4.1.e). Posle toga tekući objekt (`*this`) više ne bi bio ispravan objekat. Neki pokazivači bi pokazivali na elemente u namičkoj zoni memorije koji više ne postoje.

Operatorska funkcija za spajanje redova bez bočnog efekta, `operator+=()`, ostvarena je korišćenjem već opisanih metoda. To su konstruktor kopije za inicijalizaciju lokalnog objekta s i `operator+=()` za dopisivanje sadržaja parametra z na kraj reda s. Vrednost tekueg objekta ne podleže nikakvym izmenam. S obzirom na jednostavan sadržaj metode `operator+=()`, mogla bi biti ugrađena metoda. Međutim, prevodioци su odbili da to učine, jer je rezultat kopija lokalnog objekta. To je razlog više da se daje prednost već pomenu-tom rešenju: binarni operator s bočnim efektom ostvariti pomoću odgovarajućeg operatara bez bočnog efekta, a ne obrnuto.

Operatorska funkcija **operator--()**, slično unarnom operatoru `-`, za slučaj praznog reda daje nulu kao rezultat. Ako red nije prazan, upamti se sadržaj prvog elementa u listi, pre nego što se osloboди memorija u dinamičkoj zoni koja je bila dodeljena elementu. Treba voditi računa i o tome da se, ako se izostavlja poslednji element liste, pokazivač `posl` postavi na nulu.

Operatorska funkcija `operator-()` za obrazovanje reda koji ne sadrži elemente iz tekućeg reda koji su jednaki nekoj zadatoj vrednosti, intenzivno koristi usluge ostalih metoda klase `Red`. Lokalan objekat `r` inicijalizuje se pomoću podrazumevanog konstruktora. Celobrojni sadržaji (`int`) elemenata početnog reda, koji ulaze u rezultat operacije, pretvaraju se konstruktorom `Red(int)` u red od jednog elementa, da bi se posle dodali na kraj lokalnog reda `r` pomoću operatorske funkcije `operator+=()`.

Program 4.15 predstavlja primer za prikazivanje nekih mogućnosti klase `Random`.

Pomoćna funkcija `citaj()` obrazuje red od niza brojeva koje čita s glavnog ulaza računara. Predviđena je za čitanje unapred pripremljenih podataka iz neke datoteke, skretanjem glavnog ulaza. Zbog toga pročitane brojeve i ispisuje na glavnom izlazu. Ispred niza brojeva, kao neku vrstu naslova, ispisuje tekst koji se unosi kao parametar funkcije pomoću pokazivača `nasl`. Rezultat se obrazuje u lokalnom objektu `r` tipa `Red`. Objekat se implimenticno inicijalizuje podrazumevanim konstruktorom, kao prazan red. Pročitani brojevi dodaju se na kraj tog reda pomoću operatora `+=`. Vrednost funkcije je kopija lokalnog objekta `r`.

Pomoćna funkcija `pisi()` ispisuje sadržaj reda `red` uz ispisivanje uvodnog teksta `nasli.` Prvo se u lokalnom objektu `r` napravi kopija parametra `red`. Posle toga u toku svakog prolaza kroz ciklus ispisuje se element na početku reda, koji se tom prilikom briše iz reda `(--r)`. Ciklus se nastavlja sve dok je dužina reda `(+r)` različita od nule. Ovo nije efikasno rešenje, ali štedi programerski trud.

Glavna funkcija (funkcija `main()`) sadrži primere korišćenja svih operacija predviđenih u klasi `Red`. Rezultat 4.3 prikazuje primer rada programa 4.15.

```
// Program za ispitivanje klase Red.

#include "red.h"
#include <iostream>
using namespace std;

Red citaj (const char* nasl) {
    Red r; int n, a, i; cin >> n;
    cout << nasl << " :";
    for (i=0; i<n; i++) {cin >> a; cout << ' ' << a; r += a;} cout << endl;
    return r;
}

void pisi (const char* nasl, const Red& red) {
    Red r = red;
    cout << nasl << " :"; while (+r) cout << ' ' << --r; cout << endl;
}

int main () {
    Red r1 = citaj ("niz1 ");
    pisi ("r1 ", r1 );
    Red r2 = citaj ("niz2 ");
    pisi ("r2 ", r2 );
    pisi ("r1+r2 ", r1+r2);
    r1 += r2; pisi ("r1+=r2", r1 );
    pisi ("~r1 ", ~r1 );
    --r1; pisi ("--r1 ", r1 );
    int a; cin >> a;
    cout << "a : " << a << endl;
    pisi ("r1-a ", r1-a );
    r1 -= a; pisi ("r1-=a ", r1 );
    ~r1; pisi ("~r1 ", r1 );
}
}
```

Program 4.15 – Primer korišćenja klase Red (redt.C)

```
% redt <redt.pod
niz1 : 2 4 1 3 2 5
r1 : 2 4 1 3 2 5
niz2 : 1 2 2 3 2
r2 : 1 2 2 3 2
r1+r2 : 2 4 1 3 2 5 1 2 2 3 2
r1+=r2 : 2 4 1 3 2 5 1 2 2 3 2
~r1 : 2
--r1 : 4 1 3 2 5 1 2 2 3 2
a : 2
r1-a : 4 1 3 5 1 3
r1-=a : 4 1 3 5 1 3
~r1 :
```

Rezultat 4.3 – Obrada redova programom 4.15

5 Izvedene klase

Objekti u svakodnevnom životu mogu da se grupišu na osnovu njihovih zajedničkih osobina. Unutar grupe moguća su dalja grupisanja na osnovu nekih specifičnijih zajedničkih osobina. Svi objekti date podgrupe poseduju sve osobine svoje grupe kao i još neke specifične osobine. Na primer:

- geometrijske figure u ravni mogu da budu okarakterisane položajem, orientacijom, površinom i obimom,
 - krugovi su geometrijske figure u ravni koje su dodatno okarakterisane poluprečnikom,
 - kvadrati su geometrijske figure u ravni koje su dodatno okarakterisane dužinom ivice,
 - trouglovi su geometrijske figure u ravni koje su dodatno okarakterisane dužinama stranica,
 - ...;
- vozila prevoze stvari,
 - letelice su vozila koja lete,
 - avioni su letelice,
 - helikopteri su letelice,
 - rakete su letelice,
 - vaskonski brodovi su letelice,
 - plovila su vozila koja plove po vodi,
 - čamci su plovila,
 - jedrenjaci su plovila,
 - brodovi su plovila,
 - suvozemna vozila su vozila koja se kreću po zemlji,
 - bicikli su suvozemna vozila sa dva točka,
 - ...;
 - ...;

U jeziku C++ grupe objekata opisuju se pomoću klasa. Klase koje opisuju podgrupu objekata s nekim dodatnim, specifičnim osobinama u odnosu na opštije grupe nazivaju se

tene klase. Polazne klase, koje opisuju opšiju grupu objekata, nazivaju se **osnovne** za date izvedene klase.

U literaturi o objektno orijentisanom programiranju osnovne klase nazivaju se i natklase roditelji, klase preci, a izvedene klase potklase, klase deca ili klase potomci. Sâmo članje naziva se i proširivanje osnovne klase dodatnim osobinama. Klase se mogu izvoditi i u više koraka. Osnovna klasa za neku izvedenu klasu može i da bude izvedena iz drugih klasa. Na primer, za avione je rečeno da su letelice, a letelice su podgrupa vozila. S druge stane, i avioni bi mogli da se podele dalje na tice i teretne. Tome bi u jeziku C++ odgovarao još jedan korak u izvođenju klasa.

Definisanje izvedenih klasa

Se definišu naredbama **class** čiji je opšti oblik:

```
class Ident : način_izvođenja osnovna_klase , ... ,  
    način_izvođenja osnovna_klase  
    {  
        član član ...  
        public: član član ...  
        protected: član član ...  
        private: član član ...  
        ...  
    } ;
```

Klasa je izvedena ako u definiciji postoji niz identifikatora *osnovnih klasa*. Inače se ija već dobro poznata obična klasa (videti i definiciju klase u odeljku 3.1). Postojanje ovne klase (ili osnovnih klasa) označava se dvema tačkama (:) iza identifikatora klase i se definiše, a iza koje slede identifikatori osnovnih klasa. Članove izvedene klase čine članovi (polja i metode) njenih osnovnih klasa i članovi i se deklarišu u okviru posmatrane klase (unutar para vitičastih zagrada {}). Kaže se da edena klasa **nasleđuje** sve članove svojih osnovnih klasa. U slučaju više osnovnih klasa, ori se o **višestrukom nasleđivanju**.

Konstruktori, destruktur i metoda **operator=()** osnovne klase se ne nasleđuju, već se ierišu ako je to potrebno.

Od konstruktora generišu se podrazumevani konstruktor i konstruktor kopije. Prvi ima vno telo, a drugi kopira vrednost svog parametra polje po polje u objekat koji se stvara. Generisani destruktur, takođe ima prazno telo. Ovako generisani konstruktori i destruktur moguće su pozivanje eventualnih konstruktora i destruktora osnovnih klasa i klasnih novih izvedene klase. Za više detalja videti odeljak 5.5.

Generisana metoda **operator=()** dodeljuje vrednost polje po polje. To je osnovno načenje operatora = bez preklapanja operatora. Polja klasnih tipova dodeljuju se operatorskim funkcijama za dodelu vrednosti njihovih klasa. Nasledena polja osnovnih klasa deluju se operatorskim funkcijama za dodelu vrednosti njihovih osnovnih klasa.

Kontrola pristupa članovima klase ostvaruje se umetanjem oznaka **public**, **protected** i **private**. Time se klasa deli na javne, zaštićene i privatne delove. Javnim članovima, kao i do sada, pristup je moguć bez ograničenja kako iz unutrašnjosti klase tako i iz polja programa izvan klase, uključujući i izvedene klase. Pristup zaštićenim članovima je izvoljen samo iz same klase i iz svih klasa koje su iz nje izvedene. Kao i do sada, privatnim članovima pristup je dozvoljen samo iz klase čiji su članovi. Njima ne može da se istupa čak ni iz izvedenih klasa.

5.1 Definisanje izvedenih klasa

Zaštićeni članovi, kao međustepen kontrole pristupanja u odnosu na javne i privatne članove, uvedeni su radi veće efikasnosti pristupanja članovima osnovne klase iz izvedenih klasa. Smatra se da programeri izvedenih klasa dovoljno dobro poznaju osnovnu klasu da neće napraviti nikakvu štetu ako im se da toliki stepen slobode. Za razliku od njih, običnim korisnicima klasa (osnovnih i izvedenih) zaštićeni članovi su nepristupačni neposredno. Mogu da im pristupaju samo posredno pomoću odgovarajućih javnih metoda koje im projektanti klasa stave na raspolažanje. Taj put je, naravno, manje efikasan od neposrednog pristupa, ali je mnogo bezbedniji.

Na kraju, svaka klasa može da proglaši neke funkcije koje nisu članovi, kao i cele druge klase, prijateljima. Iz prijateljskih funkcija i/ili klasa pristup je dozvoljen svim članovima klase bez ograničenja. „Prijateljstvo“ se ne nasleđuje. Prijatelj osnovne klase ne postaje automatski prijatelj i klasama koje se iz nje izvedu. Nema nikakve prepreke da se prijatelji osnovne klase proglaše prijateljima i unutar izvedenih klasa. Štaviše, izvedena klasa može da ima i prijatelje koji to nisu i za njene osnovne klase.

Izvođenje klase iz neke osnovne klase može biti javno, zaštićeno ili privatno, dodavanjem kao *način_izvođenja* modifikator **public**, **protected** ili **private** ispred identifikatora osnovne klase. Kaže se i da je neka klasa javna, zaštićena ili privatna osnovna klasa date izvedene klase. U odsustvu modifikatora podrazumeva se privatno izvođenje (**private**).

Način izvođenja određuje stepen kontrole pristupa članovima osnovne klase preko primeraka izvedene klase prema tablici 5.1. Javni, zaštićeni i privatni članovi osnovne klase postaće javni, zaštićeni, odnosno privatni članovi javno izvedene klase. Javni članovi zaštićene osnovne klase biće zaštićeni članovi izvedene klase, dok zaštićenim i privatnim članovima stepen zaštite neće da se promeni. Na kraju, u slučaju privatnog izvođenja svi članovi osnovne klase postaće privatni članovi izvedene klase.

izvođenje	član osnovne klase		
	javan	zaštićen	privatan
javno	javan	zaštićen	privatan
zaštićeno	zaštićen	zaštićen	privatan
privatno	privatan	privatan	privatan

Tablica 5.1 – Stepeni zaštite članova osnovnih klasa u izvedenim klasama

Pošto su strukture (**struct**) potpuno ravnopravne klasama (videti odeljak 3.12), mogu da se definišu i izvedene strukture, s tim da se podrazumeva javno izvođenje. Izvođenje drugih tipova (struktura ili klasa) iz struktura je takođe moguće.

Iz unija (**union**) ne mogu da se izvedu drugi tipovi podataka niti unije mogu da se izvedu iz drugih tipova.

U slučaju javnog izvođenja svi javni članovi osnovne klase postaju javni članovi izvedene klase. Zbog toga s objektima izvedene klase mogu da se izvode sve radnje koje mogu s objektima osnovne klase i, pored njih, još neke druge radnje koje postoje samo u izvedenoj klasi. Kaže se da su izvedena klasa i osnovna klasa u odnosu *jesti* ili *je*: izvedena klasa je osnovna klasa s dodatnim mogućnostima.

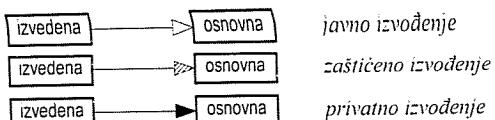
U slučaju privatnog ili zaštićenog izvođenja nasleđene metode iz osnovne klase ne mogu da se primene na objekte izvedene klase. Zbog toga oni ne mogu da se koriste kao da su

objekti tipa osnovne klase, pa odnos između izvedene i osnovne klase više nije je. U ovom slučaju može da se govori o odnosu *sadrži*: izvedena klasa *sadrži* jedan (bezimeni) primerak osnovne klase. Ovo može da se shvati i kao jedan specijalan vid kompozicije.

U praksi najveći značaj ima odnos *jeste* između izvedenih i osnovnih klasa, pa se najčešće koristi javno izvođenje klasa, a zaštićeno i privatno samo u izuzetnim slučajevima.

5.2 Dijagrami klasa za izvedene klase

Način prikazivanja nasleđivanja u dijagramima klasa prikazan je na slici 5.1.



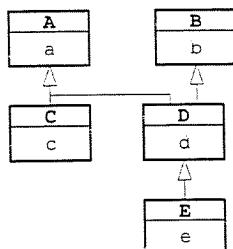
Slika 5.1 – Oznake nasleđivanja u dijagramima klasa

Linije su orijentisane od izvedene ka osnovnoj klasi jer u osnovnoj klasi se nikada ne zna koje izvedene klase postoje, a u izvedenim klasama se uvek zna iz kojih klasa su izvedene te klase. Pored toga, duž tih linija mogu da se sagledaju svi članovi koji postoje u objektu neke izvedene klase.

Uobičajeno je da se na vrhu slike crtaju osnovne klasе. Ispod njih se nalaze izvedene klasе, svaki sloj obeležava jedan korak izvođenja. To je u saglasnosti s nazivima *naslkasa* i *potklasa* koji se često koriste za osnovne i izvedene klasе.

Na slici 5.2 prikazan je dijagram klasa za klasе definisane sledećim naredbama:

```
class A { int a; };
class B { int b; };
class C : public A { int c; };
class D : public A, public B { int d; };
class E : public D { int e; };
```



Slika 5.2 – Primer dijagrama klasa s nasleđivanjem

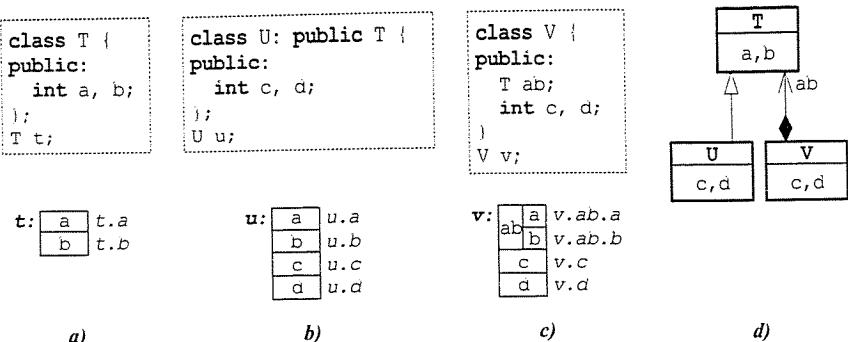
Sa slike se vidi da objekti tipa C imaju dva polja: specifično polje c i nasleđeno polje a od klase A. Objekti tipa D nasleđuju od klasa A i B, pa imaju polja d, a i b. Na kraju objekti tipa E nasleđuju sva polja od klase D, bez obzira da li su specifični za klasu E ili ih je klasa E nasledila. Zbog toga polja u klasi E su e, d, a i b.

5.3 Upotreba članova izvedenih klasa

Članovima osnovnih klasa neke izvedene klase pristupa se na isti način kao i njenim sopstvenim članovima. To znači, bez navođenja identifikatora osnovne klase. Unutar same klase samo identifikator člana je dovoljan, a izvan klase izrazima oblika `IObjekat.član` ili `IPokazivač->član`, gde su `IObjekat` primerak izvedene klase a `IPokazivač` pokazivač na primerke izvedene klase. Naravno, ovo sve važi pod uslovom da je pristup do člana osnovne klase dozvoljen.

Smatra se da izvedena klasа pored svojih članova poseduje, kao podobjekat, i jedan bezimeni primerak svoje osnovne klasе (slika 5.3.b). Pošto nema ime, podobjektu ne može da se pristupa kao celini već samo njegovim članovima pojedinačno. Treba jasno istaći razliku u odnosu na podobjekat koji je *polje tipa neke klase*. Tada se podobjektu kao celini pristupa pomoću identifikatora člana, a članovima podobjekta uobičajenom notacijom oblika `podobjekat.član` (slika 5.3.c). Izvedene klasе, naravno, nisu izmišljene samo zbog nekih notacionih pogodnosti. Njihov pravi smisao videće se u narednim odeljcima.

Dijagram klasa na slici 5.3.d prikazuje odnose među klasama T, U i V.



Slika 5.3 – Osnovna klasa i polje tipa klase

Redeklarisanje identifikatora unutar izvedene klase pokriva istoimeni član osnovne klasе. Član osnovne klasе koji je pokrivanjem postao nevidljiv može, ipak, da se koristi i unutar izvedene klase pomoću operatora za razrešenje dosega (::), tj. pomoću izraza oblika `Osnovna_klasa::član`. Posebno se naglašava, da, u slučaju metoda, sve istoimene metode u osnovnoj klasе postaju nevidljive, bez obzira na broj i tipove parametara tih metoda u osnovnoj i u izvedenoj klasi.

Evo primera za definisanje i korišćenje izvedenih klasa.

```
class Alfa {
    int a;
public:
    int b;
    void stavi (int x) { a = x; }
};
```

```

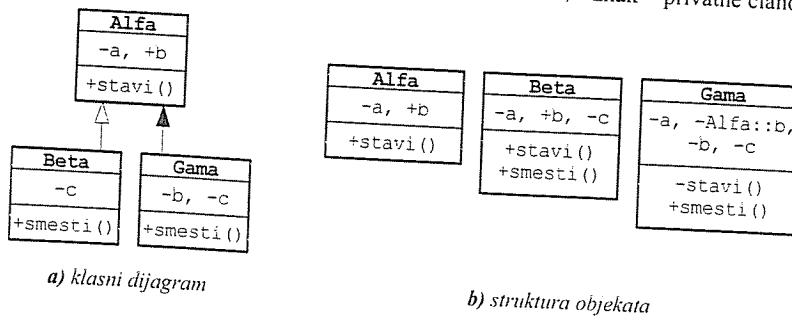
class Beta : public Alfa { // JAVNO IZVEDENA KLASA.
    int c;
public:
    void smesti (int x, int y, int z) {
        c = x; // Dodela privatnom polju Beta::c.
        b = y; // Dodela javnom polju Alfa::b.
        a = z; // GREŠKA: privatno polje Alfa::a.
        stavi (z); // Dodela privatnom polju Alfa::a.
    }
};

class Gama : Alfa { // PRIVATNO IZVEDENA KLASA.
    int b, c;
public:
    void smesti (int x, int y, int z) {
        c = x; // Dodela privatnom polju Gama::c.
        b = y; // Dodela privatnom polju Gama::b.
        Alfa::b = y; // Dodela pokrivenom polju Alfa::b.
        stavi (z); // Dodela privatnom polju Alfa::a.
    }
};

int main () {
    Alfa alfa; Beta beta; Gama gama;
    alfa.b = 55; // Javno polje klase Alfa.
    beta.b = 55; // Javno polje izvedene klase Beta.
    gama.Alfa::b = 55; // GREŠKA: U klasi Gama je privatno polje!
    alfa.stavi (23); // Javna metoda klase Alfa.
    beta.stavi (23); // Javna metoda i u klasi Beta.
    gama.stavi (23); // GREŠKA: U klasi Gama je privatna metoda!
    gama.smesti (10,55,23); // Javna metoda klase Gama.
}

```

Na slici 5.4.a prikazan je klasni dijagram navedenih klasa, a na slici 5.4.b struktura objekata tipa Alfa, Beta i Gama. Znak + označava javne članove, a znak - privatne članove.



Slika 5.4 – Javno i privatno izvođenje

Klase Alfa ima jedno privatno (a) i jedno javno (b) polje. Za pristup do privatnog polja predviđena je javna metoda `stavi()`.

Klase Beta je javno izvedena iz klase Alfa. Zbog toga ima dva privatna (a i c) i jedno javno (b) polje. Javna metoda `smesti()` služi za pristup do polja svoje klase. Neophodna je koristi i metoda `stavi()`. Štaviše, to polje je nepristupačno za neposredan pristup čak i za klasu Beta, pa i ona mora da koristi javnu metodu `stavi()` klase Alfa.

5.3 Upotreba članova izvedenih klasa

Klase Gama je privatno izvedena iz klase Alfa. Zato su svi nasleđeni članovi privatni za klasu Gama. Imala četiri privatna polja: sopstvena dva b i c i nasleđena dva a i b. Problem je što postoje dva polja s istim identifikatorom. U ovakvim situacijama identifikator b označava sopstveno (specifično) polje, dok nasleđeno polje treba označavati sa `Alfa::b`. Skreće se pažnja na to da je i nasleđena metoda `stavi()` privatna u klasi Gama. Zbog toga je metoda `smesti()` sada jedino sredstvo za pristup do svih polja klase Gama.

Na kraju, glavna funkcija prikazuje mogućnosti pristupanja članovima primeraka alfa, beta i gama klase Alfa, Beta i Gama. U sve tri naredbe za dodelu vrednosti pristupa se, u stvari, polju b (osnovne) klase Alfa. Za slučaj objekta gama to nije dozvoljeno, zbog privavnog izvođenja klase Gama iz klase Alfa. Slična je situacija i s neposrednim pozivanjem metode `stavi()`. U posmatranom primeru jedini dozvoljen način korišćenja objekta gama je pozivanje javne metode `smesti()` u klasi Gama kojoj je gama tekući objekat.

Redefinisanje identifikatora metoda u izvedenoj klasi ne smatra se preklapanjem imena funkcija. Sve metode osnovne klase koje imaju isto ime kao i metoda izvedene klase postaće nevidljive, bez obzira na broj i tipove parametara. Za pozivanje bilo koje od njih neophodna je upotreba operatora ::.

Evo primera za pokrivanje imena metoda:

```

class Osn {
    ...
public:
    void m () ;
    void m (int) ;
};

class Izv: public Osn {
    ...
public:
    void m (int); // Pokriva i Osn::m() i Osn::(int).
};

int main () {
    Izv izv;
    izv.m (); // GREŠKA: ne vidi se Osn::m().
    izv.Osn::m (); // Ovako može.
    izv.m (5); // Poziva se Izv::m().
    izv.Osn::m (5); // Poziva se Osn::m().
}

```

Metoda `Izv::m(int)` pokriva obe metode `Osn::m()` i `Osn::(int)`, bez obzira na razliku u broju i tipovima parametara. Zbog toga je u glavnoj funkciji izraz `izv.m()`; pogrešan. Za pozivanje pokrivene metode iz osnovne klase mora da se navede i ime osnovne klase (`izv.Osn::m()`).

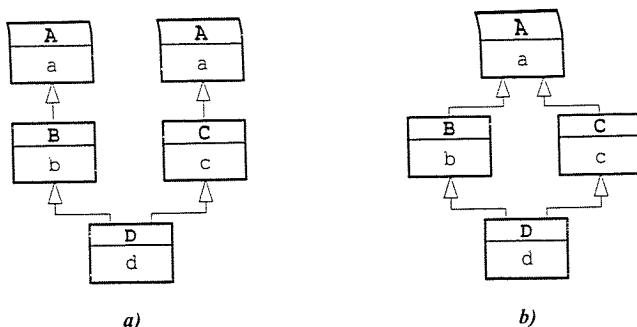
5.4 Virtuelne osnovne klase

Neka su date dve klase izvedene iz iste osnovne klase. Ako se iz njih izvede neka treća klasa, u tako dobijenoj klasi postojiće dva primerka njihove zajedničke osnovne klase. Na primer, posle sledećeg niza naredbi:

```
class A { int a; };
class B : public A { int b; };
class C : public A { int c; };
class D : public B, public C { int d; };
D x;
```

objekat x imaće pet članova. Član x.d je specifičan član klase D. Članovi x.c i x.b su nasleđeni iz neposrednih osnovnih klasa C i B. Postoje još dva člana s istim identifikatorom a koji su oba nasleđeni iz klase A, jedan preko klase C a drugi preko klase B. Mogu da se označe sa x.C::a i x.B::a, tj. treba da se naznače klase preko kojih su nasleđeni.

Slika 5.5.a prikazuje dijagram klasa za prethodni skup klasa.



Slika 5.5 – Nasleđivanje duž više putanja

Višestruko ugrađivanje članova date osnovne klase u izvedenu klasu prilikom višestrukog nasleđivanja u većini slučajeva smeta. Da bi se to izbeglo, prvočitna osnovna klasa treba da bude virtualna osnovna klasa za svoje neposredno izvedene klase. To se postiže dodavanjem modifikatora **virtual** ispred identifikatora osnovne klase pri definisanju izvedene klase.

Definicije za posmatrani skup klasa treba da izgledaju ovako:

```
class A { int a; };
class B : virtual public A { int b; };
class C : virtual public A { int c; };
class D : public B, public C { int d; };
D x;
```

Objekat x sada ima sledeća četiri člana (slika 5.5.b): x.d, x.c, x.b i x.a. Treba uočiti da je, za razliku od prethodnog slučaja, sada x.a jedinstven i zato nikakva oznaka klase nije potrebna.

Ako je neka klasa virtualna osnovna klasa za neke izvedene klase a nije virtualna za druge, prilikom višestrukog nasleđivanja naslediće se jedan zajednički podobjekat tipa te osnovne klase preko svih međuklasa za koje je ona virtualna i po jedan podobjekat preko svih međuklasa za koje ona nije virtualna osnovna klasa.

5.5 Stvaranje i uništavanje primeraka izvedenih klasa

Prilikom definisanja konstruktora za izvedenu klasu, u listi inicijalizatora pre tela konstruktora (videti odeljak 3.4.3), mogu da se navedu i inicijalizatori za osnovne klase. U fazi inicijalizacije primeraka klasa nasleđena polja još ne postoje, pa ne mogu da se navedu odvojeni inicijalizatori za njih. Moguća je samo inicijalizacija nasleđenog podobjekta tipa osnovne klase kao celine. Zbog toga ispred argumenata inicijalizatora treba navesti identifikatore osnovnih klasa. Naravno, nasleđena polja unutar tela konstruktora mogu da se koriste i pojedinačno, ali postavljanje njihovih vrednosti na taj način više nije inicijalizacija već dodela vrednosti.

Tok stvaranja novog primeraka izvedene klase poseduje jedan korak više od toka stvaranja primerka obične klase, koji je opisan u odeljku 3.4.5. Potpuni tok stvaranja objekta tipa izvedene klase je sledeći:

- Izvršavaju se konstruktori osnovnih klasa po redosledu navođenja osnovnih klasa u definiciji izvedene klase (ne po redosledu navođenja inicijalizatora u definiciji konstruktora). U odsustvu inicijalizatora koristiće se podrazumevani konstruktori osnovnih klasa. Na taj način se inicijalizuju nasleđena polja iz osnovnih klasa.
- Izvršavaju se konstruktori polja klasnih tipova po redosledu navođenja polja u definiciji izvedene klase. Ako postoje inicijalizatori njihovi argumenti predstavljaju argumente pozivanih konstruktora. Ako nema inicijalizatora, opet, pozivaju se podrazumevani konstruktori.
- Inicijalizuju se polja standardnih tipova za koje postoje inicijalizatori.
- Izvršava se telo konstruktora izvedene klase. To je druga mogućnost za postavljanje vrednosti svih polja, kako specifičnih tako i nasleđenih. Tom prilikom se, međutim, za dodele vrednosti klasnim poljima više ne koriste konstruktori već metode **operator=()**.

Prilikom uništavanja objekata izvedenih klasa redosled uništavanja podobjekata je obrnut od redosleda njihove inicijalizacije (videti i odeljak 3.5). Preciznije rečeno:

- prvo se izvršava telo destruktora izvedene klase,
- zatim se izvršavaju destruktori polja klasnih tipova po obrnutom redosledu izvršavanja njihovih konstruktora, i
- na kraju se izvršavaju destruktori osnovnih klasa, opet, po obrnutom redosledu pozivanja konstruktora tih klasa.

Evo primera za prikazivanje toka inicijalizacije izvedenih objekata:

```
class Osn1 {
    int* a;
public:
    Osn1 () : a (0) {}
};

class Osn2 {
    char* b;
    double c;
public:
    Osn2 () { b = 0; c = 5; }
    Osn2 (char* x, double y) : c (y) { b = x; }
};
```

```

class Izv : public Osn2, public Osn1 {
    int p, q;
    Osn1 r;
    Osn2 s, t;
    double u, v;
public:
    Izv (int, char*, double);
    Izv::Izv (int x, char* y, double z) : p(x+2), Osn2(y, z), s("ne", x) {
        q = p + z;
    }
int main () { Izv k (1, "Zdravo", 2); }
Tok inicijalizacije objekta k u glavnoj funkciji je sledeći:
Osn2 ("Zdravo", 2.0)
    ~ k.c = 2.0
    ~ k.b = "Zdravo"
    ~ k.a = 0
    ~ k.r.a = 0
    ~ k.s.b = "ne"
    ~ k.s.c = 1.0
    ~ k.t.b = 0
    ~ k.t.c = 5.0
    ~ k.p = 3
    ~ k.q = 5
    ~ k.u = 3 + 2.0

```

Treba uočiti da polja k.u i k.v ostaju neinicijalizovani jer niti su navedeni inicijalizatori za njih niti su im dodeljene vrednosti u telu konstruktora.
Skrće se pažnja na to da se za inicijalizaciju nasleđenih podobjekata i polja klasnih konstruktora poziva neki konstruktor. U odsustvu inicijalizatora poziva se podrazumevani konstruktor. Ako ne postoji podrazumevani konstruktor, a postoje drugi konstruktori, prevođilac će da prijavlji grešku. Jedina opasna situacija je kada u osnovnoj klasi ili u klasi praznog tela i poziva njega, pa nasledeni podobjekat ili polje klasnog tipa, u suštini, ostaje neinicijalizovan.

5.6 Konverzija tipa između osnovnih i izvedenih klasa

Objekat osnovne klase može da se inicijalizuje objektom javno izvedene klase. Tom prilikom će konstruktor kopije osnovne klase prekopirati samo polja izvedene klase koja su nasleđena od osnovne klase.

Objektu osnovne klase može da se dodeljuje vrednost objekta javno izvedene klase. Tom prilikom će operatorska funkcija za dodelu vrednosti osnovne klase prekopirati samo polja izvedene klase koja su nasleđena od osnovne klase.

Ovakva inicijalizacija i dodela vrednosti je dozvoljena, jer u slučaju javnog izvođenja izvedena klasa je osnovna klasa s nekim dodatnim mogućnostima. Naravno, te dodatne mogućnosti se izgube u toku tih operacija.

U slučaju privatnog ili zaštićenog izvođenja izvedena klasa se ne ponaša istovetno svojoj osnovnoj klasi, zbog toga tada automatski nije dozvoljena inicijalizacija objekta osnovne klase objektom izvedene klase, niti da se objektu osnovne klase dodeljuje vrednost objekta izvedene klase. Ako je to baš potrebno, neophodno je obezbediti konverziju tipa iz

izvedene klase u osnovnu klasu preklapanjem operatora za konverziju tipa u izvedenoj klasi.

Inicijalizacija objekta izvedene klase objektom osnovne klase, odnosno dodela vrednosti objekta osnovne klase objektu izvedene klase nikada nije automatski dozvoljena. Za to je neophodno da postoji definisana konverzija tipa iz osnovne klase u izvedenu klasu, na primer, u obliku konstruktora konverzije u izvedenoj klasi. Ta konverzija treba da odredi vrednosti specifičnih polja u izvedenoj klasi, a koja ne postoje u osnovnoj klasi.

Evo primera za ilustraciju prethodnih stavova:

```

class Osn { ... };
class Izv : public Osn { ... };
void f (Osn);
int main () {
    Izv izv;
    Osn osn;
    osn = izv;           // Dodata vrednosti.
    izv = osn;           // GREŠKA: Šta da se dodeli specifičnim poljima?
    f (izv);             // Inicijalizacija parametra konstruktorom kopije.
}

```

Pokazivaču po na objekte osnovne klase Osn može da se dodeli vrednost pokazivača pi na objekte javno izvedene klase Izv bez eksplisitne konverzije tipa. Tom prilikom se podrazumeva automatska konverzija tipa vrednosti pokazivača pi iz tipa Izv* u tip Osn*. Drugim rečima, izraz po=pi tumači se kao po=static_cast<Osn*>(pi). Ovo je dozvoljeno zato što je pomoću takvog pokazivača moguće pristupiti samo nekim od članova izvedenog objekta, tačnije samo članovima nasleđenim od osnovne klase. Pošto je time sužen broj pristupačnih članova, ova operacija je sigurno bezopasna. Dalje, skrće se pažnja na činjenicu, da je javnim izvođenjem stepen kontrole pristupanja nasleđenim članovima u izvedenoj klasi isti kao i u osnovnoj klasi.

Pokazivaču na objekte osnovne klase dodata vrednosti pokazivača na objekte privatno ili zaštićeno izvedene klase je dozvoljena samo uz eksplisitnu konverziju tipa. Drugim rečima, u ovom slučaju izraz po=pi nije dozvoljen, već mora da se piše eksplisitno po=static_cast<Osn*>(pi). Ovo ograničenje je uvedeno zato što je stepen zaštite nasleđenim članovima u izvedenoj klasi restriktivniji od zaštite u osnovnoj klasi. Posledica toga je da pomoću pokazivača po može da se pristupa i nekim članovima objekta izvedene klase, kojima pomoću pokazivača pi ne bi moglo. Eksplisitna konverzija se zahteva da bi programer time dao do znanja prevodiocu da je svestan opasnosti kojima se izlaže tom dodelom vrednosti. Ovaj postupak treba izbegavati.

Dodata vrednosti pokazivača na objekte osnovne klase pokazivaču na objekte izvedene klase je moguća isključivo pomoću eksplisitne konverzije tipa. Drugim rečima, izraz pi=po nije dozvoljen, već mora da se piše eksplisitno pi=static_cast<Izv*>(po). Ovo je krajnje opasan poduhvat i treba ga koristiti maksimalno oprezno. Pomoću pokazivača na izvedenu klasu može da se označi član koji ne postoji u osnovnoj klasi. U tom slučaju pristupiće se delu memorije koji ne pripada datom objektu! Eksplisitna konverzija tipa se i ovde zahteva da bi programer naznačio da ono što radi, radi svesno. Skrće se pažnja na to da u praksi nisu retke situacije kada po u stvari pokazuje na objekat tipa izvedene klase. Kada se sa sigurnošću zna da je Izv tip objekta na koji pokazuje po, sme da se iskoristi izraz static_cast<Izv*>(po) radi pristupanja specifičnim članovima tog objekta.

Upućivači osnovnih i izvedenih klasa mogu da se koriste na isti način kao i pokazivači. Upućivač uo na objekte tipa javne osnovne klase `Osn` može prilikom inicijalizacije da se dodeli vrednost upućivača u na objekte tipa izvedene klase `Izv` bez eksplisitne konverzije. U slučaju da izvođenje nije javno, mora da se koristi eksplisitna konverzija. Eksplisitna konverzija je neophodna i prilikom inicijalizacije upućivača u na izvedene objekte upućivačem uo na osnovne objekte. Razlozi su isti ako i kod pokazivača.

Skreće se pažnja na to da su najčešći trenuci inicijalizacije upućivača pozivanja funkcija. U tim prilikama parametri koji su upućivači inicijalizuju se upućivačima na odgovarajuće argumente. Iz gore rečenog sledi, da bez problema može da se navede kao argument objekat tipa izvedene klase kad god je odgovarajući parametar upućivač na osnovnu klasu te izvedene klase, pod uslovom da je izvođenje izvršeno javno.

Pošto je uobičajeno da se u dijagramima klasa osnovne klase crtaju iznad izvedenih klasa, konverzija tipa pokazivača ili upućivača na osnovnu klasu u tip pokazivača, odnosno upućivača na izvedenu klasu naziva se i **konverzija nadole**. Slično, konverzija u suprotnom smeru naziva se i **konverzija nagore**.

Evo primera za korišćenje pokazivača i upućivača na osnovne i izvedene klase:

```
class Osn { ... };
class Izv : public Osn { ... };

void f (Osn&);

int main () {
    Izv izv, *pi = &izv;
    Osn* po = pi;           // po = static_cast<Osn*>(pi);
    f (izv);                // f (static_cast<Osn&>(izv));
}
```

Interesantna posledica činjenice da pokazivači na osnovnu klasu mogu da pokazuju na objekte izvedenih klasa je mogućnost obrazovanja nizova i listi čiji elementi pokazuju na objekte različitih (srodnih) tipova. Tako, na primer, mogu da se obrazuju skupovi vozila. Ništa ne smeta da među njima postoje avioni, brodovi i bicikli. Ako zatreba, ima načina da se ti objekti tretiraju primereno svojim specifičnostima. Kako se to postiže, tema je sledećeg odeljka.

5.7 Virtuelne metode

Jedna od osnova objektno orijentisanog programiranja je **polimorfizam**, pod čime se podrazumeva mogućnost ponašanja programa shodno tipovima obradivanih objekata. Na primer, prilikom obrade skupa geometrijskih figura površine figura se izračunavaju pomoću različitih formula u zavisnosti od vrste figura.

U jeziku C++ mehanizam virtuelnih metoda je sredstvo koje omogućava automatsku primenu odgovarajućeg postupka za dobijanje iste stvari (na primer, površine figure) za objekte različitih tipova (na primer, krugova i trouglova).

Virtuelna metoda je metoda osnovne klase koja može da se zameni istoimenom metodom odgovarajuće izvedene klase kada se pozove za objekat koji pripada izvedenoj klasi, čak i u slučajevima kada se poziva pomoću pokazivača ili upućivača na objekte osnovne klase. Naglašava se da prijateljske funkcije neke klase ne mogu da budu virtuelne za tu klasu.

Virtuelne metode se deklarišu dodavanjem modifikatora **virtual** na početku njihove deklaracije u osnovnoj klasi. Prilikom ponovnog deklarisanja u izvedenim klasama modifikator **virtual** se podrazumeva, ne mora da bude eksplisitno naveden.

Deklaracije date virtuelne metode u osnovnoj i u svim izvedenim klasama moraju da budu istovetne. Drugim rečima, sve moraju da imaju isti broj i tipove (pravih) parametara i da daju rezultat istog tipa. Jedino u čemu se one razlikuju je tip tekućeg objekta, tj. tip skrivenog parametra koji poseduju sve metode klase pa i virtuelne metode. U specijalnim slučajevima dozvoljena je još jedna razlika: ako je vrednost virtuelne metode u osnovnoj klasi pokazivač ili upućivač na osnovnu klasu, vrednost u javno izvedenoj klasi može da bude pokazivač, odnosno upućivač na tu izvedenu klasu.

Prilikom pozivanja virtuelne metode pomoću pokazivača ili upućivača na objekte

osnovne klase, pozivaće se istoimena metoda one izvedene klase čiji primerak u tom momentu predstavlja dati pokazivač ili upućivač.

Klase koje poseduju bar jednu virtuelnu metodu nazivaju se **polimorfne klase**.

Za pokazivače na objekte polimorfne klase kaže se da imaju statički i dinamički tip. **Statički tip** je tip naveden u naredbi za definisanje, zna se već u vreme prevodenja i ne može da se promeni u toku izvršavanja programa. **Dinamički tip** je pokazivač na objekte istog tipa kao i objekat na koji pokazivač pokazuje u datom trenutku. Može da bude jednak statičkom tipu ili pokazivač na objekte bilo koje klase koja je izvedena iz klase na koju pokazuje statički tip. Pomoću pokazivača mogu da se pozivaju samo metode koje postoje u klasi koja odgovara statičkom tipu (tj. u osnovnoj klasi). Ako se desi da je pozvana metoda virtuelna, umesto nje biće pozvana istoimena metoda na osnovu dinamičkog tipa (tj. iz odgovarajuće izvedene klase). Može se reći da statički tip pokazivača određuje *šta može da se uradi* s pokazanim objektom, a dinamički tip *kako da se to uradi*.

I upućivači na polimorfne klase imaju statički i dinamički tip, i ponašaju se analogno pokazivačima na objekte polimorfne klase.

Data virtuelna metoda ne mora da se definiše u svim izvedenim klasama iz date osnovne klase. Ako u nekoj od izvedenih klasa ne postoji pozvana virtuelna metoda, pozivaće se virtuelna metoda osnovne klase iz koje je izvedena klasa objekta za koju se poziva data virtuelna metoda. Nije bitno kog je tipa identifikator koji se koristi za označavanje objekta. Pored identifikatora objekta tipa izvedene klase, može da se koristi i pokazivač ili upućivač na osnovnu ili na izvedenu klasu.

Treba posebno istaknuti razliku između preklapanja imena metoda (funkcija) i virtuelnih metoda. U oba slučaja radi se o više istoimenih metoda od kojih treba pozvati jednu. Međutim, mehanizam odabiranja se kvalitativno razlikuje.

Kod preklapanja imena metoda bitno je da se sve istoimene metode međusobno razlikuju dovoljno na osnovu broja i tipova parametara da bi prevodilac mogao u vreme prevodenja da učini jednoznačan izbor. Tipovi vrednosti tih metoda nisu bitni. Tu se, dakle, radi o statičkom izboru pre početka izvršavanja programa. Bira se među metodama koje pripadaju istoj klasi.

Kod virtuelnih metoda sve moraju da imaju parametre istih tipova i da daju rezultat istog tipa. Štaviše, pozivanje se vrši, na izgled, za istu vrstu objekata, za objekte tipa osnovne klase (koristi se pokazivač ili upućivač na objekte osnovne klase). Pozvana metoda se bira dinamički, u momentima pozivanja. Kriterijum odabiranja je tip objekta na koji pokazuje ili upućuje korišćeni identifikator, a ne tip identifikatora. Bira se među metodama koje pripadaju različitim klasama koje sve imaju neku zajedničku osnovnu klasu.

Upravo korišćenje virtualnih metoda čini programiranje na jeziku C++ objektno orijentisanim. Ponašanje programa zavisi od tipa obradivog objekta, a ne od tipa identifikatora pomoću kojeg se dolazi do objekta.

Čitajući (statički) tekst programa, kada se vidi poziv neke virtualne metode, ne može da se utvrdi koja će, iz skupa istoimenih metoda, biti stvarno pozvana. Zbog toga ceo skup može da se smatra jednom **polimorfnom metodom** čije ponašanje zavisi od trenutne situacije, tj. od konkretnog objekta, u momentu pozivanja.

Ako u izvedenoj klasi postoji metoda čije je ime jednako imenu neke virtualne metode u osnovnoj klasi ali koja ima parametre koji se ne slažu po broju i po tipovima, metoda u izvedenoj klasi se ne smatra virtualnom metodom i ne može da se poziva mehanizmom virtualnih metoda. Tada se radi samo o sakrivanju imena metoda (videti odeljak 5.3).

Greška je ako se neka metoda od istomene virtualne metode u osnovnoj klasi razlikuje samo po tipu vrednosti funkcije. Izuzev, ako je tip vrednosti u osnovnoj klasi pokazivač ili upućivač na osnovnu klasu, kada tip vrednosti sme de bude i pokazivač, odnosno upućivač na izvedenu klasu, pod uslovom da je izvođenje javno.

Konstruktori ne mogu da budu virtualne metode. Kada se stvara novi objekat, mora eksplicitno da se kaže kojeg je tipa objekat koji treba napraviti.

Destruktori mogu da budu virtualni. Kada se uništava objekat, zna se kog je tipa polimorfnih osnovnih klasa budu virtualne metode. To garantuje da će objekat u dinamičkoj klasi, biti ipak uništen destruktorm svoje (izvedene) klase. Ako destruktur nije virtualna metoda, uništavanje objekta izvedene klase posredstvom pokazivača ili upućivača na osnovnu klasu dovelo bi do uništavanja samo delova koji su nasledeni iz osnovne klase. To obično nije ono što bi trebalo da se desi.

Evo primera za definisanje i korišćenje virtualnih metoda:

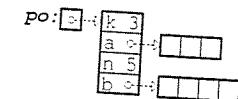
```
class Osn {
    int k, *a;
public:
    Osn ();
    virtual void vml (); // Konstruktor, ne može da bude virtualan.
    virtual void vm2 ();
    virtual void vm3 ();
    void m ();
    ~Osn (); // Obična metoda.
}; // Destruktor, trebalo bi da bude virtualan!

class Izv : public Osn {
    int n, *b;
public:
    Izv ();
    void vml ();
    void vm2 (int); // Virtuelna metoda.
    char vm3 ();
    void m ();
    ~Izv (); // Obična metoda.
}; // Nije virt. metoda (samo pokrivanje imena)!
```

```
int main () {
    Osn* po = new Izv;
    po->vml ();
    po->vm2 ();
    po->m ();
    delete po;
    Izv* pi = new Izv;
    pi->vm2 ();
    pi->Osn::vm2 ();
    pi->vm2 (5);
    delete pi;
}
```

// Inicijalizacija pomocu Izv::Izv(), posle standardne konverzije Izv u Osn.
// Poziva se Izv::vml().
// Poziva se Osn::vm2() (ne postoji Izv::vm2())
// PROBLEM: poziva se Osn::~Osn()!
// GREŠKA: ne vidi se Osn::vm2().
// Ovako može (više nije polimorfno).
// Poziva se Izv::vm2(int).
// Poziva se Izv::~Izv().

U prvoj naredbi glavne funkcije objekat tipa izvedene klase se inicijalizuje konstruktorom te izvedene klase Izv(). Taj konstruktor pre izvršavanja tela pozvaće konstruktor osnovne klase Osn() (videti odeljak 5.5). Rezultat operatora new, koji je tipa Izv*, pre inicijalizacije pokazivača po konvertuje se u tip Osn*. Slika 5.6 prikazuje mogući izgled specifična polja n i b telo konstruktora Izv().



Slika 5.6 – Objekat izvedene klase s dinamičkim sadržajem

Izrazom po->vml() prvo se, na osnovu statičkog tipa Osn* pokazivača po, potraži metoda vml() u osnovnoj klasi. Pošto je ona virtualna metoda, na osnovu dinamičkog tipa Izv* pokazivača po potraži se i pozove se metoda vml() iz izvedene klase.

Slično tome, izrazom po->vm2() prvo se u osnovnoj klasi potraži metoda vm2() i, pošto je ona virtualna, potražiće se u izvedenoj klasi. Pošto u klasi Izv ne postoji ta metoda (postoji samo vm2(int)) koja ne može da bude zamena za metodu vm2()), pozvaće se metoda vm2() iz osnovne klase.

Metoda m() nije virtualna ni u osnovnoj klasi, pa se izrazom po->m() poziva metoda iz osnovne klase, bez obzira što je dinamički tip pokazivača po tog momenta Izv*. U datoj situaciji, metoda m() iz izvedene klase mogla bi da se pozove izrazom ((Izv*)po)->m(), ili izrazom pi->m(), gde je pi pokazivač čiji je (statički) tip pokazivač na objekte izvedene klase (Izv*).

U naredbi delete po, pošto destruktur osnovne klase nije virtualna metoda i pošto je pokazivač po tipa Osn*, na objekat tipa Izv primeniće se destruktur iz klase Osn. To u posmatranom slučaju nije zadovoljavajuće. Naime, destruktur ~Osn() će da oslobođi samo memoriju koja je dodeljena dinamičkom nizu a (slika 5.6), dok će memorija koja je dodeljena dinamičkom nizu b ostati neoslobođena. Kad bi destruktur ~Osn() bio virtualan, na osnovu dinamičkog tipa pokazivača po pozvao bi se destruktur ~Izv() u čijem telu bi se uništio dinamički niz b. Posle bi se pozvao destruktur ~Osn() (videti odeljak 5.5) koji bi uništio dinamički niz a. Na taj način bi se sva memorija koja je dinamički dodeljena pri stvaranju objekta tipa Izv oslobođila prilikom uništavanja objekta.

Zbog svega rečenog, preporučuje se da se u polimorfnoj osnovnoj klasi uvek definiše virtualan destruktur, makar i praznog tela (ako u osnovnoj klasi nema šta da se uništava).

Nekoj od budućih izvedenih klasa biće to od koristi. Skreće se pažnja na to da u nedostatku destruktora u osnovnoj klasi generiraće destruktor praznog tela (videti odeljak 3.5), ali taj destruktor neće biti virtualan.

Poslednja grupa naredbi u prethodnoj glavnoj funkciji pokazuje korišćenje objekta izvedene klase pomoću pokazivača na izvedenu klasu. Tada se, kad god pozvana metoda postoji u izvedenoj klasi, poziva metoda iz te klase. Ako neka metoda ne postoji u izvedenoj klasi, poziva se nasleđena metoda iz osnovne klase. Problemi mogu nastati ako je ta metoda pokrivena istoimenom metodom u izvedenoj klasi. U posmatranom primeru to je slučaj s metodom `vm2()`. Da bi se ona pozvala, neophodno je pisati `pi->Osn::vm2()`, čak i ako u izvedenoj klasi ne postoji metoda s istim prototipom.

5.8 Apstraktne klase

Virtuelna metoda koja nije definisana u osnovnoj klasi naziva se **apstraktna metoda**. Naziva se i *čista virtuelna metoda*. Apstraktna metoda označava se sa `=0`; umesto tala funkcije prilikom deklarisanja.

Klasa koja sadrži bar jednu apstraktnu metodu naziva se **apstraktna klasa**. Ne mogu da se stvaraju objekti tipa apstraktne klase.

Svrha apstraktnih klasa je da se iz njih izvode druge klasе. Mogu da se definišu pokazivači i upućivači na apstraktne klasе. Oni, u nedostatku objekata tipa osnovne klase, mogu da označavaju samo objekte tipa izvedenih klasа.

Klasa izvedena iz apstraktne klase koja ne sadrži definiciju bar za jednu apstraktnu metodu osnovne klase, i sama je apstraktna klasа.

Imena apstraktnih klasа i metod u dijagramima klasа se obično pišu kurzivom.

Evo primera apstraktne klase:

```
class Osn {
    ...
public:
    virtual void avm () =0;           // Apstraktna metoda.
    virtual void vm ();               // Obična virtuelna metoda.
};

class Izv : public Osn {
    ...
public:
    void avm ();                     // Ostvarenje apstraktne metode.
};

void main () {
    Izv izv, *pi=&izv, &ui=izv; // Objekat, pokazivač i upućivač
                                // tipa izvedene klase.
    Osn osn;
    Osn *po=&izv, &uo=izv;

    po->avm ();
    po->vm ();
}
```

5.9 Dinamička konverzija tipa podataka

Pokazivač na osnovnu klasu može da se dodeli vrednost pokazivača na javno izvedenu klasu, odnosno upućivač na osnovnu klasu može da se inicijalizuje objektom javno izvedene klase (videti odeljak 5.6). Ovo je naročito korisno kada treba obezbediti polimorfnu obradu skupa raznorodnih ali srodnih objekata (koji imaju zajedničku osnovnu klasu).

Kad pokazivač na osnovnu klasu pokazuje na objekat tipa izvedene klase, da bi se pristupilo specifičnim članovima izvedene klase, neophodno je primeniti eksplicitnu verziju tipa.

Za nepolimorfne klasе (koje ne sadrže virtualne metode), programer bi trebalo da je u stanju da u vreme pisanja programa odredi stvarni tip objekta i da konverziju zatraži operatom `static_cast` (videti odeljak 2.3.2) ili *zastarelim operatorom za konverziju tipa* (`tip`). Posledice su nepredvidive ako se desi da pokazivač ne pokazuje na tip koji je naznačio programer.

U slučaju polimorfnih klasа može se desiti da pokazivač na određenom mestu u programu u različitim trenucima pokazuje na objekte različitih (doduše srodnih) tipova. Statička konverzija tipa tada nije zadovoljavajuća, već je potrebna dinamička konverzija koja je u stanju da proveri ispravnost konverzije u datom trenutku.

Dinamička konverzija tipa može da se uradi operatom `dynamic_cast` izrazom oblika:

```
dynamic_cast < Izv * > ( po )
```

gde je `po` pokazivač na polimorfnu osnovnu klasu (tip `Osn*`), a `Izv` klasа izvedena iz klase `Osn` u proizvoljnom broju koraka izvođenja. Rezultat je pokazivač tipa `Izv*` koji pokazuje na objekat na koji pokazuje pokazivač `po`, pod uslovom da je pokazani objekat stvarno tipa `Izv`, ili je tipa klase koja je izvedena iz klase `Izv`. Vrednost izraza je nula ako se desi da `po` pokazuje na objekat koji nije odgovarajućeg tipa, na primer ako je tipa druge izvedene klasе iz klase `Osn`, ali koja nije potklasa klasе `Izv`.

U slučaju konverzije tipa upućivača, rezultat izraza

```
dynamic_cast < Izv & > ( uo )
```

će biti upućivač tipa `Izv&` na objekat na koji upućuje upućivač `uo` tipa `Osn&` pod uslovom da je upućivan objekat stvarno tipa `Izv` ili je tipa klase koja je izvedena iz klase `Izv`. Ako `uo` upućuje na objekat neodgovarajućeg tipa greška se prijavljuje izuzetkom tipa `bad_cast`. Ako se taj izuzetak ne obrađuje od strane programa, program se prekida (izuzeci su objašnjeni u poglavljju 6). Tip `bad_cast` definisan je u zagлавlu `<typeinfo>`.

Upotreba operatora `dynamic_cast` za konverziju tipa pokazivača ili upućivača nagore (iz izvedene klase u osnovnu klasu) je dozvoljena, ali nije neophodna jer se takva konverzija vrši i automatski.

Evo primera za upotrebu operatora `dynamic_cast`:

```
class A
    | virtual void vm () {} ;
class B: public A
    | public: int b; B (int x): b(x) {} ;
class C: public A
    | public: int c; C (int x): c(x) {} ;
```

// Polimorfna osnovna klasa.

// Prva javno izvedena klasa.

// Druga javno izvedena klasa.

```

int main () {
    B* pa = new B (5);
    if (B* pc = dynamic_cast<B*>(pa)) // Da li pa pokazuje na objekat tipa B.
        cout << pc->b << endl;
    if (C* pc = dynamic_cast<C*>(pa)) // Da li pa pokazuje na C?
        cout << pc->c << endl;
    B* uc = dynamic_cast<B*>(*pa); // Tumači *pa kao objekat tipa B.
    C* uc = dynamic_cast<C*>(*pa); // GREŠKA: *pa nije tipa C
}

```

Prvom naredbom glavne funkcije definisce se pokazivač pa na osnovnu klasu A i inicijalizuje se adresom dinamičkog objekta tipa izvedene klase B.

U uslovu prve naredbe if definisce se pokazivač pc tipa izvedene klase B rezultatom konverzije pokazivača pa u tip B*. Pošto pokazivač pa stvarno pokazuje na objekat tipa B, rezultat je različit od nule i izvršiće se ispisivanje polja b klase B (pb->b). Pošto pa ne pokazuje na objekat tipa C, pokazivač pc u drugoj naredbi if se inicijalizuje nulom i zbog toga neće doći do pokušaja ispisivanja vrednosti nepostojećeg polja c (pc->c).

U nastavku, upućivač uc će uspešno da se inicijalizuje objektom na koji pokazuje pa konverzije objekta na koji pokazuje pa (*pa) koji nije tipa C u upućivač na klasu C (C&), već se primećuju tek u vreme izvršavanja programa.

5.10 Dinamičko određivanje tipa podataka △

Informacije o tipu podataka u toku izvršavanja programa mogu da se dobiju pomoću operatora typeid izrazom opsteg oblika:

```
typeid ( izraz )
typeid ( naziv_tipa )
```

ili

Izraz može da bude proizvoljnog tipa: standardnog, nestandardnog, prostog ili složenog. Vrednost izraza se ne izračunava, ako sadrži operatore s bočnim efektima oni neće da dese. Naziv tipa takođe, može da označava bilo koji tip.

Rezultat operatora typeid je upućivač na nepromenljiv objekat tipa type_info (tj. stavlja objekat polimorfne klase rezultat će se odnositi na dinamički tip operanda, a na prijavljuje izuzetkom tipa bad_typeid). Ako se taj izuzetak ne obrađuje od strane programa program se prekida (izuzeći su objašnjeni u poglavlju 6).

Klasa type_info je definisana u zagлавju <typeinfo> i sadrži sledeće javne metode:

```

bool operator== (const type_info&) const;
bool operator!= (const type_info&) const;

```

Ove operatorske funkcije ispituju da li dva objekta tipa type_info predstavljaju isti tip ili ne. Obavezno treba upoređivati objekte, a ne njihove adrese, jer nije sigurno da postoji samo jedan objekat koji predstavlja dati tip.

```
bool before (const type_info&) const;
```

Ova metoda upoređuje dva objekta tipa type_info za potrebe uređivanja. Redosled uređivanja zavisi od prevodioca. Logično je očekivati da to bude po abecednom redosledu imena tipova.

```
const char* name () const;
```

Ova metoda vraća pokazivač na naziv tipa (sa završnim znakom '\0') koga predstavlja objekat tipa type_info. Tačan sadržaj teksta zavisi od prevodioca.

Objekti tipa type_info ne mogu da se stvaraju naredbama za definisanje podataka (jer ne postoji nijedan javan konstruktor, i iz istog razloga ne mogu da se prenose u funkcije pomoću vrednosti). Ne mogu ni da se međusobno dodeljuju, jer je i operatorska funkcija za dodelu vrednosti privatna.

Evo primera za dinamičko određivanje tipova:

```

#include <iostream>
#include <typeinfo>
using namespace std;
class Alfa {};
class Beta : public Alfa
{
    virtual void vm () {} ;
};
class Gama : public Beta {};
int main () {
    Alfa* pa = new Beta;
    Beta* pb = new Gama;
    cout << boolalpha;
    cout << (typeid(*pa)==typeid(Beta)) << endl; // false
    cout << (typeid(*pb)==typeid(Gama)) << endl; // true
    cout << typeid(Alfa).name() << endl; // Alfa
    cout << typeid(*pa).name() << endl; // Alfa
    cout << typeid(pa).name() << endl; // Gama
    cout << typeid(pb).name() << endl; // Alfa*
    cout << typeid(*pb).name() << endl; // Beta*
    cout << typeid(Gama).before(typeid(Beta))<<endl; // false
    int m[100][20];
    cout << typeid(m).name() << endl; // int[100][20]
    cout << typeid(m).before(typeid(Alfa)); // false
}

```

Pošto je pa pokazivač na objekte nepolimorfne klase Alfa, typeid(*pa) nije jednak typeid(Beta), bez obzira što pokazivač pa pokazuje na objekat klase Beta koja je izvedena iz klase Alfa. S druge strane, klasa Beta je polimorfna, pa kada pokazivač pb na objekte tipa Beta pokazuje na objekat klase Gama koja je izvedena iz klase Beta, typeid(*pb) je jednak typeid(Gama).

Mada Standard ne obavezuje, logično je očekivati da rezultati metode name() budu jednaki imenima odgovarajućih standardnih tipova, odnosno imenima klasa. U posmatranom primeru navedeni su rezultati dobijeni jednim prevodiocem. Nema razloga da drugim prevodiocem rezultati budu mnogo drugačiji. Eventualne razlike se mogu očekivati za izvedene tipove, kao što su pokazivači i nizovi.

Za rezultate uređivanja metodom before(), što Standard takođe ne propisuje, logično je očekivati uređivanje na osnovu leksikografskog poretku imena tipova. Razlike od prevodioca do prevodioca mogu da se očekuju u poretku izvedenih tipova. Na primer u poretku tipova int, int*, int[5] itd.

5.11 Zadaci

5.11.1 Obrada geometrijskih figura u ravni

Zadatak:

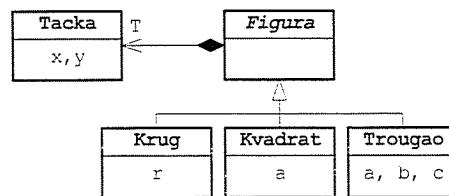
Napisati na jeziku C++ klase za obradu geometrijskih figura u ravni. Napisati na jeziku C++ program za prikazivanje mogućnosti tih klasa.

Rešenje:

Geometrijske figure u ravni su trouglovi, kvadrati, pravougaonici, rombovi, trapezi, mnogouglovi, krugovi itd. Sve figure imaju neke zajedničke parametre kao što su položaj i orijentacija u ravni, boja, površina i obim. U specifične parametre spadaju informacije vezane za pojedine vrste figura. To su, na primer, dužine stranica za trouglove, četvorougue, mnogouglove i poluprečnik za krugove. Što se obima i površine tiče, oni su kao pojam zajednički za sve vrste figura, ali postupak za njihovo određivanje je specifičan za svaku vrstu figura ponašob.

U okviru ovog rešenja zadatka usvojen je relativno jednostavan skup vrsta figura i jednostavan način za njihovo predstavljanje. Od vrsta figura obrađeni su krugovi, kvadrati i trouglovi. Položaj figura u ravni zadaje se koordinatama težišta a za orijentaciju se podrazumeva vodoravna ivica kvadrata, odnosno vodoravna prva ivica trouglova. Krugovi nemaju orijentaciju.

Zajedničke osobine svih vrsta figura predstavlja osnovna klasa Figura (slika 5.7). Jedino polje te klase je težište T. Težište je, kao pojam, tačka za čije predstavljanje se koristi klasa Tacka.



Slika 5.7 – Klase za geometrijske figure u ravni

Iz klase Figura izvedene su tri klase za predstavljanje krugova, kvadrata i trouglova. Klasa Krug sadrži specifično polje r za poluprečnik kruga. U klasi Kvadrat specifično polje je dužina ivice kvadrata a. Na kraju, klasa Trougao sadrži tri dodatna polja a, b i c koja predstavljaju ivice trouglova.

Što se klase Tacka tiče, ona je nezavisna od ostale četiri klase. Služi za predstavljanje tačaka u ravni. Prirodno je da ima dva polja, koordinate x i y. Koristi se kao tip za težište figura.

Za ovaj jednostavan sistem nije bilo neophodno uvođenje zasebne klase za tačke. Ipak je to učinjeno iz principijelnih razloga. Tačka kao geometrijski pojam je neizbežna za posmatranu vrstu problema. Kada bi se krenulo u proširivanje obima obrade figura vrlo brzo bi se pojavili i drugi elementi koji su tačke. Na primer, koordinate temena trouglova.

Program 5.1 prikazuje definiciju klase Tacka.

```

// Definicija klase tačaka (Tacka).

#ifndef _tacka2_h_
#define _tacka2_h_

typedef double Real;                                // Tip za realne vrednosti.

#include <iostream>
using namespace std;

class Tacka {
    Real x, y;                                         // Koordinate.

public:
    Tacka (Real xx=0, Real yy=0) { x = xx; y = yy; }   // Konstruktor.
    Real aps () const { return x; }                      // Apscisa.
    Real ord () const { return y; }                      // Ordinata.
    friend istream& operator>> (istream& ut, Tacka& tt) // Čitanje.
    { return ut >> tt.x >> tt.y; }
    friend ostream& operator<< (ostream& it, const Tacka& tt) // Pisanje.
    { return it << '(' << tt.x << ',' << tt.y << ')'; }
};

const Tacka ORG;                                     // Koordinatni početak.

#endif
  
```

Program 5.1 – Definicija klase tačaka (tacka2.h)

U iole složenijim programskim sistemima, kao što je i rešenje ovog zadatka, može da se desi da isto zaglavje (datoteka .h) bude u toku jednog prevodenja više puta uključeno (direktivom #include) u tekst koji se prevodi. U tom slučaju prevodilac bi video više definicija iste klase, što nije dozvoljeno. Svaka klasa mora i sme da se definiše tačno jednom.

Sprečavanje da se sadržaj istog zaglavlja više puta prevodi postiže se odgovarajućim direktivama preprocessora za uslovno prevodenje. U tom cilju svakoj datoteci treba pridružiti neki identifikator koji se inače ne koristi u programu. Taj identifikator obično se izvodi iz imena datoteke. U posmatranom slučaju datoteci tacka2.h pridružen je identifikator _tacka2_h_. Ukoliko taj identifikator nije definisan (direktiva #ifndef), prevodiće se deo teksta do direktive #endif. Ako je identifikator definisan, preskočiće se taj deo teksta. U uslovno prevodenom delu teksta potrebno je direktivom #define definisati pridruženi identifikator. Na taj način prevodilac će preskočiti deo koji se uslovno prevodi kada u toku istog prevodenja drugi put čita sadržaj te datoteke.

Drugi koristan tehnički detalj u programu 5.1 je uvođenje, naredbom typedef, novog identifikatora Real za obeležavanje tipa numeričkih podataka. U ovom slučaju taj identifikator predstavlja tip double. Na taj način prelazak na drugu vrstu numeričkih podataka postiže se izmenom samo posmatrane naredbe i ponovnim prevodenjem celokupnog programskog sistema. Mogući drugi tipovi su standardni tipovi float i long double. U nekoj drugoj primeni možda bi u obzir došli i celobrojni tipovi ili pak tipovi koje definiše programer, na primer klasa kompleksnih brojeva.

Što se same klase Tacka tiče, sadrži dva polja za koordinate predstavljene tačke i neophodan broj metoda za preostali deo rešenja ovog zadatka.

Jedini konstruktor predstavlja i podrazumevani konstruktor i konstruktor koji se koristi pri automatskoj konverziji. Pored toga, može da stvara tačku na osnovu dve realne koordinate.

Koordinate tačke u sadržaju primeraka klase mogu da se dohvate pomoću funkcija `aps()` za apscisu i `ord()` za ordinatu. Ovde su ostvarene kao metode, mada se u praksi u sličnim situacijama češće koriste prijateljske funkcije.

Za čitanje i pisanje podataka tipa Tacka postoje prijateljske funkcije `operator>>()` i `operator<<()`. One ne mogu da budu metode, jer prvi operand im nije tipa Tacka.

Na kraju programa 5.1 nalazi se definicija simboličke konstante ORG koja predstavlja koordinatni početak. Inicijalizuje se koordinatama (0,0) pomoću podrazumevanog konstruktora klase Tacka.

Pošto su sve metode i prijateljske funkcije klase Tacka ugrađene funkcije, ne postoji datoteka s odvojenim definicijama takvih funkcija (datoteka .C).

Program 5.2 prikazuje definiciju apstraktne osnovne klase Figura.

```
// Definicija klase geometrijskih figura (Figura).
#ifndef _figural_h_
#define _figural_h_

#include "tacka2.h"

class Figura {
    Tacka T; // Težiste figure.
public:
    Figura (const Tacka& tt=ORG): T(tt) {} // Konstruktor.
    virtual ~Figura () {} // Destruktor.
    virtual Figura* kopija () const =0; // Stvaranje kopije.
    Figura& postavi (Real xx, Real yy) // Postavljanje figure.
        { T = Tacka (xx, yy); return *this; }
    Figura& pomeri (Real dx, Real dy) // Pomeranje figure.
        { T = Tacka (T.aps() + dx, T.ord() + dy); return *this; }
    virtual Real O () const =0; // Obim.
    virtual Real P () const =0; // Površina.
protected:
    virtual void citaj (istream& ut) { ut >> T; } // Čitanje.
    virtual void pisi (ostream& it) const { it << "T=" << T; } // Pisanje.
    friend istream& operator>> (istream& ut, Figura& ff) // Uopšteno čitanje
        { ff.citaj (ut); return ut; }
    friend ostream& operator<< (ostream& it, const Figura& ff) // pisanje.
        { ff.pisi (it); return it; }
};

#endif
```

Program 5.2 – Definicija klase geometrijskih figura (figural.h)

Postoji samo jedno polje T tipa Tacka koje predstavlja težiste geometrijske figure. Polje je proglašen privatnim jer izvedene klase nemaju potrebe da ga koriste neposredno.

Klasa Figura je apstraktna jer sadrži i apstraktne metode. To znači da ne mogu da se stvaraju primerci te klase. Uprkos tome, postoji konstruktor s jednim podrazumevanim parametrom tipa Tacka. On može da se koristi u izvedenim klasama za inicijalizaciju nasleđenog podobjekta.

Destruktor klase Figura ima prazno telo, jer nema šta da se radi prilikom uništavanja primeraka te klase. Za obične klase ne bi morao da se definiše, jer bi se ionako izgenerisao automatski. Ovde je definisan samo zato da bi bio virtuelan (dodatak je modifikator `virtual` na početku definicije). Možda će neka od izvedenih klasa koristiti dinamičko dodeljivanje memorije i imati destruktora za oslobođenje te memorije. Ako je destruktorni virtuelan, prilikom uništavanja bilo koje vrste figure pomoću pokazivača ili upućivača na osnovnu klasu Figura, odabrat će destruktora iz one izvedene klase kojoj pripada konkretna figura. Automatski generisani destruktorski ne bi bio virtuelan, što bi moglo da predstavlja problem.

Pored konstruktora i destruktora, vrlo je korisno da se u bilo kojoj osnovnoj klasi (bez obzira da li je apstraktna ili ne) predviđa virtuelna metoda za stvaranje kopije tekućeg objekta u dinamičkoj zoni memorije. Takva metoda će omogućiti opremanje klasa za zbirke (liste, stekovi, redovi, nizovi itd.) objekata tipa te osnovne klase i njenih potomaka konstruktorskom kopijom i operatom `=`. U posmatranom slučaju radi se o metodi `kopija()` koja vraća pokazivač na stvorenu kopiju (Figura*). Pošto je klasa Figura apstraktna zbog kasnije objašnjениh apstraktnih metoda, i ova metoda mora da bude apstraktna. To je označeno sa `=0`; umesto tela metode.

Od metoda koje su specifične za problem geometrijskih figura prve služe za podešavanje položaja figura u ravni. Parametri metode `postavi()` su nove koordinate težišta figure, bez obzira na prethodni položaj u ravni. Za razliku od toga, parametri metode `pomeri()` su promene koordinata u odnosu na prethodni položaj. Vrednost obe metode je upućivač na tekući objekat (Figura*) u promjenjenom stanju, što omogućava lančanje operacija nad figurama. Vrednost metoda ne može da bude tipa Figura, jer je klasa apstraktna! Pošto način izvođenja ovih radnji ne zavisi od vrste figure na koju se primeni, posmatrane metode nisu virtuelne. Mogu da se koriste za pomeranje svih vrsta figura. U izvedenim klasama ne bi trebalo da se ove metode redefinišu. Nažalost, u jeziku C++ to ne može da se zabrani. U svakom slučaju, eventualno redefinisana metoda u izvedenoj klasi sakrivalo bi metodu iz osnovne klase s vrlo problematičnim posledicama.

Naredne dve metode određuju obim (O()) i površinu (P()) figure. Bez obzira što su pojmovi zajednički za sve vrste figura, postupak kojim se dolazi do njihovih vrednosti je različit za svaku vrstu figura. Zato će svaka izvedena klasa morati da definiše odgovarajuće metode za ta izračunavanja. S druge strane, pošto bez poznavanja vrste figura ovi parametri ne mogu nikako da se odrede, odgovarajuće metode nisu ni definisane u osnovnoj klasi Figura, tj. radi se o apstraktnim metodama (`=0`; umesto tela). Zbog toga je klasa Figura apstraktna klasa.

Na kraju, svaka klasa često ima mogućnosti za postavljanje stanja objekata čitanjem podataka iz nekih tekstualnih ulaznih tokova i za prikazivanje sadržaja objekata pisanjem u neke tekstualne izlazne tokove. Pošto u posmatranom slučaju ostali podaci, izuzev koordinata težišta figure, zavise od vrste figure, čitanje i pisanje figura mora da se ostvaruju virtuelnim metodama.

Bilo bi interesantno da se čitanje i pisanje podataka o figurama ostvaruje uobičajenim preklapanjem operatora `>>` i `<<`. Međutim, operatorske funkcije `operator>>()` i `operator<<()`

`tor<<()` moraju da budu prijateljske funkcije (prvi operand im nije tipa klase za koju se prave), a prijateljske funkcije ne mogu da budu virtuelne.

Problem je u programu 5.2 prevaziđen uvedenjem „međufunkcija” `citaj()` i `pisi()` koje su virtuelne metode. Ne očekuje se, međutim, da ih korisnik poziva neposredno. Umesto toga treba da koristi preklopljene operatore `>>` i `<<`. Odgovarajuće operatorske funkcije `operator>>()` i `operator<<()` sadrže samo poziv metode `citaj()` odnosno `pisi()`. Pošto su tako jednostavne, predviđene su da budu ugradene funkcije.

Da bi se sprečilo da korisnik neposredno poziva metode `citaj()` i `pisi()`, one su proglašene zaštićenim metodama stavljanjem oznake `protected`: pre njihove definicije. Time je omogućeno da se iz izvedenih klasa mogu pozivati neposredno, ali ne i iz klasa koje nisu potomci klase `Figura`. Skreće se pažnja na to da prijateljske funkcije nisu članovi klase čiji su prijatelji, pa nije bitno u kom delu, privatnom, zaštićenom ili javnom, se „proglašavaju” prijateljskim. Eventualno ograničavanje njihovog korišćenja može da nametne samo njihova matična klasa, ako su one istovremeno i metode neke klase.

Što se samih metoda `citaj()` i `pisi()` tiče, predviđeno je da pročita odnosno ispišu koordinate težišta tekuće figure. Naravno, to se izvodi automatskim pozivanjem operatorskih funkcija `operator>>()` i `operator<<()` iz klase `Tacka` (`ut>>T`, odnosno `it<<T`). Međutim ovo je samo jedan deo posla, ali više u klasi `Figura` ne može da se uradi. Koje ostale parametre treba čitati ili pisati znaće se samo unutar izvedenih klasa kojima pripadaju konkretnе figure.

Program 5.3 prikazuje definiciju klase `Krug`.

```
// Definicija klase krugova (Krug).

#ifndef _krug2_h_
#define _krug2_h_

#include "figural.h"

namespace { const Real PI = 3.14159265359; }

class Krug : public Figura {
    Real r;                                // Poluprečnik.
public:
    Krug (Real rr=1, const Tacka& tt=ORG)   // Konstruktor.
        : Figura (tt) { r = rr; }
    Krug* kopija () const                  // Stvaranje kopije.
        { return new Krug (*this); }
    Real O () const { return 2 * r * PI; }    // Obim.
    Real P () const { return r * r * PI; }    // Površina.
private:
    void citaj (istream& ut);              // Čitanje.
    void pisi (ostream& it) const;          // Pisanje.
};

#endif
```

Program 5.3 – Definicija klase krugova (krug2.h)

Pre svega, definisana je simbolička konstanta `PI` tipa `Real` za predstavljanje vrednosti π . Da bi identifikator `PI` imao unutrašnje povezivanje njegova definicija je stavljena u bezimeni prostor imena. Da identifikator `Real` predstavlja neki tip, definisano je u zaglavlju `tacka.h`. Pošto se u datoteci `figura.h` nalazi direktiva `#include "tacka.h"`, a u posmatranoj datoteci `#include "figura.h"`, ta definicija i sada стоји na raspolaganju prevodiocu.

Nije tipično oslanjati se na ovakvo „nasleđivanje” definicija i deklaracija u više koraka. Situacija je mnogo jasnija ako se na licu mesta navode direktive `#include` za sva zaglavila iz kojih se nešto koristi. To bi za slučaj programa 5.3 značilo još dve direktive, za datoteke `tacka.h` i za standardno zaglavje `<iostream>`. Da to ne dovede do konfliktne situacije, sprečava se već pomenutim uslovnim prevodenjem sadržaja takvih zaglavila (`#ifndef ... #endif`).

Što se same klase `Krug` tiče, javno je izvedena iz klase `Figura`. Kao specifične elemente poseduje polje `r` za poluprečnik kruga i konstruktor za stvaranje objekata tipa `Krug`.

Navedeni konstruktor, pošto svi parametri imaju podrazumevane vrednosti, može da se koristi i kao podrazumevani konstruktor i kao konverzija iz tipa `Real` u tip `Krug`. Kao podrazumevani konstruktor stvara krug poluprečnika `1` s centrom u koordinatnom početku. Rezultat konverzije je krug navedenog poluprečnika s centrom u koordinatnom početku. Objekat se stvara kombinacijom inicijalizacije nasledenog podobjekta (`Figura (tt)`) i dodelje vrednosti polju standardnog tipa (`r=r`). Za polja standardnih tipova nema posebnu prednost korišćenje inicijalizatora pre tela umesto dodelje vrednosti unutar tela konstruktora.

Pošto se ne koristi dinamička zona memorije ili neke druge specifičnosti (na primer brojač svih objekata kao zajednički član), destruktor nije potreban.

Metoda `kopija()` je jednostavna i ima isti oblik za sve klase. Memorija se dodeljuje operatorom `new`, a dodeljeni prostor se inicijalizuje (u ovom slučaju automatski generisanim) konstruktorom kopije (`Krug (*this)`, gde je `*this` tekući objekat). Potreba za uvođenjem metode `kopija()` javlja se zato što se novi objekti inicijalizuju konstruktorima, a konstruktori ne mogu biti virtuelni. Virtuelna metoda `kopija()` omogućava da se, kada je potrebno polimorfno stvaranje kopije objekta nekog tipa, mehanizmom pozivanja virtuelnih metoda pozove metodu `kopija()` odgovarajuće (izvedene) klase koja će konstruktorom sopstvene klase da inicijalizuje stvorenu kopiju. Skreće se pažnja na to da je tip vrednosti metode `kopija()` `Krug*`, iako je tip vrednosti istoimene virtuelne metode `Figura*`. To je dozvoljeno kada je tip vrednosti metode pokazivač ili upućivač na sopstvenu klasu. Inače tip vrednosti virtuelne metode u svim izvedenim klasama mora da bude jednak tipu vrednosti istoimene metode u osnovnoj klasi.

Specifične virtuelne metode za izračunavanje obima i površine kruga su krajnje jednostavne, pa su predviđene da budu ugradene metode. One zamenuju odgovarajuće apstraktne metode osnovne klase `Figura`.

Preostale dve virtuelne metode za čitanje i pisanje podataka o krugovima su nešto složenije pa su odvojeno definisane u programu 5.4. One nisu ugradene metode.

Čitanje podrazumeva čitanje koordinata centra (težišta) `T` i poluprečnika `r`. Težište `T` je privatno polje osnovne klase, pa ne može da mu se pristupa neposredno. Zbog toga za čitanje koordinata treba pozivati zaštićenu metodu `citaj()` iz osnovne klase. Pošto je ta metoda sakrivena od strane istoimene metode u klasi `Krug`, mora se operatom za razrešenje dosega `(::)` naznačiti da se želi pozvati metoda iz klase `T::citaj()`.

Definicije metoda uz klasu Krug.

```
#include "krug2.h"

void Krug::citaj (istream& ut)          // Čitanje.
{ Figura::citaj (ut); ut >> r; }

void Krug::pisi (ostream& it) const {    // Pisanje.
    it << "krug ";
    Figura::pisi (it);
    it << ", r=" << r << ", O=" << O() << ", P=" << P() << ')';
}
```

Program 5.4 – Definicije metoda uz klasu Krug (krug2.C)

(Figura::citaj(ut)). Treba još naglasiti da na ovom mestu mehanizam virtualnih metoda više ne dejstvuje. Pozvaće se metoda citaj() iz klase Figura, bez obzira što je ona virtualna i bez obzira što je tekući objekat tipa Krug.

Pisanje podataka o krugovima napravljeno je malo specifično da bi se, na kraju ovog izlaganja, omogućilo što jasnije prikazivanje funkcionalnosti celokupnog sistema opisanih klasa. Na početku se ispisuje reč krug, da bi čitaocu ispisa bilo jasno o čemu se radi. Iza toga, unutar para uglastih zagrada [], prikazuju se podaci o krugu; kako vrednosti polja, tako i rezultati postojećih metoda. Koordinate centra, slično čitanju, ispisuju se metodom pisi() iz osnovne klase (Figura::pisi(it)).

Ostvarenje klase kvadrata je vrlo slično ostvarenju klase krugova. Program 5.5 prikazuje definiciju same klase Kvadrat, a program 5.6 definicije pratećih metoda.

```
// Definicija klase kvadrata (Kvadrat).

#ifndef _kvadrat_h_
#define _kvadrat_h_

#include "figural.h"

class Kvadrat : public Figura {
    Real a;                                // Osnovica.
public:
    Kvadrat (Real aa=1, const Tacka& tt=ORG) // Konstruktor.
        : Figura(tt) { a = aa; }
    Kvadrat* kopija () const                // Stvaranje kopije.
        { return new Kvadrat (*this); }
    Real O () const { return 4 * a; }         // Obim.
    Real P () const { return a * a; }         // Površina.
private:
    void citaj (istream& ut);               // Čitanje.
    void pisi (ostream& it) const;           // Pisanje.
};

#endif
```

Program 5.5 – Definicija klase kvadrata (kvadrat.h)

Definicije metoda uz klasu Kvadrat.

```
#include "kvadrat.h"

void Kvadrat::citaj (istream& ut)          // Čitanje.
{ Figura::citaj (ut); ut >> a; }

void Kvadrat::pisi (ostream& it) const {    // Pisanje.
    it << "kvadrat ";
    Figura::pisi (it);
    it << ", a=" << a << ", O=" << O() << ", P=" << P() << ')';
}
```

Program 5.6 – Definicije metoda uz klasu Kvadrat (kvadrat.C)

Klase trouglova je nešto složenija od klase krugova i kvadrata jer su za trouglove potrebna tri specifična podatka, dužine tri stranice trougla. Program 5.7 prikazuje definiciju klase Trougao.

```
// Definicija klase trouglova (Trougao).

#ifndef _trougao1_h_
#define _trougao1_h_

#include "figural.h"

class Trougao : public Figura {
    Real a, b, c;                          // Stranice.
public:
    Trougao (Real aa=1, const Tacka& tt=ORG) // jednakostranični
        : Figura(tt) { a = b = c = aa; }
    Trougao (Real aa, Real bb, const Tacka& tt=ORG) // jednakokraki
        : Figura(tt) { a = aa; b = bb; c = cc; }
    Trougao (Real aa, Real bb, Real cc, const Tacka& tt=ORG) // opšti
        : Figura(tt) { a = aa; b = bb; c = cc; }
    Trougao* kopija () const               // Stvaranje kopije.
        { return new Trougao (*this); }
    Real O () const { return a + b + c; }   // Obim.
    Real P () const;                      // Površina.
private:
    void citaj (istream& ut);             // Čitanje.
    void pisi (ostream& it) const;          // Pisanje.
};

#endif
```

Program 5.7 – Definicija klase trouglova (trougao1.h)

Glavna razlika je u postojanju tri konstruktora. Konstruktor s dva parametra stvara jednakostranični trougao sa stranama zadate dužine. Pošto oba parametra imaju podrazumevane vrednosti (1 za dužine stranica i koordinatni početak za težište), može da se koristi i kao podrazumevani konstruktor i za automatsku konverziju tipa. Konstruktor s tri parametra stvara jednakokraki trougao, a s četiri parametra opšti trougao.

Ovoliki broj konstruktora je bio potreban da bi se omogućilo da prvi parametri budu dužine stranica. Poslednji parametar, koji može da ima podrazumevanu vrednost, uvek je težiste trougla. Nekako se činilo prirodnije da ako se nešto podrazumeva, neka to bude položaj težišta, a ne dužina jedne ili više stranica.

Razlike manjeg značaja između trouglova i ostalih vrsta figura jesu u tome što su formule za izračunavanje nekih od parametara trouglova složenije. Zato, odgovarajuće metode nisu predviđene da budu ugradene metode.

Program 5.8 prikazuje definicije metoda koje idu uz klasu Trougao.

```
// Definicije metoda uz klasu Trougao.

#include "trougao.h"
#include <cmath>
using namespace std;

Real Trougao::P () const { // Površina.
    Real s = (a + b + c) / 2;
    return sqrt (s * (s-a) * (s-b) * (s-c));
}

void Trougao::citaj (istream& ut) // Čitanje.
{ Figura::citaj (ut); ut >> a >> b >> c; }

void Trougao::pisi (ostream& it) const { // Pisanje.
    it << "trougao [";
    Figura::pisi (it);
    it << ", a=" << a << ", b=" << b << ", c=" << c
        << ", O=" << O() << ", P=" << P() << ']';
}
```

Program 5.8 – Definicije funkcija uz klasu Trougao (trougao1.C)

Na kraju, program 5.9 služi za ispitivanje gore opisanih klasa geometrijskih figura. Kao primer, stvara listu geometrijskih figura.

Elementi liste su strukture (**struct**) s dva polja: pokazivač na figuru i pokazivač na sledeći element u listi. Upotreba pokazivača na figure je neophodna da bi u listu istovremeno mogle da se stave figure različitih vrsta. Naravno, potreban je pokazivač na osnovnu klasu, dakle tip **Figura***. Ako bi se pravila lista koja sadrži samo, na primer, krugove, elementi liste mogli bi da sadrže objekte (a ne pokazivače) tipa Krug i pokazivač na sledeći element liste. Pošto u jeziku C++ i strukture mogu imati metode, u strukturi **Elem** napravljeni su jedan konstruktor i destruktur. Pomoću njih rad s elementima liste je udobniji. Ova struktura predstavlja primer lokalne klase (strukture).

Na početku svakog prolaska kroz ciklus za stvaranje liste pročita se jedan znak koji treba da označi vrstu figure za koju treba citati podatke u nastavku. Skreće se pažnjava da se i pojedinačni znakovi čitaju uz prethodno preskakanje eventualnih belih znakova.

Zavisno od toga da li je pročitano slovo o, k ili t, operatorom **new** se dodeljuje prostor u dinamičkoj zoni memorije za jedan objekat tipa Krug, Kvadrat odnosno Trougao. Rezultujući pokazivač se u sva tri slučaja dodeljuje istom pokazivaču pf na objekte osnovne klase. Ako pročitani znak nije nijedno od pomenutih slova, pomoću **dalje=false** signalizira se da nema više podataka za čitanje.

```
// Program za ispitivanje klasa za geometrijske figure.

#include "krug2.h"
#include "kvadrat.h"
#include "trougao.h"
#include <iostream>
using namespace std;

int main () {

    // Element liste figura.
    struct Elem {
        Figura* fig; Elem* sled;
        Elem (Figura* ff, Elem* ss=0) { fig = ff; sled = ss; }
        ~Elem () { delete fig; }
    };

    Elem *prvi = 0, *posl = 0;

    // Stvaranje liste figura čitajući s glavnog ulaza.
    for (bool dalje=true; dalje; ) {
        Figura* pf;
        char vrsta; cin >> vrsta;
        switch (vrsta) {
            case 'o': pf = new Krug; break;
            case 'k': pf = new Kvadrat; break;
            case 't': pf = new Trougao; break;
            default: dalje = false;
        }
        if (dalje) {
            cin >> *pf;
            Elem* novi = new Elem (pf);
            posl = (!prvi ? prvi : posl->sled) = novi;
        }
    }

    // Prikazivanje sadržaja liste na glavnom izlazu.
    for (Elem* tek=prvi; tek; tek=tek->sled) cout << *tek->fig << endl;

    // Čitanje pomeraja za pomeranje figura.
    Real dx, dy; cin >> dx >> dy;
    cout << "\ndx, dy= " << dx << ", " << dy << "\n\n";

    // Stvaranje kopije liste uz pomeranje figura.
    Elem *poc = 0, *kraj = 0;
    for (tek=prvi; tek; tek=tek->sled) {
        Elem* novi = new Elem (tek->fig->kopija());
        novi->fig->pomeri(dx,dy);
        kraj = (!poc ? poc : kraj->sled) = novi;
    }

    // Uništavanje prve liste.
    while (prvi) { Elem* stari=prvi; prvi=prvi->sled; delete stari; }
    posl = 0;

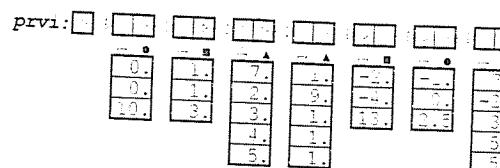
    // Prikazivanje sadržaja kopirane liste na glavnom izlazu.
    for (tek=poc; tek; tek=tek->sled) cout << *tek->fig << endl;
```

Program 5.9 – Obrazovanje liste geometrijskih figura (figurait.C)

Ako ima još podataka, izrazom `cin>>*pf` pročitaju se podaci za odgovarajuću figuru. Tu se, u prvom koraku, poziva funkcija `operator>>()` koja je definisana uz osnovnu klasu `Figura`. Mehanizam virtualnih metoda će se pobrinuti da izraz `ff.citaj()` unutar te operatorske funkcije pozove funkciju za čitanje koja odgovara tipu tekućeg objekta.

Posle toga dodeljuje se prostor u dinamičkoj zoni memorije za novi element liste figura uz popunjavanje pomoću konstruktora strukture `Elem`. Zatim se novi element jednom na-ređbom dodaje na kraj liste na način koji je objašnjen u zadatku 4.11.3.

Slika 5.8 prikazuje kako može da izgleda lista geometrijskih figura po izlasku iz ciklusa za njeno obrazovanje. Kao što se vidi, elementi liste pokazuju na objekte različitih, ali naravno srodnih, tipova.



Slika 5.8 – Lista geometrijskih figura

U nastavku programa 5.9 sledi ciklus za prikazivanje sadržaja obrazovane liste. Ciklus je krajnje jednostavan. Odgovarajuć funkcija za prikaz pojedinih vrsta objekata bira se automatski mehanizmom virtualnih metoda.

Na početku drugog dela programa 5.9, posle čitanja željene promene koordinata `dx` i `dy`, stvara se kopija prve liste uz istovremeno pomeranje svih figura za pročitanu udaljenost. U novi element druge liste stavljaju se pokazivač na kopiju figure koja se nalazi u tekućem elementu prve liste (`tek->fig->kopija()`). Zavisno od vrste figure, mehanički figura u novom elementu menja obična (ne virtuelna) metoda iz klase `Figura`, ne-zavisno od tipa pomerane figure (`novi->fig->pomeri(dx, dy)`).

Po završetku kopiranja sledi uništavanje prve liste. Skreće se pažnja na to da se prilikom oslobođanja memorije u dinamičkoj zoni, koju zauzima tekući element liste (`delete stari`), poziva destruktur strukture `Elem`. Taj destruktur će da oslobođi memoriju (`delete fig`) koju zauzima figura koja je sadržana u elementu liste koji se uništava.

Program 5.9 se završava ispisivanjem sadržaja pomerene kopije početne liste.
Rezultat 5.1 prikazuje primer rada programa 5.9.

5.11.2 Obrada zbirki geometrijskih figura u ravni △

Zadatak:

Napisati na jeziku C++ apstraktnu klasu za obradu zbirki geometrijskih figura s pristupom elementima po proizvolnjem redosledu i iz nje izvedene klase za nizove i liste geometrijskih figura u ravni. Napisati na jeziku C++ apstraktne klase za iteratore koji omogućavaju obilazak uzastopnih elemenata apstraktnih zbirki i iz nje izvedene klase za iteratore za nizove i za liste. Napisati na jeziku C++ program za prikazivanje mogućnosti tih klasa.

```
CC figuralt.C krug2.C kvadrat.C trougaol.C -o figuralt
figuralt <figuralt.pod
```

```
krug {T=(0,0), r=10, O=62.8319, P=314.159}
kvadrat {T=(1,1), a=3, O=12, P=9}
trougaol {T=(7,2), a=3, b=4, C=5, O=12, P=6}
trougaol {T=(1,3), a=1, b=1, C=1, O=3, P=0.433013}
kvadrat {T=(-2,-4), a=33, O=132, P=1089}
krug {T=(-2,0), r=2.5, O=15.708, P=19.635}
trougaol {T=(7,-2), a=3, b=5, C=5, O=13, P=7.15454,
```

```
xx, dy = 3, 5
```

```
krug {T=(3,5), r=10, O=62.8319, P=314.159}
kvadrat {T=(4,6), a=3, O=12, P=9}
trougaol {T=(10,7), a=3, b=4, C=5, O=12, P=6}
trougaol {T=(4,14), a=1, b=1, C=1, O=3, P=0.433013}
kvadrat {T=(1,1), a=33, O=132, P=1089}
krug {T=(2,5), r=2.5, O=15.708, P=19.635}
trougaol {T=(10,3), a=3, b=5, C=5, O=13, P=7.15454}
```

Rezultat 5.1 – Obrada geometrijskih figura programom 5.9

Rešenje:

Zbirke podataka mogu da budu skupovi, redovi, stekovi, uređeni nizovi (u smislu da je redosled u nizu važan) itd. Za uskladištanje elemenata zbirke mogu da se koriste nizovi, liste ili neke druge strukture podataka. Osnovne radnje sa zbirkama, pored stvaranja i uništavanja, mogu da budu dodavanje novog elementa u zbirku, vodenje elementa iz zbirke, praznjenje zbirke, prikazivanje sadržaja zbirke itd.

Elementima zbirke može da se pristupa po proizvolnjem redosledu, na osnovu njihovih rednih brojeva ili redom, kako se oni nalaze u zbirci. Uzimanje uzastopnih elemenata zbirke naziva se iteracija, a sredstva koja to omogućavaju **iteratori**. Iteratorske operacije mogu da budu ugrađene u klase za zbirke, ali mogu da se ostvaruju i u obliku zasebnih klasa. Iteratori kao zasebne klase su interesantni, jer omogućavaju da više obilazaka elemenata iste zbirke budu istovremeno u toku. Svaki objekat iteratora prati odvijanje jednog obilaska.

Program 5.10 prikazuje definiciju apstraktne klase za zbirke geometrijskih figura. Sadržaj zbirke čine objekti tipa `Figura` iz zadatka 5.11.1 (tačnije tipa klasa koje su izvedene iz klase `Figura`, pošto je ona apstraktna klasa). Pošto je `Figura` apstraktna klasa stvarni sadržaj zbirki figura čine pokazivači na figure.

Pošto se klasa `Iter` u nastavku pomije samo kao pokazivački ili upućivački parametar ili vrednost funkcija, dovoljno je da bude samo deklarisana. Potpuna definicija klase je potrebna samo ako se stvaraju objekti tipa te klase ili ako se pristupa njenim članovima.

U klasi `Zbirka` ništa nije rečeno kako se zbirke fizički ostvaruju, već su samo navedene operacije koje treba da ostvaruju buduće izvedene klase za konkretnе vrste zbirki. Zbog toga većina metoda je apstraktna. Radi udobnijeg rada sa zbirkama korišćene su operatorske funkcije, gde god se to činilo primerenim.

Na prvom mestu je virtualni destruktur. Njegova uloga je da omogući polimorfno uništavanje objekata tipa konkretnih zbirki pomoću pokazivača na klasu `Zbirka`.

Unarni operator + je iskorišćen za dobijanje informacije o broju elemenata zbirke.

```
// Definicija apstraktne klase za zbirke figura (Zbirka).
#ifndef _zbirka_h_
#define _zbirka_h_

#include <iostream>
using namespace std;

#include "figural.h"
class Iter;

class Zbirka {
public:
    virtual ~Zbirka () {} // Virtuelni destruktor.
    virtual int operator+ () const =0; // Broj elemenata zbirke.
    virtual Zbirka& operator+=(Figura* fig)=0; // Dodavanje figure.
    friend Zbirka& operator+= (Zbirka& zb, const Figura& fig)
        { return zb += fig.kopija (); }
    virtual Figura*& operator[] (int ind) =0; // Dohvatanje figure.
    virtual const Figura* operator[] (int ind) const =0;
    virtual Zbirka& operator~ () =0; // Pražnjenje zbirke.
    virtual Zbirka* kopija () const =0; // Stvaranje kopije zbirke.
    friend ostream& operator<< (ostream& it, const Zbirka& zb):
        {
            virtual Iter* iter () =0; // Stvaranje iteratora.
            virtual const Iter* iter () const =0;

            enum { G_IND, G_TEK }; // Šifre grešaka.
            static void greska (int g); // Prekid programa.
        };
#endif

```

Program 5.10 – Definicija apstraktne klase za zbirke figura (zbirka.h)

Binarni operator `+=` je predviđen za dodavanje figura zbirci (po, zasad, nedefinisanom postupku) na dva načina. Da li će ovom operacijom svaki put da se promeni broj elemenata zbirke zavisi od vrste zbirke (na primer, u slučaju skupa zbirka ne sme da sadrži elemente jednakih vrednosti). Operatorska funkcija `operator+=(Figura*)` samo stavlja pokazivač na figuru u zbirku, koji mora da pokazuje na figuru u dinamičkoj zoni memorije. Metoda je apstraktna, jer način umetanja pokazivača zavisi od konkretnе vrste zbirki. Primenom ove operacije pokazivana figura prelazi u isključivu nadležnost zbirke, i korisnik zbirke ne sme više da je koristi, naročito ne da je uništi. Eventualnim kasnijim menjanjem sadržaja figure od strane korisnika došlo bi samo do posrednog menjanja sadržaja zbirke, ali kasnijim uništavanjem figure oštetila bi se zbirka, jer bi sadržala pokazivač na nešto što više ne postoji.

Prijateljska funkcija `operator+=(Zbirka&, const Figura&)` je predviđena za stavljanje kopije figure u zbirku, pa originalna figura i dalje ostaje u nadležnosti korisnika zbirke. Ova operacija može da se ostvari već u klasi `Zbirka` pozivanjem prve varijante operatora `+=` (u izrazu `zb+=fig.kopija()`) desni operand je tipa `Figura*` – videti program 5.2 u rešenju zadatka 5.11.1). Korišćena je prijateljska funkcija, a ne metoda, da

bi mogla da se koristi i u izrazima oblika `niz+=fig`, gde je `niz` objekat tipa `Niz`, a `Niz` je klasa izvedena iz klase `Zbirka`. U ovakovom izrazu je neophodna primena automatske konverzije iz tipa `Niz` u `Zbirka&`, koja je dozvoljena, ali se ne primenjuje automatski na tekuće objekte metoda. Ovu situaciju treba razlikovati od slučaja izraza oblika `zb+=fig`, gde je `zb` upućivač na zbirku (`Zbirka&`) koji upućuje na objekat tipa `Niz`. Pošto je statički tip levog operanda `Zbirka&`, konverzija tipa nije potrebna i zbog toga ne bi smetala ni da je druga varijanta operatora `+=` ostvarena metodom.

Prirodan izbor za dohvatanje figure (tačnije pokazivača na figuru) s određenim rednim brojem bio je preklapanje operatora `[]`. Operacija nije zamišljena tako da odgovaraajuću figuru izbaci iz zbirke, već samo da omogući pristup do nje. Pomoću dobijenog pokazivača može da se pristupa članovima (javnim poljima i metodama) figure i time da se figura obrađuje. Ako korisnik zbirke želi da preuzme figuru u svoju isključivu nadležnost, mora da smesti nulu ili pokazivač na neku drugu figuru na njeno mesto u zbirci. U svakom slučaju, broj elemenata zbirke ne može da se promeni. Sam operator `[]` ne menja ništa u zbirci, ali može posredno da omogući menjanje sadržaja odabrane figure, i time i cele zbirke. Da bi se to sprečilo za nepromenljive zbirke postoje dve varijante metode `operator[]`. Prva varijanta, za koju nije rečeno da ne menja vrednost tekućeg objekta, pozivaće se za promenljive zbirke. Rezultat je upućivač na pokazivač na promenljive figure. To omogućava da se promeni sadržaj odabrane figure, a i sam pokazivač na figuru unutar zbirke može da se promeni (zamenjujući time u zbirci jednu figuru drugom). Druga varijanta operatorske funkcije `[]`, pošto je rečeno da ne menja vrednost svog tekućeg objekta (`const`), pozivaće se za nepromenljive zbirke. Vrednost metode je kopija odgovarajućeg pokazivača iz zbirke kao pokazivač na nepromenljivu figuru (`const Figura*`). Zbog toga neće moći da se promeni vrednost pokazane figure. Ako se promeni vrednost pokazivača, menjajuće se kopija pokazivača iz zbirke, pa se sama zbirka ne menja.

Naredne tri operacije su uslužnog karaktera. Unarni operator `~` je predviđen za pražnjenje zbirke, metoda `kopija()` za polimorfno kopiranje zbirki, a operatorska funkcija za operator `<<` treba da ispisuje sadržaj zbirke. Prve dve radnje ne mogu da se ostvaruju bez poznavanja strukture zbirke, pa su odgovarajuće metode apstraktne. Ispisivanje sadržaja zbirke može da se ostvari pomoću već opisanih operacija, što je i učinjeno u ovom rešenju. Da je to prepusteno klasama za konkretnе zbirke, u nekim od njih bi moglo, možda, da se nađe i efikasnije rešenje.

Za potrebe polimorfног stvaranja iteratora, koji odgovara konkretnоj vrsti zbirke, postoje dve varijante metode `iter()`. Prva će se pozivati za promenljive zbirke i stvaraće iteratore pomoću kojih će moći da se promeni sadržaj zbirke u toku njenog sekvencijalnog obilaska. Druga varijanta će da se poziva za nepromenljive zbirke i stvaraće takve iteratore koji će omogućiti samo uzimanje podataka iz zbirke, a ne i da se bilo šta promeni u njoj.

Na kraju, za rukovanje greškama postoji zajednička (`static`) metoda `greska()`, koja po ispisivanju odgovarajuće poruke, prekida program. U trenutnom rešenju ovog zadatka uočene su dve moguće greške kojima su pridružene simboličke konstante u obliku nabranja (`enum`). Jedna moguća greška je nedozvoljeni indeks prilikom pristupanja elementima po proizvoljnom redosledu (`G_IND`), a druga je nepostojanje tekućeg elementa u toku sekvencijalnog obilaska zbirke iteratorima (`G_TEK`).

Program 5.11 prikazuje definicije metoda i prijateljskih funkcija klase `Zbirka` koje nisu apstraktne i nisu ugrađene.

Definicije metoda u klasu Zbirka.

```
#include "zbirka.h"
#include <iostream>
#include <csdlib>
using namespace std;

ostream& operator<< (ostream& it, const Zbirka& zp) { // Pisanje zbirke.
    it << typeid(zp).name() << ':' << endl;
    for (int i=0; i < +zb; it << " " << *zp[i++] << endl);
    return it << '}' << endl;

void Zbirka::greska (int g) { // Prekid programa.
    const char* const poruke[] = {
        "Nedozvoljeni indeks elementa zbirke!",
        "Ne postoji tekuci element zbirke!",
    };
    cout << "GRESKA: " << poruke[g] << endl;
    exit (g+1);
}
```

Program 5.11 – Definicije metoda u klasu Zbirka (zbirka.C)

Operatorom << prvo se ispisuje ime klase zbirke i zatim, između para zagrada {} ispisuju podaci o pojedinim figurama, svaka figura u zasebnom redu. Ime klase se dobija metodom name() objekta tipa type_info koji se dobija operatorom typeid. Broj figura u zbirci se dobija unarnim operatorom +. Pojedine figure se dohvataju operatorom za indeksiranje [], koji vraća pokazivač na figuru. Za ispisivanje podataka o figurama poziva se operatorska funkcija << (priateljska funkcija klase Figura) s parametrom tipa Figura&, u okviru koje se polimorfno poziva virtualna metoda pisi() iz one klase kojoj pripada trenutna figura.

Metoda greska() sadrži niz poruka o greškama. Parametar g, koji predstavlja šifru greške, koristi se kao indeks za odabiranje poruke za ispisivanje. Po ispisivanju poruke program se prekida bibliotečkom funkcijom exit(). Izraz za završni status je g+1, da nikada ne bi imao vrednost nula, pošto se nula kao završni status smatra uspešnim završetkom programa.

U programu 5.10 predviđeno je samo dohvatanje pojedinih figura po proizvolnjem redosledu indeksiranjem. Za potrebe sekvenčnog obilaska elemenata zbirke, s mogućnošću intervencije na mestu tekućeg elementa, napravljena je zasebna apstraktna klasa Iter (program 5.12).

Kao i u klasi Zbirka, i u klasi Iter većina metoda je apstraktna. Jedan od izuzetaka je virtualni destruktor, čije postojanje je preporučljivo u svim osnovnim klasama. Možda će iteratori obično imaju vrlo jednostavne strukture podataka, malo je verovatno da će to i da se desi u bilo kojoj klasi izvedenoj iz klase Iter.

Prilikom rada s iteratorima centralno mesto zauzima pojam tekućeg elementa zbirke. Obrada je moguća dok postoji tekući element. Uloga apstraktne metode ima() je da potvrди postojanje tekućeg elementa.

Definicija apstraktne klase za iteratore (Iter).

```
#ifndef _iter_h_
#define _iter_h_

class Figura; // Deklaracija klase Figura.

class Iter {
public:
    virtual ~Iter () {}
    virtual bool ima () const =0;
    virtual Iter& poc () =0;
    virtual Iter& operator++ () =0;
    virtual Figura* operator-> () =0;
    friend Figura& operator* (Iter& i)
    { return *i.operator->(); }
    virtual Iter& operator+= (Figura* fig) =0;
    virtual Iter& operator-= (Figura*& fig) =0;
    Figura* operator~ ()
    { Figura* fig; *this -= fig; return fig; }

    // Varijante ranijih operacija za nepromenljive zbirke.
    virtual const Iter& poc () const =0; // Pomeranje na početak.
    virtual const Iter& operator++ () const =0; // Pomeranje korak napred.
    virtual const Figura* operator-> () const =0; // Pokazivač na figuru.
    friend const Figura& operator* (const Iter& i) // Vrednost figure.
    { return *i.operator->(); }

};

#endif
```

Program 5.12 – Definicija apstraktne klase za iteratore (iter.h)

Za manipulisanje iteratorom predviđene su samo najosnovnije radnje. Metodom poc() postavlja se na početak zbirke, tj. prvi element zbirke postaje tekući element. Metodom operator++() element iza tekućeg postaje novi tekući element. Greška je ako pre primene operatora ++ ne postoji tekući element. Posle pomeranja može da više ne bude tekućeg elementa, ako se stiglo do kraja zbirke. Redosled elemenata u zbirci zavisi od vrste same zbirke. U praksi se ponekad koriste i „dvosmerni“ iteratori, koji omogućavaju kretanje po zbirci i od poslednjeg elementa ka prvom elementu. Tada treba da postoje još dve metode: za postavljanje na poslednji element zbirke i za pomeranje na prethodni element u odnosu na tekući element. Ne podržavaju sve zbirke kretanje u dva smera.

Za pristup do tekuće figure u zbirci preklapljeni su operatori -> i unarna *. Rezultat operatorka -> je adresa tekuće figure u zbirci, što omogućava pisanje izraza i->P() da bi se pozvala metoda P() za tekuću figuru. Rezultat unarnog operatorka * je upućivač na tekuću figuru, što omogućava da se piše izraz cout<<*i da bi se sadržaj tekuće figure ispisao na glavnom izlazu. Operatorska funkcija operator->() je apstraktna metoda jer dohvatjanje adrese tekuće figure nije moguće bez poznavanja strukture zbirke. Za razliku od operatorka ->, unarni operatorka * mogao je da se ostvari primenom standardnog unarnog operatorka * na pokazivač koji je rezultat operatorka ->. Skreće se pažnja na to da se na ovom mestu operatorska funkcija za operatorka -> može da poziva samo izrazom

`i.operator->()`, jer samo `i->` (bez imena nekog člana s desne strane operatora) je neispravan izraz. Unarni operator `*` je ostvaren prijateljskom funkcijom, a ne metodom, da bi mogla da se poziva i za konkretnе vrste figura neposredno, a ne samo polimorfno pomoću pokazivača ili upućivača na apstraktну figuru.

Za umetanje nove figure ispred tekuće figure zbirke u programu 5.12 iskorišćen je operator `+=`. Desni operand je pokazivač na figuru koja se (slično operatoru `==`) klase Zbirka u programu 5.10) predaje u isključivu nadležnost zbirke s kojom je iterator povezan.

Za vađenje tekuće figure iz zbirke iskorišćen je operator `-=`. Desni operand je upućivač na pokazivač na figuru u koji se, kao drugi bočni efekat, stavlja pokazivač na izvađenu figuru (prvi bočni efekat je menjanje iteratatora koji je prvi operand). Ovom operacijom figura se predaje u nadležnost korisnika, što znači da je njegova dužnost da tu figuru uništi (operatorom `delete`) kada ona više ne bude potrebna. Za vađenje figure iz zbirke postoji i metoda `operator~()`, kojom se preklapa unarni operator `~`. Kod ove metode pokazivač na izvađenu figuru vraća se kao vrednost funkcije, a ne bočnim efektom kao kod operatora `-=`. Ostvarena je korišćenjem upravo tog operatora `-=` i zato nije apstraktna, čak nije ni virtualna.

Predviđeno je (očekuje se od izvedenih klasa da to ispoštuju) da vrednost nekih od prethodnih operatora bude upućivač na prvi operand kao *vrednost (Iter&)*. Time se omogućava izvođenje više radnji nad iteratom, odnosno zbirkom samo jednim izrazom. Na primer, izrazom `(+++(i.poc()+=pf1)-=pf2)-=pf3` prvo se iterator i postavi na početak zbirke (`i.poc()`), ispred tekućeg (prvog) elementa se ubaci figura na koju pokazuje pokazivač `pf1 (...+=pf1)`, pomeri se za dva koraka unapred u zbirci (`++ +...`), izvadi se tekuća figura iz zbirke i njena adresa se stavlja u pokazivač `pf2 (...-=pf2)` i izvadi se još jedna figura stavljajući njenu adresu u pokazivač `pf3 (...-=pf3)`.

Od gore navedenih operacija, za one koje ima smisla izvoditi i nad nepromenljivim zbirkama (`poc()`, `++`, `->` i `*`), u programu 5.12 predviđene su zasebne varijante. One se prepoznaju po tome da im je tekući objekat ili pravi parametar tipa `const Iter&`, a vrednosti su upućivači ili pokazivači na nepromenljive iteratore ili figure. Skreće se pažnja na to da nepromenljivi iteratori (tipa `const Iter`) nemaju nikakvu upotrebnu vrednost. Zbog toga tip `const Iter` ovde ne znači „nepromenljiv iterator”, već „iterator za nepromenljive zbirke”. Ovo, pored načina korišćenja operatora `->` i unarna `*`, predstavlja još jednu sličnost iteratora i pokazivača. Kod pokazivača na nepromenljive podatke (`const tip*`), modifikator `const` ne znači da je pokazivač nepromenljiv, već da su pokazivani podaci nepromenljivi. Pokazane sličnosti su savim prirodne, ako se uzme u obzir da osnovna uloga iteratora slična ulazi pokazivača: i jedni i drugi omogućavaju pristup drugim podacima.

Ako nepromenljivim zbirkama budu pridruživani „iteratori za nepromenljive zbirke”, biće obezbjeđeno da pomoću takvih iteratora neće moći da se promene sadržaji tih zbirki.

Varijante operacija koje su za slučaj promenljivih zbirki predstavljene apstraktnim metodama, za slučaj nepromenljivih zbirki su, takođe, predstavljene apstraktnim metodama. Od navedenih operacija samo operatorska funkcija za preklapanje unarnog operatora `*` nije apstraktna. Sadržaj obe varijante funkcije `operator*()` je doslovce isti, ali pošto je tip parametra u jednom slučaju `Iter&`, a u drugom `const Iter&`, izrazom `i.operator->()` pozivaće se jedna ili druga varijanta metode `operator->()`.

Pošto su sve metode u klasi `Iter` apstraktne ili su definisane u samoj klasi i sve prijateljske funkcije su definisane u samoj klasi, ne postoji zasebna datoteka s odvojenim definicijama metoda i prijateljskih funkcija (datoteka `.C`).

Program 5.13 prikazuje definiciju klase `Niz` za dinamičke nizove geometrijskih figura, kao javno izvedenu klasu iz klase `Zbirka`. Činjenica da je klasa `Niz` izvedena iz klase `Zbirka`, garantuje prisustvo svih operacija predviđenih za zbirke, a mogu da postoje i druge operacije.

```
// Definicija klase za nizove figura (Niz).

#ifndef _niz2_h_
#define _niz2_h_

#include "zbirka.h"

class Niz: public Zbirka {
    friend class NIter;
    Figura** niz;                                // Iterator za nizove.
    int kap, vel;                                 // Elementi niza.
    void kopiraj (const Niz& n);                 // Kapacitet i veličina.
    void brisi () { ~*this; delete [] niz; }      // Kopiranje u niz.
    public:
        explicit Niz (int k=10) { niz = new Figura* [kap = k]; vel = 0; } // Uništavanje niza.
        Niz (const Niz& n) { kopiraj (n); }          // Stvaranje praznog niza.
        ~Niz () { brisi (); }                         // Konstruktor kopije.
        Niz& operator= (const Niz& n) {              // Destruktor.
            if (this != &n) { brisi (); kopiraj (n); } // Dodata vrednosti.
            return *this;
        }
        int operator+ () const { return vel; }        // Broj elemenata zbirke.
        Niz& operator+= (Figura* fig);                // Dodavanje figure.
        Figura*& operator[] (int ind) {               // Donvatanje figure.
            if (ind<0 || ind >= vel) greska (G_IND);
            return niz[ind];
        }
        const Figura* operator[] (int ind) const {     // Praznjenje niza.
            return const_cast<Niz&>(*this)[ind];
        }
        Niz& operator~ () {                           // Stvaranje kopije niza.
            Niz* kopija () const {                   // Stvaranje iteratora.
                return new Niz (*this);
            }
            NIter* iter () { }                      // Stvaranje iteratora.
            const Iter* iter () const { }           // Stvaranje iteratora.
        }
};

#endif
```

Program 5.13 – Definicija klase za nizove figura (niz2.h)

Prvom naredbom u definiciji klase `Niz` deklariše se klasa `NIter` (definicija se nalazi u kasnjem programu 5.14), koja predstavlja klasu iteratora za obilazak elemenata objekata tipa `Niz`. U paru klasa zbirka – iterator, zbirka uvek mora da podržava rad iteratora. Ovde se ta podrška ogleda proglašavanjem klase `NIter` prijateljskom klasom (`friend`). Ovakvo

rešenje je prihvatljivo kada isti programer projektuje i klasu zbirke i klasu iteratora. Druga moguća podrška je postojanje javnih metoda u zbirci čijim pozivanjem iterator može da obilazi zbirku. Ovo rešenje je manje efikasno, ali je bezbednije, naročito ako pomenute klase projektuju dva programera. Projektant klase iteratora možda nije dovoljno dobro upoznat s pojedinostima klase zbirke.

Interna struktura objekata tipa `Niz` je dinamički niz pokazivača na figure, predstavljen poljem `niz` koje je tipa `Figura**`. Fizička veličina niza (kapacitet) predstavljena je poljem `kap`, a logička veličina (broj popunjениh mesta) poljem `vel`.

U privatnom delu klase `Niz` nalaze se još dve metode koje treba da pojednostavne neke od javnih metoda. Metoda `kopiraj()` kopira svoj parametar u tekući objekat. Metoda `brisi()` se koristi pri uništavanju niza, što podrazumeva uništavanje sadržanih figura (pozivanjem kasnije definisane operatorske funkcije `operator~()`, izrazom `~*this`) i oslobadanjem memorije dodeljene za smeštanje pokazivača na figure (`delete`).

Na početku javnog dela klase `Niz` nalaze se obavezni konstruktori, destruktur i operator parametar ima podrazumevanu vrednost, istovremeno predstavlja i podrazumevani konstruktor. Da se ne bi koristio kao konstruktor konverzije (što nema smisla), sprečava se na uobičajeni način, pozivanjem metoda `brisi()` i `kopiraj()` (videti klasu Red u programu 4.13).

U nastavku programa 5.13 navedene su definicije jednostavnijih, odnosno deklaracije složenijih apstraktnih metoda predviđenih u osnovnoj klasi `Zbirka` (program 5.10).

U jednostavne operacije spadaju dohvatanje broja figura u nizu (`vel()`) i indeksiranje (`[]`). Ako je indeks pri indeksiranju izvan dozvoljenih granica program se prekida pozivom metode `greska()`, simboličkom konstantom `G_IND` kao argumentom (metoda i simbolička konstanta su nasledene iz klase `Zbirka`). Varijanta operatora `[]` za nepromenljive promenljive nizove (`operator[](int)`) je ostvarena pozivanjem varijante istog operatora za `(*this)` tipa `const Niz&`, potrebno je operatorom `const_cast` skinuti modifikator `const`, posle čega će operator `[]` pozivati prvu varijantu iste operacije. Skidanje modifikatora `const` u ovom slučaju nije opasno, pošto se sa sigurnošću zna da ni prva varijanta operatora `[]` ne menja vrednost svog tekućeg objekta. Dobijeni rezultat tipa `Figura*` se automatskom konverzijom tipa pretvara u potrebnii tip `const Figura*`.

Pre definicije metoda klase `Niz` koje nisu definisane unutar klase, neophodno je definisati klasu za iteratore `NIter`, pošto neke metode klase `Niz` ne mogu da se definisu bez poznavanja potpunog sadržaja klase `NIter`. Program 5.14 prikazuje definiciju klase `NIter`.

Privatna polja u klasi `NIter` su pokazivač `niz` na niz čije elemente obilazi posmatrani iterator i indeks `tek` tekućeg elementa u nizu. Modifikatorom `mutable` je omogućeno da se vrednost indeksa `tek` može menjati čak i u objektima tipa `const NIter`. Ovim je postignuta već pomenuta osobina iteratora da `const NIter` označava „iterator nepromenljivog niza“ umesto „nepromenljivog iteratora“.

Jedini javan konstruktor samo preuzima pokazivač na niz kome se pridružuje stvoreni iterator. Parametar je deklarisana kao pokazivač na nepromenljiv niz (`const Niz*`), da bi mogao da se stvara iterator i za promenljive i za nepromenljive nizove. Pošto je polje `niz`

```
Definicija klase iteratora za nizove (NIter).

#ifndef _niter_h_
#define _niter_h_

#include "iter.h"
#include "niz2.h"

class NIter: public Iter {
    Niz* niz; // Pokazivač na niz.
    mutable int tek; // Indeks tekućeg elementa.

public: // Stvaranje iteratora.
    NIter (const Niz* n) // Ima li još elemenata?
        : niz = const_cast<Niz*>(n), tek = 0; // Pomeranje na početak.
    bool ima () const // Pomeranje korak napred.
    { return tek >= 0 && tek < niz->vel; }
    NIter& poc () // Pomeranje na početak.
    { tek = 0; return *this; } // Umetanje figure.
    NIter& operator++ () // Vadenje figure.
    { if (!ima ()) Zbirka::greska (Zbirka::G_TEK); // Pokazivač na figuru.
        tek++; // Vadenje figure.
        return *this; }
    Figura* operator-> () // Varenje ranijih operacija za nepromenljive nizove.
    { if (!ima ()) Zbirka::greska (Zbirka::G_TEK); // Pomeranje na početak.
        return niz->niz[tek]; } // Pomeranje korak napred.
    NIter& operator+= (Figura* fig); // Umetanje figure.
    NIter& operator-= (Figura* fig); // Vadenje figure.

    // Varijante ranijih operacija za nepromenljive nizove.
    const NIter& poc () const // Pomeranje na početak.
    { return const_cast<NIter*>(*this)->poc(); }
    const NIter& operator++ () const // Pomeranje korak napred.
    { return ++ const_cast<NIter*>(*this); } // Pokazivač na figuru.
    const Figura* operator-> () const // Varenje ranijih operacija za nepromenljive nizove.
    { return (const_cast<NIter*>(*this))->operator->(); }

#endif
```

Program 5.14 – Definicija klase iteratora za nizove (niter.h)

definisano kao `Niz*` (bez `const`), pri dodeli vrednosti neophodno je skinuti modifikator `const` sa tipa parametra konstruktora. Tu se nalazi slaba tačka celog sistema. Da ne bi došlo do problema korisnik klase `Niz` i `NIter`, iterator za nepromenljiv niz `cniz` (tipa `const Niz`) mora da definiše naredbom `const NIter ci(&cniz);`. Pomoću iteratora `ci` neće moći ništa da se promeni u objektu `cniz`. Nažalost, korisnik može da definiše iterator i naredbom `NIter i(&cniz);`, kada će pomoći iteratora i moći da promeni vrednost objekta `cniz`! Ne postoji način da se to spreči.

Skreće se pažnja na to da, bez obzira što objekti tipa `NIter` sadrže pokazivačko polje, nije potrebno praviti konstruktor kopije, destruktur, ni operator za dodelu vrednosti. Ako se iterator kopira kopija treba da bude pridružena istom nizu, pa generisani konstruktor kopije

i operator = zadovoljavaju. Kada se iterator uništava, niz kome je pridružen ne treba da se uništi. Generisani destruktur, opet, zadovoljava.

U nastavku programa 5.14 definisane su jednostavnije i deklarisane složenije apstraktne metode nasleđene iz klase Iter.

Tekući element postoji ako je indeks tek u opsegu od nula do niz->vel-1 (metoda ima()).

Pomeranje na početak niza podrazumeva stavljanje nule u indeks tek (metoda poc()).

Pomeranje korak napred zahteva samo povećavanje indeksa tek za jedan (operator ++), pod uslovom da na početku operacije postoji tekući element. Ako ne, program se prekida pozivanjem metode greska() iz klase Zbirka. Pošto se trenutno ne nalazi u klasi Zbirka, niti u iz nje izvedenoj klasi Niz, morao da se kristi operator za razrešenje dosega (::) kako za pozivanje metode greska() (Zbirka::greska(...)) tako i za korišćenje simboličke konstante G_TEK (Zbirka::G_TEK).

Dohvatjanje pokazivača na tekuću figuru (operator ->) zahteva samo dohvatanje odgovarajućeg elementa niza niz u objektu na koji pokazuje pokazivač niz (niz->niz[tek]). Naravno, samo ako postoji tekući element.

Umetanje (operator +=) i vađenje (operator -=) figure, u slučaju niza su složene radnje, pa je njihova definicija odložena za kasnije.

Na kraju programa 5.14 nalaze se definicije varijanti operacija za nepromenljive nizove za koje to ima smisla. Jednoobraznosti radi, sve su ostvarene pozivanjem varijanata za promenljive nizove istih operacija.

Posle definicija klase Niz i NIter, njihove metode izvan klase mogu da se odvojeno definišu po proizvoljnom redosledu. Kao prve, u programu 5.15 navedene su definicije metoda uz klasu Niz.

U metodi kopiraj() prvo se dodeli memorija za niz pokazivača na figure iste veličine kao i kapacitet originalnog niza (n.kap). Posle se polimorfno kopiraju svih n.vel figura pozivanjem virtualne metode kopija() iz klase figura kojoj pripadaju pojedine figure.

Dodavanje figure u metodi operator+=() izvodi se uz automatsko povećavanje kapaciteta niza kad god je niz pun. Povećavanje kapaciteta je dugotrajna radnja jer podrazumeva dodelu memorije pomoćnom nizu s povećanom veličinom (pom=new Figura* [...]), kopiranje pokazivača iz starog niza u novi (pom[i]=niz[i] u ciklusu), uništavanje starog niza (delete [] niz) i proglašavanje novog niza važećim sadržajem objekta (niz=pom). Da se sve to ne bi radilo pri svakom dodavanju novog elementa nizu, kapacitet se uvek povećava za 10% trenutnog kapaciteta, ali bar za 10 elemenata. Kada u nizu postoji slobodno mesto (zatećeno ili novododata), pokazivač nove figure se prosti stavlja iza poslednje popunjene mesta (niz[vel++]=fig). Skreće se pažnjava na to da se ne pravi kopija figure, pa se smatra da je pokazivana figura prešla u nadležnost niza. Stari vlasnik figure nikako ne sme da uništi tu figuru, jer bi time ošteto i sâm niz.

Pražnjenje niza (operator~()) podrazumeva samo uništavanje figura u nizu, ne menjajući pri tome kapacitet niza.

Stvaranje iteratora za promenljive nizove metodom iter(), odnosno za nepromenljive nizove metodom iter() const je jedini bezbedan način stvaranja iteratora (u odnosu na ranije pomenuto neposredno pozivanje konstruktora NIter()). Naime cniz.iter(), gde je cniz tipa const Niz, sigurno stvara iterator pomoću kojeg neće moći da se promeni sadržaj objekta cniz. Ranije pomenuti nepouzdani način stvaranja iteratora mogao bi da se odstrani proglašavanjem konstruktora NIter() privatnim konstruktorm i obe varijante

```
// Definicije metoda uz klasu Niz.

#include "niz2.h"
#include "niter.h"

void Niz::kopiraj (const Niz& n) { // Kopiranje u niz.
    niz = new Figura* [kap = n.kap];
    vel = n.vel;
    for (int i=0; i<vel; i++) niz[i] = n.niz[i]->kopija();
}

Niz& Niz::operator+= (Figura* fig) { // Dodavanje figure.
    if (vel == kap) {
        Figura** pom = new Figura* [kap += (kap<100) ? 10 : kap/10];
        for (int i=0; i<vel; i++) pom[i] = niz[i];
        delete [] niz; niz = pom;
    }
    niz[vel++] = fig;
    return *this;
}

Niz& Niz::operator~ () { // Pražnjenje niza.
    for (int i=0; i<vel; delete niz[i++]);
    vel = 0;
    return *this;
}

Iter* Niz::iter () { // Stvaranje iteratora.
    const Iter* Niz::iter () const
    { return new NIter (this); }
}
```

Program 5.15 – Definicije metoda uz klasu Niz (niz2.C)

metode Niz::iter() prijateljskom funkcijom klase NIter. Tada jedini bi način za stvaranje iteratora bio pomoću metoda iter(). Postoji jedan tehnički problem s tim rešenjem: tada se iteratori stvaraju isključivo u dinamičkoj zoni memorije, kojima se pristupa pokazivačima. Izrazi s pokazivačima na iteratore su nezgrapniji od izraza s iteratorima (na primer: (*pi)->P() umesto i->P(), odnosno cout<<*pi umesto cout<<i). Takođe, bez konverzije tipa, pokazivač pi morao bi da se definiše da je tipa Iter* (Iter* pi=niz.iter()). Za naglašavanje da se radi o iteratoru za nizove moralо bi da se piše NIter* pi=(NIter*)niz.iter().

Obe verzije metode iter() su vrlo jednostavne, ali nisu mogle da se definišu unutar same klase Niz, jer u vreme definisanja klase Niz (u programu 5.13) klasa NIter još nije bila definisana, već samo deklarisana (klasa NIter je definisana tek u programu 5.14). Skreće se pažnja na to da ovaj problem ne bi mogao da se razreši ni obrnutim redosledom definisanja klase Niz i NIter, pošto obe klasе koriste ponešto iz druge klase. Ako bi se prvo definisala klasа NIter, u to vreme klasа Niz mogla bi da bude samo deklarisana. Zbog toga metode u klasi NIter koje koriste članove klase Niz ne bi mogle biti ugrađene metode, bilo definisanjem unutar klase, bilo izvan klase. Pošto takvih metoda u klasi NIter ima više i češće se pozivaju od dve varijante metoda Niz::iter(), takvo rešenje bi bilo nepovoljnije od ovde prikazanog.

U opštem slučaju, ako dve klase A i B uzajamno koriste objekte tipa suprotne klase ili članove suprotne klase, postupak za njihovo ostvarivanje je sledeći:

- deklarisati klasu B;
- definisati klasu A, pri čemu metode koje na bilo koji način koriste klasu B (parametri ili rezultati su tipa B* ili B&, odnosno koriste lokalne objekte tipa B);
- definisati klasu B, pri čemu čak i metode koje na bilo koji način koriste klasu A mogu da se definišu i unutar klase B;
- ako se obe klase definišu unutar iste datoteke (što nije baš tipično u praksi), mogu da se definišu ugrađene metode klase A koje koriste klasu B;
- definisati sve još nedefinisane neugrađene metode obe klase u jednoj ili više odvojenih datoteka po proizvoljnom redosledu.

Skreće se pažnja na to da ako se klase A i B definišu u dvema odvojenim datotekama A.h i B.h, tada metode klase A pomenute gore u četvrtom koraku ne mogu nikako da budu ugrađene metode. Odlučivanje koja će od dve klase, koje se međusobno koriste, da bude A. a koja B, treba sprovesti tako da šteta zbog nemogućnosti ugrađivanja u kôd nekih metoda klase A bude što manja.

Program 5.16 prikazuje definicije metoda klase NIter koje nisu ugrađene metode.

```
// Definicije metoda uz klasu NIter.

#include "niter.h"

NIter& NIter::operator+= (Figura* fig) { // Umetanje figure.
    *niz += 0;
    for (int i=niz->vel-1; i>tek; i--) niz->niz[i] = niz->niz[i-1];
    return *this;
}

NIter& NIter::operator-= (Figura* fig) { // Vadenje figure.
    if (!ima ()) Zbirka::greska (Zbirka::G_TEK);
    fig = niz->niz[tek];
    for (int i=tek+1; i<niz->vel; i++) niz->niz[i-1] = niz->niz[i];
    return *this;
}
```

Program 5.16 – Definicije metoda uz klasu NIter (niter.C)

Treba uočiti da su kako umetanje figure u niz (**operator+=()**), tako i vadenje figure iz niza (**operator-=()**) relativno dugotrajne operacije, jer podrazumevaju pomeranje izvesnog broja elemenata niza za po jedno mesto naviše, odnosno naniže. Prilikom umetanja (pokazivača) figure ispred tekućeg elementa, prvo se izrazom ***niz+=0** niz povećava za jednu praznu figuru (dodaje se pokazivač 0), pri čemu se poziva operatorska funkcija **operator+=()** iz klase Niz. Taj će operator obezbediti eventualno povećavanje kapacitete niza (sama logička veličina mogla bi da se uveća i jednostavno izrazom **niz->vel++**). Kao i kod dodavanja na kraj niza u klasi Niz, i pri umetanju ispred tekućeg elementa u niz se stavlja pokazivač na figuru i time se figura prenese u nadležnost niza. Dosadašnji vla-

snik figure više ne bi trebalo da menja sadržaj figure, a nikako ne sme da uništi tu figuru. Kod vadenja figure pokazivač na figuru se dostavlja korisniku kao bočni efekat. Time mu se prenese i pravo i obaveza da figuru uništi kada mu više ne bude potrebna.

Program 5.17 služi za ispitivanje do sada napravljenih klasa za zbirke figura.

// Ispitivanje klasa Niz i NIter.

```
#include "niz2.h"
#include "niter.h"
#include "krug2.h"
#include "kvadrat.h"
#include "trougao.h"
#include <iostream>
using namespace std;

int main () {
    // Stvaranje niza.
    Niz niz; niz += new Krug (3, Tacka(1,2));
    niz += new Trougao (3, 4, 5, Tacka(7,8));
    niz += new Kvadrat (2, Tacka(9,8));
    for (int n=niz, i=0; i<n; niz=*niz[i++]);
    cout << "Pocetni niz:\n" << niz << endl;

    // Uredjivanje niza prema površinama figura.
    for (int i=0; i<+niz-1; i++)
        for (int j=i+1; j<+niz; j++)
            if (niz[j]->P() < niz[i]->P())
                { Figura* f = niz[i]; niz[i] = niz[j]; niz[j] = f; }
    cout << "Uredjeni niz:\n";
    for (const NIter i(&niz); i.ima(); ++i)
        cout << " " << *i << endl;

    // Vadenje svakog drugog elementa niza.
    cout << "\nIzvadnjene figure:\n";
    for (NIter i(&niz); i.ima(); ++i)
        { Figura* f = ~i; cout << " " << *f << endl; delete f; }
    cout << "\nPrepolovljeni niz:\n" << niz << endl;
}
```

Program 5.17 – Ispitivanje klasa Niz i NIter (niz2t.C)

Na početku prve grupe naredbi definiše se prazan niz **niz** i u njega se stavljuju tri figure različitih vrsta. Posle toga se kopije tih figura dodaju na kraj niza, čime se udvostruči dužina niza. Dužina ciklusa jednaka je prvo bitnoj veličini niza (**n+=niz**). Pokazivači na figure za kopiranje dohvataju se operatorskom funkcijom za indeksiranje (**niz[i++]**). Tip vrednosti izraza ***niz[i++]** je **Figura&**, pa izrazom **niz+=*niz[i++]** poziva se operatorska funkcija **operator+=(Zbirka&, const Figura&)**, koja kopiju figure stavlja u Zbirku, tj. u ovom slučaju u **niz**. Poslednjom naredbom prve grupe naredbi se ispisuje sadržaj niza, pozivanjem operatorske funkcije **operator<<(ostream&, const Zbirka&)**.

U drugoj grupi naredbi **niz** figura **niz** se uređuje prema neopadajućem poretku veličina površina sadržanih figura. U toku uređivanja unutar niza se premeštaju pokazivači na figu-

re. Elementi uređenog niza ispisuju se jedan po jedan u ciklusu čijim tokom se upravlja iteratorm. Pošto se u toku ispisivanja sadržaj niza ne menja, stvara se iterator i za nepromenljive nizove (tip `const NIter`) koji se inicijalizuje adresom niza `niz`. Ne smeta što je niz promenljiv objekat. Ciklus traje sve dok postoji tekući element (`i.ima()`), tj. dok se nije došlo do kraja niza. Unutar ciklusa izrazom `*i` se dohvata tekuća figura i ispisuje se njen sadržaj. Na kraju svakog prolaza kroz ciklus izrazom `++i` se prelazi na sledeći element niza.

U poslednjoj grupi naredbi se iz niza izvadi svaka druga figura. Posto se sada menjaju sadržaj niza, stvara se iterator i za promenljive nizove (tip NIter). U toku prvog prolaza kroz ciklus izrazom ~i izvadi se prva figura (smanjuje se dužina niza za jedan) i dobijeni pokazivač se smešta u lokalni pokazivač f. Posle ispisivanja sadržaja te figure (...<<*f<<...), figura se uništi (**delete** f). Na kraju prvog prolaza izrazom ++i preskače se druga figura prvobitnog niza (to je, u stvari, prva figura skraćenog niza). U drugom prolazu se na sličan način izvadi tekuća figura (treći element prvoibtnog niza, odnosno drugi element trenutnog niza), a sledeći element se preskoči. Ovo se nastavlja sve dok se ne dođe do kraja niza koji se u toku iteracija još i skraćuje. Po izlasku iz ciklusa preostali sadržaj niza se ispisuje operatorom <<. Skreće se pažnja na to da se posmatrani ciklus ispravno završava samo ako početni niz ima paran broj elemenata.

Rezultat 5.2 prikazuje primer rada programa 5.17.

Program 5.18 prikazuje definiciju apstraktne klase `Lista` za ostvarenje zbirki geometrijskih figura u obliku liste.

Klasa iteratora za liste zove se **LIter** (definisana je kasnije u programu 5.19). Da bi se njene funkcije mogle što efikasnije ostvariti ona je proglašena prijateljem klase **Lista**.

Za predstavljanje elemenata jednostrukog povezane liste definisana je unutrašnja struktura `Elem` koja sadrži pokazivač na sadržanu figuru, pokazivač na sledeći element liste i konstruktor radi lakše izgradnje liste.

Na kraju privatnog dela klase `Lista` nalaze se uobičajene pomoćne metode `kopiraj()` i `brisici()` čijom upotreboom su ostvareni obavezni konstruktor kopije, destruktorni operator = na početku javnog dela klase. Od konstruktora postoji još podrazumevani konstruktor koji stvara praznu listu.

U nastavku klase `Lista` nalaze se definicije jednostavnijih i deklaracije složenijih metoda kao ostvarenja odgovarajućih apstraktnih metoda osnovne klase `Zbirka`. Kao jednostavnije metode uzete su sve metode koje ne sadrže ciklus. Iz istih tehničkih razloga kao i kod nizova, i ovde obe varijante metode `iter()` mogле su samo da se deklarišu.

Program 5.19 prikazuje definiciju klase `LIter` kao ostvarenje iteratora za zbirke geometrijskih figura u obliku liste.

Kao privatna polja postoje pokazivač `lst` na listu kojoj se pridružuje iterator i uvek promenljivi (**mutable**) pokazivači `tek` na tekući element i `preth` na prethodni element u odnosu na tekući. Pokazivač `preth` je uveden da olakša izbacivanje tekućeg elementa iz liste. Podsećanja radi, pokazivači `tek` i `preth` su proglašeni uvek promenljivim, da bi tip `const Iterator` označavao iterator nepromenljive liste, a ne nepromenljiv iterator.

Vidi se da su sve iteratorske operacije u slučaju liste jednostavne, u smislu da mogu da se ostvare bez ciklusa. Za razliku od nizova, kada su umetanje i vađenje figure zahtevali premeštanje pokazivača u ciklusu.

1.6. Saita zhirka krug2 kvadrat trouga

CC niz

niz2t

```

Pocetni niz:
Niz{
    krug      [T=(1,2), r=3, O=18.8496, P=28.2743]
    trougao  [T=(7,8), a=3, b=4, c=5, O=12, P=6]
    kvadrat  [T=(9,8), a=2, O=8, P=4]
    krug      [T=(1,2), r=3, O=18.8496, P=28.2743]
    trougao  [T=(7,8), a=3, b=4, c=5, O=12, P=6]
    kvadrat  [T=(9,8), a=2, O=8, P=4]

```

Uredjeni niz:

kyadrat [$T=9,8$], $a=2$, $O=8$, $P=4$

kvadrat [$T=(9,8)$, $a=2$, $O=8$, $P=4$]
 trougao [$T=(7,8)$, $a=3$, $b=4$, $c=5$, $O=12$, $P=6$]
 trougao [$T=(7,8)$, $a=3$, $b=4$, $c=5$, $O=12$, $P=6$]
 krug [$T=(1,2)$, $r=3$, $O=18.8496$, $P=28.2743$]
 krug [$T=(1,2)$, $r=3$, $O=18.8496$, $P=28.2743$]

Izvadadjene figure:

kvadrat [T=(9,8), a=2, O=8, P=4]
 trougao [T=(7,8), a=3, b=4, c=5, O=12, P=6]
 krug [T=(1,2), r=3, O=18.8496, P=28.2743]

Prepolovljeni nizi

```

Nizi
kvadrat [T=(9,8), a=2, O=8, P=4]
trougaoo [T=(7,8), a=3, b=4, c=5, O=12, P=6]
kruga   [T=(1,2), r=3, O=18.8496, P=28.2743]

```

Rezultat 5.2 – Obrada nizova geometrijskih figura programom 5.17

Pošto su sve metode definisane u samoj klasi nije potrebna zasebna datoteka (.C) za njihovo definisanje.

Program 5.20 prikazuje definicije neugrađenih metoda klase Lista.

Skreće pažnja na to da, za razliku od nizova, liste nisu pogodne za pristup do elemenata liste na osnovu rednog broja (indeksa). Zbog sekvencijalne prirode, za svaki takav pristup mora da se u ciklusu traži element datog rednog broja od početka liste.

Slika 5.9 prikazuje dijagram klasa za ostvarene klase zbirki geometrijskih figura i pratećih iteratorskih klasa.

Mogu da se uoče dva paralelna hijerarhijska sistema klasa. Jedan sistem čine same zbirke a drugi njima pridruženi iteratori. Svakoj klasi zbirke odgovara jedna klasa iteratora. Zbirka zavisi od iteratora (zbog operacije stvaranja iteratora). Postoji jednosmerna asocijacija od iteratora ka zbirci kojoj je pridružena (iterator poseduje pokazivač na zbirku koju obilazi, a zbirka ne zna za iteratore koji je obilaze). Same zbirke su agregati figura koje mogu da se stavljaju u zbirke i da se vade iz zbirki.

Klasa za liste geometrijskih figura (Lista).

```
#ifndef _lista_h_
#define _lista_h_

#include "zbirka.h"

class Lista: public Zbirka {
    friend class LIter;
    struct Elem {
        Figura* fig;
        Elem* sled;
        Elem (Figura* f, Elem* s=0)
            : fig(f), sled(s) {}
        Elem *prvi, *posl;
        int vel;
        void kopiraj (const Lista& lst);
        void brisi ();
    public:
        Lista () { prvi = posl = 0; vel = 0; }
        Lista (const Lista& lst) { kopiraj (lst); }
        ~Lista () { brisi (); }
        Lista& operator= (const Lista& lst) {
            if (this != &lst) { brisi (); kopiraj (lst); }
            return *this;
        }
        int operator+ () const { return vel; }
        Lista& operator+= (Figura* fig) {
            posl = (!prvi ? prvi : posl->sled) = new Elem (fig);
            vel++;
            return *this;
        }
        Figura* operator[] (int ind);
        const Figura* operator[] (int ind) const
            { return const_cast<List*>(*this)[ind]; }
        Lista& operator~ ()
            { brisi (); return *this; }
        Lista* kopija () const
            { return new Lista (*this); }
        Iter* iter ();
        const Iter* iter () const;
    };
#endif
```

Program 5.18 – Klasa za liste geometrijskih figura (lista.h)

Definicija klase iteratore za liste (LIter).

```
#ifndef _liter_h_
#define _liter_h_

#include "iter.h"
#include "lista.h"

class LIter: public Iter {
    Lista* lst;
    mutable Lista::ELEM *preth, *tek;
public:
    LIter (const Lista& lic)
        : lst = const_cast<List*>(&lic), preth=0, tek = lst->prvi {}
    bool ima () const
        { return tek != 0; }
    LIter poc ()
        { preth = 0; tek = lst->prvi; return *this; }
    Iter operator++ () {
        if (!ima ()) Zbirka::greska (Zbirka::G_TEK);
        preth = tek; tek = tek->sled;
        return *this;
    }
    Figura* operator-> () {
        if (!ima ()) Zbirka::greska (Zbirka::G_TEK);
        return tek->fig;
    }
    LIter& operator+= (Figura* fig) {
        Lista::ELEM novi = new List::ELEM (fig, tek);
        (!preth ? lst->prvi : preth->sled) = tek;
        if (!tek) lst->posl = novi;
        return *this;
    }
    LIter& operator-= (Figura* fig) {
        if (!ima ()) Zbirka::greska (Zbirka::G_TEK);
        fig = tek->fig;
        List::ELEM stari = tek;
        tek = (!preth ? lst->prvi : preth->sled) = tek->sled;
        delete stari; lst->vel--;
        return *this;
    }
    // Varijante ranijih operacija sa nepromjenljivim listama.
    const LIter poc () const
        { return const_cast<LIter*>(&(this)->poc()); }
    const LIter& operator++ () const
        { return ++ const_cast<LIter*>(&(this)); }
    const Figura* operator-> () const
        { return (const_cast<LIter*>(&(this))->operator->()); }
};
```

#endif

Program 5.19 – Definicija klase iteratara za liste (liter.h)

```

// Definicije metoda uz klasu Lista.

#include "lista.h"
#include "liter.h"

void Lista::kopiraj (const Lista& lst) { // Kopiranje u listu.
    prvi = posl = 0; vel = lst.vel;
    for (Elem* tek=lst.prvi; tek; tek=tek->sled)
        posl = (!prvi ? prvi : posl->sled) = new Elem (tek->fig->kopija());
}

void Lista::brisi () { // Uništavanje liste.
    while (prvi) {
        Elemt stari = prvi; prvi = prvi->sled;
        delete stari->fig; delete stari;
    }
    posl = 0; vel = 0;
}

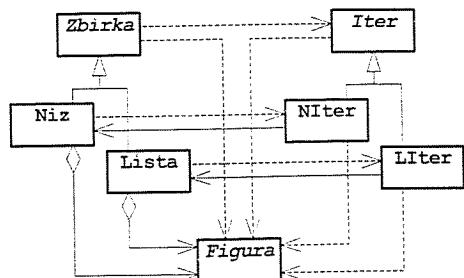
Figura* & Lista::operator[] (int ind) { // Dohvatanje figure.
    if (ind<0 || ind >= vel) greska (G_IND);
    Elemt tek = prvi;
    for (int i=0; i<ind; i++) tek = tek->sled;
    return tek->fig;
}

Iter* Lista::iter () { // Stvaranje iteratora.
    return new LIter (this); }

const Iter* Lista::iter () const { // Stvaranje iteratora.
    return new LIter (this); }

```

Program 5.20 – Definicije metoda uz klasu Lista (lista.C)



Slika 5.9 – Klase za zbirke geometrijskih figura

Na kraju, programi 5.21 i 5.22 sadrže interaktivni program za ispitivanje svih prikazanih klasa.

Program 5.21 sadrži dve pomoćne funkcije `citaj()` za stvaranje nove figure čitajući podatke s glavnog ulaza i `stvari()` za stvaranje nove prazne zbirke čitajući vrstu zbirke s glavnog ulaza. Ove dve funkcije su jedina mesta u celokupnom programskom sistemu gde

// Ispitivanje klasa za zbirke figura.

```

#include "krug2.h"
#include "kvadrat.h"
#include "trougaol.h"
#include "niz2.h"
#include "lista.h"
#include "liter.h"
#include "figura.h"
#include <iostream>
using namespace std;

```

Figura* citaj () { // ČITANJE FIGURE.

```

cout << "Vrsta (o, k, t)? "; char vrsta; cin >> vrsta;
Figura* pf = 0;
switch (vrsta) {
    case 'o': case 'O': pf = new Krug; cout << "x, y, r? "; break;
    case 'k': case 'K': pf = new Kvadrat; cout << "x, y, a? "; break;
    case 't': case 'T': pf = new Trougao; cout << "x, y, a, b, c? "; break;
}
if (pf) cin >> *pf;
return pf;
}

```

Zbirka* stvari () { // STVARANJE ZBIRKE.

```

cout << "Vrsta (n,l)? "; char vrsta; cin >> vrsta;
switch (vrsta) {
    case 'n': case 'N': return new Niz;
    case 'l': case 'L': return new Lista;
    default: return 0;
}
}

```

int main () { // GLAVNA FUNKCIJA.

```

Zbirka* pz = new Niz; // Pokazivač na zbirku.
Iter* pi = pz->iter(); // Pokazivač na iterator.
for (bool dalje=true; dalje;) {

```

// Ispisivanje menija.
cout << "\nMoguce operacije su:\n\n";
 "1 Citaj figuru i stavi
 "2 Dohvati figuru i stavi
 "3 Donvati figuru i prikazi
 "4 Promeni vrstu zbirke
 "5 Prikazi zbirku\n"
 "6 Isprazni zbirku
 "\nUnesite svoj izbor: ";
char izb; cin >> izb;
switch (izb) {

7 Pocni iteracije\n"
8 Idi na sledecu i prikazi\n"
9 Prikazi tekucu figuru\n"
0 Brisi tekucu figuru\n"
k Zavrsi s radom\n"

```

    // Čitanje figure i stavljanje na kraj zbirke.
    case 'i': if (Figura* pf = citaj ()) *pz += pf;
                else cout << "**** Neispravna vrsta!\n\";
                break;
}

```

Program 5.21 – Ispitivanje klasa za zbirke figura (zbirkat.C, prvi deo)

bilo šta treba menjati ako se uvođe još neke vrste figura ili zbirki. To je i razumljivo, jer kada se stvara neki novi objekat, treba eksplisitno reći kog je tipa taj objekat i da se eksplisitno pozove odgovarajući konstruktor. Kasnije, kada se obrađuju postojeći objekti, mehanizam virtualnih metoda može automatski da prilagodi ponašanje programa novim upovima objekata, čak i ako u vreme pisanja programa nisu svi (izvedeni) tipovi bili poznati. Funkcije `citaj()` i `stvori()` napravljene su samo zato da tekst glavne funkcije bude nezavisan od spiska korišćenih vrsta geometrijskih figura i vrsta zbirki geometrijskih figura.

Na početku glavne funkcije prvo se stvara jedan prazan niz tipa `Niz` čija adresa se smesti u pokazivač `pz` tipa `Zbirka*`. Ovaj pokazivač će kasnije moći da pokazuje i na neku listu, pa i na neku treću vrstu zbirki kada takva vrsta bude definisana. Stvorenom nizu se odmah pridruži i odgovarajući iterotor pomoću izraza `pz->iter()`. Dobijena adresa se stavlja u pokazivač `pi` tipa `Iter*`.

Program je interaktivni, što znači da korisnik može po želji da biru operacije iz skupa raspoloživih operacija po proizvolnjem redosledu. Spisak mogućih operacija se ispisuje na početku svakog prolaska kroz glavni ciklus programa. Operacije dejstvuju na zbirku prevačem `pz`, a iteratorske radnje se izvode iterotorom predstavljenim pokazivačem `pi`.

Operacije su birane tako da pomoću njih može da se ispita većina, ako ne i sve, mogućnosti klasa za zbirke geometrijskih figura i njima odgovarajućih klasa za iteratore. Detaljna analiza svih operacija prepusta se čitaocu. Ovde su istaknuti samo neki od važnijih detalja.

Operacije 1, 2 i 3 koriste operatore `+= i []` klase `Zbirka`, a operacije 7, 8, 9 i 0 koriste metodu `poc()` i operatore `++, * i ~` klase `Iter` za manipulacije pojedinačnim figurama. Operacije 5 i 6 koriste operatore `<< i ~` klase `Zbirka` za manipulacije zbirkom u celini.

Operacija 4 obavlja relativno složenu obradu i njoj vredi posvetiti više pažnje. Njome se zatečeni sadržaj obradivanje zbirke premesti u novostorenou zbirku koja može biti druge vrste od vrste stare zbirke. Ostvarena je iz oznake `case '4'` u programu 5.22.

Prvo se funkcijom `stvori()` napravi nova prazna zbirka, čija adresa se stavlja u potpunoči pokazivač `ppz`. Vrstu zbirke bira korisnik, i za nastavak programa nije bitna koja vrsta zbirke je napravljena. Ako je stvaranje zbirke uspelo (`ppz!=0`) započinje se nova iteracija u staroj zbirci (`pi->poc()`) i sve dok postoji tekući element (`pi->ima()`) izvadi (`~*pi`). Skreće se pažnja na to da se ne prave duplikati figura iz stare zbirke, već se samo premeštaju njihovi pokazivači iz stare u novu zbirku. Po izlasku ciklusa uništi se stara, sada već prazna zbirka (`delete pz`) i nova zbirka se proglaši važećom zbirkom (`pz=ppz`). Posle se uništi stari iterotor (`delete pi`) i napravi se novi iterotor koji odgovara vrsti nove zbirke (`pi=pz->iter()`). Skreće se pažnja na to da gledajući tekst programa ne može da se kaže kog će tipa biti taj iterotor. To zavisi od tipa zbirke na koju pokazuju pokazivač `pz` u momentu pozivanja polimorfne metode `iter()`.

Na kraju, pošto je svesno izostavljena naredba `break` iz opisane sekvence radnji, program se produžuje naredbama iz oznake `case '5'` i na taj način, bez novog zahteva korisnika, automatski ispisuje i sadržaj novostvorene zbirke. Rešenja u kojima se u naredbi `switch` delovi obrada delimično preklapaju, treba izbegavati, jer čine program teško razumljivim!

Rezultat 5.3 prikazuje primer rada programa za obradu zbirki geometrijskih figura. Iz potpunog ispisa na ekranu izostavljeno je ispisivanje menija na početku svake operacije.

5.11.2 Obrada zbirki geometrijskih figura u ravni

```
// Kopiranje odabrane figure na kraj zpirke.
case '2': { int ind; cout << "Redni broj figure? "; cin >> ind;
    if (ind >= 0 && ind < *pz) *pz += *(pz)|ind;
    else cout << "*** Neispravan indeks!\n\a";
    break;
}

// Prikazivanje odabrane figure.
case '3': { int ind; cout << "Redni broj figure? "; cin >> ind;
    if (ind >= 0 && ind < *pz)
        cout << " " << *(pz)[ind] << endl;
    else cout << "*** Neispravan indeks!\n\a";
    break;
}

// Promena vrste zbirke.
case '4': { Zbirka* ppz = stvori ();
    if (!ppz) { cout << "*** Neispravna vrsta!\n\a"; break; }
    for (pi->poc(); pi->ima(); *ppz += ~*pi);
    delete pz; pz = ppz;
    delete pi; pi = pz->iter ();
}

// Prikazivanje sadržaja zbirke.
case '5': cout << *pz; break;

// Pražnjenje zbirke.
case '6': ~*pz; pi->poc(); break;

// Početak novih iteracija.
case '7': pi->poc(); break;

// Prelazak na sledeći element zbirke.
case '8': ++*pi;

// Prikazivanje tekućeg elementa zbirke.
case '9': if (pi->ima()) cout << " " << *pi << endl;
    else cout << "*** Nema tekuce figure!\n\a";
    break;

// Brisanje tekuće figure iz zbirke.
case '0': if (pi->ima()) delete ~*pi;
    else cout << "*** Nema tekuce figure!\n\a";
    break;

// Završetak rada.
case 'K': case 'k': dalje = false; break;

// Poruka o neispravnom izboru.
default : cout << "*** Neispravan izbor!\n\a";
}
```

Program 5.22 – Ispitivanje klase za zbirke figura (zbirkat.C, drugi deo)

```
§ CC zbirkat zbirka niz2 niter lista krug2 kvadrat trougao
§ zbirkat
```

Moguce operacije su:

- | | |
|----------------------------|----------------------------|
| 1 Citaj figuru i stavi | 7 Pocni iteracije |
| 2 Dohvati figuru i stavi | 8 Idi na sledecu i prikazi |
| 3 Dohvati figuru i prikazi | 9 Prikazi tekucu figuru |
| 4 Promeni vrstu zbirke | 0 Brisi tekucu figuru |
| 5 Prikazi zbirku | k Zavrsi s radom |
| 6 Prazni zbirku | |

Unesite svoj izbor: 1

Vrsta (o, k, t)? o

x, y, r? 1 2 3

Unesite svoj izbor: 1

Vrsta (o, k, t)? t

x, y, a, b, c? 7 8 3 4 5

Unesite svoj izbor: 1

Vrsta (o, k, t)? k

x, y, a? 9 8 2

Unesite svoj izbor: 5

Niz!

```
krug [T=(1,2), r=3, O=18.8496, P=28.2743]
trougao [T=(7,8), a=3, b=4, c=5, O=12, P=6]
kvadrat [T=(9,8), a=2, O=8, P=4]
```

Unesite svoj izbor: 4

Vrsta (n,l)? 1

Lista!

```
krug [T=(1,2), r=3, O=18.8496, P=28.2743]
trougao [T=(7,8), a=3, b=4, c=5, O=12, P=6]
kvadrat [T=(9,8), a=2, O=8, P=4]
```

Unesite svoj izbor: 7

Unesite svoj izbor: 9

```
krug [T=(1,2), r=3, O=18.8496, P=28.2743]
```

Unesite svoj izbor: 8

```
trougao [T=(7,8), a=3, b=4, c=5, O=12, P=6]
```

Unesite svoj izbor: 0

Unesite svoj izbor: 9

```
kvadrat [T=(9,8), a=2, O=8, P=4]
```

Unesite svoj izbor: 5

Lista!

```
krug [T=(1,2), r=3, O=18.8496, P=28.2743]
kvadrat [T=(9,8), a=2, O=8, P=4]
```

Unesite svoj izbor: 2

Redni broj figure? 1

Unesite svoj izbor: 5

Lista!

```
krug [T=(1,2), r=3, O=18.8496, P=28.2743]
kvadrat [T=(9,8), a=2, O=8, P=4]
kvadrat [T=(9,8), a=2, O=8, P=4]
```

Unesite svoj izbor: 3

Redni broj figure? 0

```
krug [T=(1,2), r=3, O=18.8496, P=28.2743]
```

Unesite svoj izbor: k

Rezultat 5.3 – Obrada zbirki geometrijskih figura programom 5.21

6 Izuzeci

Bez posebne podrške od strane programskog jezika konfliktne situacije (greške) u programima se obično obrađuju na sledeći način:

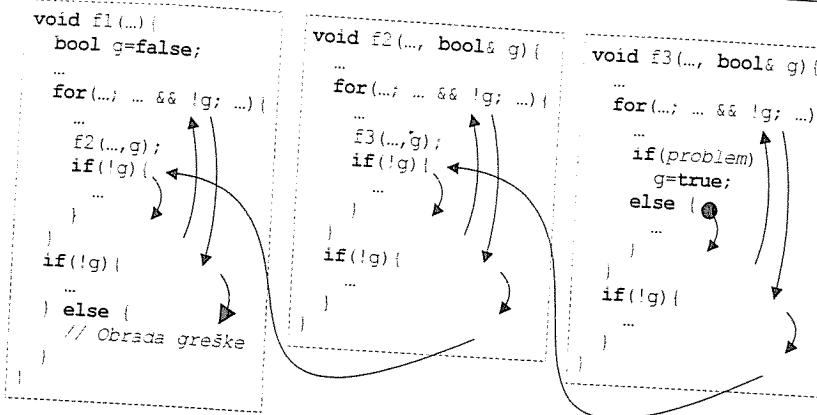
- Ako se unutar neke funkcije otkrije greška funkcija se završi vraćajući neku informaciju o tome pozivajućoj funkciji. Ta informacija obično bude neka celobrojna šifra, koja se često vraća kao vrednost funkcije.
- Ako ni funkcija koja dobija informaciju o grešci nije u stanju da razreši nastalu situaciju prosleduje šifru greške funkciji koja je nju pozvala. Ako se na taj način stiže do glavne funkcije prekida se izvršavanje celog programa.
- Kada se stigne do funkcije koja ume da razreši nastali problem ona preduzme odgovarajuće korektivne akcije i izvršavanje programa se nastavi dalje.

Glavni problem je što su sve funkcije složenog programskog sistema opterećene brigom *Da li je negde otkrivena greška?*, a ne samo funkcija u kojoj ta greška može da nastane.

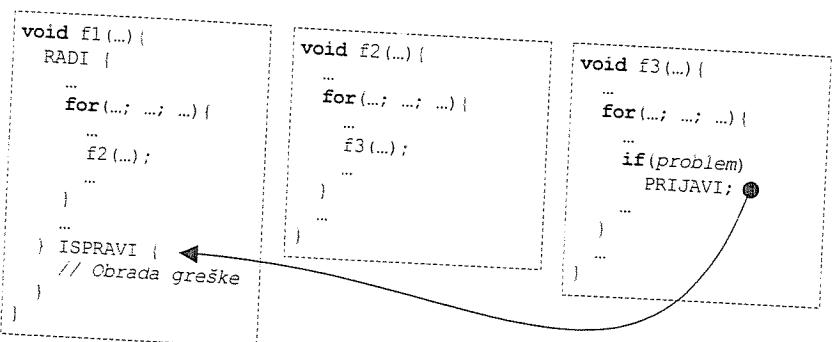
Na slici 6.1 prikazane su tri funkcije od kojih f1 poziva f2 i f2 poziva f3. Greška se otkrije u funkciji f3 koju može da obrađuje funkcija f1. Prenos upravljanja od mesta otkrivanja greške do mesta obrade greške omogućen je indikatorom g čija vrednost true označava pojavu greške. Vrednost tog indikatora mora svaki čas da se proverava. Strelice na slici 6.1 označavaju putanje od mesta otkrivanja greške do mesta njene obrade.

Jezik C++ nudi efikasan mehanizam **izuzetaka** za obradu grešaka koji rasterećuje programera od većine, ako ne i od svih, navedenih problema. Kad se na nekom mestu u programu otkrije greška, potrebno je samo da se ona prijavi odgovarajućim **izuzetkom**. Bezbedno prekidanje započetih aktivnosti i predaja upravljanja nadležnom **rukovaocu izuzecima** dešava se automatski.

Na slici 6.2 prikazane su funkcije sa slike 6.1 za slučaj upotrebe mehanizma izuzetaka. Pojedine funkcije se programiraju za slučaj da je sve u redu i nigde se ne postavlja pitanje da li je otkrivena greška. Jedino se funkcija f1, koja je u stanju da obradi greške, deli na dva dela. Prvi deo (blok RADÍ) odnosi se na obradu kada je sve u redu, a drugi deo (blok ISPRAVI) sadrži korake za obradu otkrivene greške. Kada se u funkciji f3 otkrije greška, njenim prijavljivanjem (naredba PRIJAVI) upravljanje se prenosi neposredno na blok ISPRAVI u funkciji f1, pri čemu se funkcija f2 u potpunosti zaobilazi. Treba uočiti da su time prekinute sve do tada započete aktivnosti (ciklusi i funkcije).



Slika 6.1 – Obrada greške pomoću indikatora



Slika 6.2 – Mechanizam izuzetaka

Izuzeci u jeziku C++ se predstavljaju podacima standardnih ili klasnih tipova koji se dostavljaju rukovaocu izuzecima. Tip podatka koji se šalje rukovaocu naziva se i **tip izuzetka**.

Može da se definiše veći broj rukovalaca, za svaki tip izuzetaka po jedan. Rukovalac koji će se pozvati bira se na osnovu tipa nastalog izuzetka.

6.1 Rukovanje izuzecima

Rukovanje izuzecima definiše se naredbom **try** čiji je opšti oblik:

```

try blok catch ( tip identifikator ) telo
      catch ( tip identifikator ) telo
      ...
      catch ( tip identifikator ) telo
      catch ( ... )           telo

```

6.1 Rukovanje izuzecima

Blok predstavlja složenu naredbu u toku čijeg izvršavanja mogu da se pojave izuzeci. Iza **bloka** sledi niz definicija rukovalaca tim izuzecima.

Definicije rukovalaca liče na definicije funkcija. Umesto identifikatora funkcije koristi se službena reč **catch**. Postoji tačno jedan parametar, a **telo** predstavlja složenu naredbu koja čini sadržaj rukovaoca.

Parametar se sastoji od oznake **tipa** i **identifikatora** parametra. Koristi se za dostavljanje informacija o nastalom izuzetku. Ukoliko se parametar ne koristi u telu rukovaoca, **identifikator** može da se izostavi.

Kaže se da je rukovalac tipa T ako je njegov parametar tipa T. Rukovalac tipa T služi za prihvatanje i obradu izuzetaka tipa T. Tri tačke (...) umesto parametra označavaju da rukovalac može da prihvati izuzetke svih tipova. Ovakav rukovalac naziva se **univerzalni rukovalac**.

Obrada definisana **blokom** može da bude proizvoljne složenosti uključujući i pozive funkcija do proizvoljne dubine. Izuzeci mogu da se pojave i unutar tih funkcija. Ne pravi se razlika između izuzetaka koji se pojave neposredno u **bloku** ili pak u nekoj od pozvanih funkcija.

Unutar **bloka**, neposredno ili unutar pozivanih funkcija, mogu da se nalaze i naredbe **try**. Tada se govori o uklapanju naredbi **try**.

Izvršavanje naredbe **try** počinje izvršavanjem **bloka**. Ako se do napuštanja **bloka** ne pojavi izuzetak, skokom na kraj naredbe **try** preskaču se svi rukovaoci izuzecima. Pojavi li se neki izuzetak unutar **bloka**, prekida se njegovo izvršavanje i odabere se rukovalac čiji tip se slaže s tipom nastalog izuzetka. Posle izvršavanja **tela** tog rukovaoca skoči se na kraj naredbe **try**.

Ovo je pojednostavljeni tok događaja. Detaljniji opis dat je u odeljku 6.3.

Evo primera naredbe **try**:

```

try {                                // Obrada gde se očekuje pojava izuzetaka.
    ...
    radi ( ... );
    ...
} catch (const char* pz) { // Obrada izuzetaka tipa niski.
    ...
} catch (int i) {                  // Obrada izuzetaka celobrojnog tipa.
    ...
} catch (...) {                   // Obrada izuzetaka proizvoljnih tipova.
    ...
}

```

6.2 Prijavljivanje izuzetaka

Pojava izuzetka se prijavljuje izrazom oblika:

```
throw izraz
```

gde vrednost **izraza** predstavlja informaciju koja se dostavlja rukovaocu izuzecima i čiji tip određuje kom rukovaocu se ta informacija dostavlja. Tip **izraza** predstavlja tip izuzetka. Vrednost **izraza** predstavlja dodatne informacije o nastalom problemu. U slučaju klasnih tipova može i velika količina informacija da se dostavi rukovaocu izuzecima.

Operator `throw` je unarni prefiksni operator prioriteta 2. Tip rezultata celokupnog izraza s operatom `throw` je `void`, tj. nema rezultata. Zbog toga izraz ne sme da se koristi kao operand drugih operatora, izuzev operatora `zarez (,)` i operatora uslovnog izraza `(? :)`.

Izuzetak može da se prijavljuje neposredno unutar bloka naredbe `try` (videti prethodni odeljak) i unutar bilo koje funkcije koja se neposredno ili posredno poziva iz bloka. U obzir dolaze kako obične funkcije tako i metode klase, uključujući i konstruktoare, destruktore i operatorske funkcije.

U deklaraciji funkcija može da se navede spisak tipova izuzetaka koje data funkcija može da prijavljuje stavljanjem:

```
throw ( niz_identifikatora )
```

ispred tela funkcije (ispred zagrade `{`) u slučaju definicije, odnosno ispred tačka-zareza `(;)` u slučaju deklaracije. *Niz identifikatora* predstavlja tipove izuzetaka, međusobno razdvojenih zarezima, koji mogu da se prijave. Ako unutar para zagrada ne postoji nijedan identifikator funkcija ne može da prijavljuje nijedan izuzetak. Skup mogućih tipova izuzetaka u svakoj deklaraciji i u definiciji date funkcije mora da bude isti.

Greška je ako funkcija za koju je naveden spisak mogućih izuzetaka prijavljuje izuzetak koji nije na spisku. Funkcije za koje ništa nije rečeno o mogućim izuzecima smiju da prijavljuju izuzetke bilo kog tipa.

Virtuelna (polimorfna) metoda u izvedenoj klasi ne sme da prijavljuje nijedan izuzetak čiji tip nije predviđen u deklaraciji te metode u osnovnoj klasi. Naravno, skup mogućih tipova izuzetaka u izvedenoj klasi može da bude uži nego u osnovnoj klasi. Ako u osnovnoj klasi nema ograničenja za moguće tipove izuzetaka, u izvedenoj klasi još uvek može da postoji ograničenje.

U slučaju uklapanja naredbi `try` rukovaoci izuzecima unutrašnje naredbe `try` takođe mogu da prijave izuzetke. Ti izuzeci se prijavljuju spoljašnjoj naredbi `try`.

Unutar rukovalaca izuzecima iza operatora `throw` sme da se izostavi *izraz*. U tom slučaju izuzetak za koji je pozvan posmatrani rukovalac prosleđuje se odgovarajućem rukovaocu spoljašnje naredbe `try`. Ako je parametar rukovaoca pokazivač ili upućivač eventualne izmene vrednosti pokazanog ili upućivanog podatka, učinjene u unutrašnjem rukovaocu, videće se u spoljašnjem rukovaocu ako se izuzetak prosleđuje na viši nivo.

Evo primera funkcije koja prijavljuje izuzetke:

```
void radi ( ... ) throw (char*, int) {
    ...
    if ( ... ) throw "Upomoc!";           // Izuzetak tipa char*.
    ...
    if ( ... ) throw 55;                  // Izuzetak tipa int.
    ...
    if ( ... ) throw Tacka (3, 5);       // GREŠKA: tip Tacka nije predviđen.
    ...
}
```

Ovde je pretpostavljeno da je `Tacka` klasa čiji se konstruktor poziva za stvaranje objekta koji se prosleđuje rukovaocu izuzecima. Pošto u spisku mogućih izuzetaka na početku nije naveden tip `Tacka`, prijavljivanje izuzetka tog tipa nije dozvoljeno, bez obzira na činjenicu što naredba `try` u primeru iz prethodnog odeljka sadrži rukovaoca koji bi mogao da prihvati taj izuzetak. To je poslednji rukovalac koji može da prihvata sve izuzetke, bez obzira na njihove tipove.

6.3 Prihvatanje izuzetaka

Rukovalac tipa `R` može da prihvati izuzetak tipa `I` ako:

- `R` i `I` su istih tipova,
- `R` je pristupačna (javna) osnovna klasa za klasu `I` na mestu prijavljivanja izuzetka, ili
- `R` i `I` su pokazivački ili upućivački tipovi i `I` može standardnim konverzijama da se pretvori u `R` na mestu prijavljivanja izuzetka.

Na mestu prijavljivanja izuzetka prvo se stvara kopija podatka (objekta) koji je rezultat izraza iza operatora `throw`. Tako dobijeni privremeni podatak prosleđuje se „najbližem“ rukovaocu koji može da prihvati dati izuzetak. Skreće se pažnja na to da ako se za prijavljivanje izuzetaka koriste objekti u odgovarajućoj klasi mora da postoji javan konstruktor kopije (automatski generisani ako to zadovoljava, ili specijalno pravljeni).

„Najbliže“ u ovom slučaju znači prvi rukovalac po redosledu njihovog navođenja unutar naredbe `try` u čijem bloku je prijavljen izuzetak. U slučaju uklapljenih naredbi `try`, prvo se posmatraju rukovaoci najdublje naredbe `try`. Ako nijedan od njih ne može da prihvati dati izuzetak, kandidat se traži među rukovaocima naredbe `try` na prvom višem nivou uklapanja. Ako se za dati izuzetak ne pronalazi odgovarajući rukovalac izuzecima ni u naredbi `try` na najvišem nivou, program se prekida.

Prilikom navođenja rukovalaca izuzecima treba se držati sledećih pravila:

- Rukovaoce tipa izvedene iz neke osnovne klase treba stavljati ispred rukovaoca tipa te osnovne klase. Uloga rukovaoca tipa osnovne klase je da prihvati sve izuzetke tipa izvedenih klasa iz zajedničke osnovne klase za koje ne postoje zasebni rukovaoci.
- Univerzalnog rukovaoca (s parametrom `...`) treba stavljati na poslednje mesto. Njegova uloga je da prihvati sve izuzetke, bez obzira na njihove tipove, za koje ne postoje zasebni rukovaoci.

Predaja upravljanja rukovaocu izuzecima uzrokuje napuštanje kako bloka naredbe `try`, tako i svih blokova u koje se ušlo kao posledica uklapanja upravljačkih struktura (sekvenci, selekcija i/ili ciklusa) i pozivanja funkcija. Kako napuštanje datog bloka podrazumeva uništavanje svih podataka (objekata) koji su lokalni za taj blok, u ovom slučaju potrebno je uništiti sve podatke (objekte) koji su lokalni za sve blokove u koje se ušlo od početka posmatrane naredbe `try`.

Objekti se, pomoću odgovarajućih destruktora, uništavaju po obrnutom redosledu njihovog stvaranja. Prvo se uniše svi objekti koji su stvoreni unutar najdubljeg bloka. Posle objekti unutar bloka na prvom višem nivou. Na kraju ostaju objekti iz samog bloka tekuće naredbe `try`.

Ako se izuzetak prijavljuje u konstruktoru uništavaće se samo do tog momenta stvoreni podobjekti. To znači pozivanje destruktora za ona polja klasnih tipova i osnovne klase za koje je konstruktor izvršen do kraja. Ako se to desi u toku inicijalizacije nekog elementa niza, posle primene prethodnog postupka na taj element, pozivaće se destruktur samo za elemente niza koji su do tog momenta bili uspešno inicijalizovani.

Podatak koji se pojavi kao operand pri prijavljivanju izuzetka najverovatnije je lokalan za najdublji blok i biće uništen među prvima. Zato se rukovaocu prosleđuje privremeni podatak koji je kopija tog operanda. Privremeni podatak biće uništen tek po napuštanju `try`

rukovaoca koji je prihvatio prijavljeni izuzetak. Ako se izuzetak prosleduje rukovaocu naredbe `try` na višem nivou (pomoću `throw` bez `izraza`), uništavanje privremenog podatka odlaže se do napuštanja tog rukovaoca.

Operand operatora `throw` ne sme da bude pokazivač ili upućivač na prolazan podatak koji je lokalan za bilo koji blok koji će biti napušten u toku predaje upravljanja odgovarajućem rukovaocu izuzecima. Taj podatak biće uništen pre nego što rukovalac bude mogao da ga koristi!

6.4 Neprihváčeni i neočekívani izuzeci △

Ako se za neki izuzetak ne pronađe rukovalac koji može da ga prihvati, izvršava se sistemska funkcija čiji je prototip:

```
void terminate();
```

Dok se drugačije ne kaže, ona poziva sistemsku funkciju `abort()` koja nasilno prekida program i vraća se na operativni sistem računara.

Drugačiji završetak programa može da se postigne pozivanjem sistemske funkcije:

```
typedef void (*PF) ();
PF set_terminate(PF pf);
```

gde je `pf` pokazivač na funkciju bez parametara, koja ne daje nikakvu vrednost funkcije (tip `void`). Ona predstavlja novu funkciju koju treba da pozove funkcija `terminate()` umesto funkcije `abort()`. Vrednost funkcije `set_terminate()` je pokazivač na staru funkciju koja je bila predviđena za završavanje programa u posmatranoj situaciji.

Programerova funkcija (`*pf`) posle izvršavanja radnji koje programer smatra za potrebne, mora da dovede do završetka programa i povratka na operativni sistem (pozivanjem funkcije `exit(int)`, `terminate()` ili `abort()`). Pokušaj drugačijeg završetka programa (`*pf` na primer, naredbom `return`), doveće do nasilnog prekida programa pozivanjem funkcije `abort()`.

Ako neka funkcija prijavi izuzetak koji se ne nalazi na spisku predviđenih za tu funkciju, izvršava se sistemska funkcija čiji je prototip:

```
void unexpected();
```

Dok se drugačije ne kaže ona poziva sistemsku funkciju `terminate()` koja dovodi do završetka programa i povratka na operativni sistem računara.

Drugačiji završetak programa može da se postigne pozivanjem sistemske funkcije:

```
typedef void (*PF) ();
PF set_unexpected(PF pf);
```

gde je `pf` pokazivač na funkciju bez parametara koja ne daje nikakvu vrednost funkcije (tip `void`). Ona predstavlja novu funkciju koju treba da pozove funkcija `unexpected()` umesto funkcije `terminate()`. Vrednost funkcije `set_unexpected()` je pokazivač na staru funkciju koja je bila predviđena za završavanje programa u posmatranoj situaciji.

Programerova funkcija (`*pf`) posle izvršavanja radnji koje programer smatra za potrebne, može da uradi sledeće:

- prijavi izuzetak podatkom čiji se tip nalazi u spisku dozvoljenih tipova izuzetaka za datu funkciju;
- prijavi izuzetak objektom tipa `bad_exception`;
- završi program pozivanjem funkcije `exit(int)`, `terminate()` ili `abort()`.

Pokušaj drugačijeg završetka funkcije `*pf` doveće do nasilnog prekida programa pozivanjem funkcije `abort()`.

Funkcije `terminate()`, `set_terminate()`, `unexpected()`, `set_unexpected()` i klasa `bad_exception` opisane su u standardnom zaglavljtu `<exception>`. Funkcija `abort()` je opisana u standardnom zaglavljtu `<cstdlib>`, odnosno `<stdlib.h>`.

6.5 Standardni izuzeci △

Klase svih izuzetaka koje prijavljuju neki operatori jezika C++ ili metode standardnih klasa su izvedene (neposredno ili u više koraka) iz zajedničke osnovne klase `exception` čija definicija, u standardnom zaglavljtu `<exception>`, je:

```
class exception {
public:
    exception () throw ();
    exception (const exception&) throw ();
    exception& operator= (const exception&) throw ();
    virtual ~exception () throw ();
    virtual const char* what() const throw ();
private:
    ...
};
```

Klase poseduje sve metode koje se smatraju obaveznim (podrazumevani konstruktor, konstruktor kopije, dodelu kopije i virtuelni destruktur) i jednu virtuelnu metodu koja vraća pokazivač na tekstualni opis izuzetka. Nijedna od metoda ne prijavljuje nijedan izuzetak (`throw()`).

Standard ne propisuje tekst poruka za standardne izuzetke.
Mada nije obavezno, preporučuje se da programeri za prijavljivanje svojih izuzetaka projektuju klase koje su izvedene iz klase `exception`. To omogućava da se svi izuzeci mogu obraditi zajedničkim rukovaocem, ali da to ne bude univerzalni rukovalac koji nikakvu informaciju ne može da preuzme iz objekta kojim je izuzetak prijavljen.

Vešto izgrađenom hijerarhijskom strukturu klasa za izuzetke omogućava se obrada izuzetaka pojedinačno, zajedno u manjim ili većim grupama, ali i sve zajedno.

6.6 Zadaci

6.6.1 Obrada vektora zadatih opsega indeksa

Zadatak:

Napisati na jeziku C++ klasu za obradu vektora realnih brojeva zadatih opsega indeksa. Za razrešavanje konfliktnih situacija koristiti mehanizam rukovanja izuzecima. Napisati na jeziku C++ program za prikazivanje nekih mogućnosti te klase.

Rešenje:

Vektori su jednodimenzionalni nizovi. U većini viših programskih jezika opseg mogućih vrednosti indeksa može da se navede prilikom definisanja niza. To nije slučaj u jeziku C++, ali postoji mogućnost ostvarivanja takvih vektora u obliku klase.

Program 6.1 prikazuje definiciju klase `Vekt` koja ostvara vektore realnih elemenata, za koje je opseg indeksa parametar inicijalizacije primeraka klase.

// Definicija klase vektora realnih brojeva (Vekt).

```
class Vekt {
    // Polja:
    int min, max; // granice indeksa,
    int duz; // broj elemenata,
    double* niz; // niz elemenata.
    // Pomoćne metode:
    // stvaranje vektora,
    void pravi (int poc, int kra); // kopiranje u vektor,
    void kopiraj (const Vekt& v); // uništavanje vektora.
    void brisi ();

public:
    enum Greska { OPSEG, // Kodovi grešaka:
        PRAZAN, // neispravan opseg indeksa,
        INDEKS, // vektor je prazan,
        DUZINA }; // indeks je izvan opsega,
        // neusaglašene dužine vektora.
    // Konstruktori:
    Vekt () { niz = 0; } // prazan niz,
    explicit Vekt (int kra) { pravi(1,kra); } // podrazumevani početni indeks,
    Vekt (int poc, int kra) { pravi(poc,kra); } // sa zadatim indeksima,
    Vekt (const Vekt& v) { kopiraj (v); } // kopije.
    ~Vekt () { brisi (); } // Destruktor.
    Vekt& operator= (const Vekt& v); // Dodela vrednosti.
    double& operator[] (int ind); // Pristup elementu.
    const double& operator[] (int ind) const;
    friend double operator* (const Vekt& v, // Skalarni proizvod.
        const Vekt& w);
    int min_ind () const { return min; } // Najmanji indeks.
    int max_ind () const { return max; } // Najveći indeks.
};
```

Program 6.1 – Definicija klase vektora (vekt.h)

Polja klase su najmanja (`min`) i najveća (`max`) dozvoljena vrednost indeksa, broj elemenata vektora (`duz`) i pokazivač na same elemente (`niz`) u dinamičkoj zoni memorije. Polje `duz` nije neophodan (jednak je `max-min+1`). predviđen je da na nekim mestima pojednostavi program.

Javno nabranjanje `Greska` uvodi simboličke oznake za izuzetke (greške) koji mogu da prijavljuju funkcije za obradu vektora. Skreće se pažnja na to da su u jeziku C++ nabranjanja zasebni tipovi, pa rukovalac za prihvatanje izuzetaka treba da ima parametar tipa `Vekt::Greska`, a ne tip `int`.

Podrazumevani konstruktor je predviđen kao garancija da preostale funkcije uz klasu `Vekt` (metode ili prijateljske funkcije) sigurno zateknu „inteligentne” podatke u primerci-

6.6.1 Obrada vektora zauzeta vrednost

ma klase. To može da bude i prazan vektor (prepoznaće se po `niz==0`), kao znak da primjerak nije ni eksplicitno inicijalizovan, niti mu je dodeljena vrednost.

Javne metode koriste tri pomoćne (privatne) metode za izvođenje radnji koje se javljaju kao njihovi delovi. Uvedene su radi izbegavanja ponavljanja istih sekvenci naredbi.

Jednostavnije metode, koje ne sadrže cikluse, definisane su unutar definicije klase. Program 6.2 prikazuje ostvarenje funkcija koje podržavaju klasu `Vekt`.

// Definicije metoda i funkcija uz klasu Vekt.

```
#include "vekt.h"

// Stvaranje vektora.
void Vekt::pravi (int poc, int kra) {
    if ((min=poc) > (max=kra)) throw OPSEG;
    niz = new double [duz=kra-poc+1];
    for (int i=0; i<duz; niz[i++]=0);
}

void Vekt::kopiraj (const Vekt& v) { // Kopiranje vektora.
    if (v.niz) {
        niz = new double [duz=(max=v.max)-(min=v.min)+1];
        for (int i=0; i<duz; i++) niz[i] = v.niz[i];
    } else niz = 0;
}

void Vekt::brisi () { delete [] niz; niz = 0; } // Uništavanje vektora.

Vekt& Vekt::operator= (const Vekt& v) { // Dodela vrednosti.
    if (this != &v) { brisi (); kopiraj (v); }
    return *this;
}

double& Vekt::operator[] (int ind) { // Pristup elementu.
    if (! niz) throw PRAZAN;
    if (ind<min || ind>max) throw INDEKS;
    return niz[ind-min];
}

const double& Vekt::operator[] (int ind) const
    { return (const_cast<Vekt&>(*this))[ind]; }

double operator* (const Vekt& v, const Vekt& w) { // Skalarni proizvod.
    if (! v.niz || ! w.niz) throw Vekt::PRAZAN;
    if (v.duz != w.duz) throw Vekt::DUZINA;
    double s = 0;
    for (int i=0; i<v.duz; i++) s += v.niz[i] * w.niz[i];
    return s;
}
```

Program 6.2 – Definicije metoda i funkcija uz klasu Vekt (vekt.C)

Pomoćna (privatna) metoda `pravi()` stvara vektor od odgovarajućeg broja elemenata i nultim početnim vrednostima. To podrazumeva i dodelu memorije u dinamičkoj zoni. Ovu metodu pozivaju konstruktori s jednim i s dva celobrojna parametra, koji predstavljaju granične vrednosti indeksa za stvarani vektor.

U toku stvaranja vektora mogu da se dese dve greške. Opseg vrednosti indeksa je neispravan ako je najmanja vrednost (`pos`) veća od najveće vrednosti (`kra`). Drugo, može da se desi da u dinamičkoj zoni memorije nema dovoljno mesta za smeštanje elemenata vektora. Prvi izuzetak se prijavljuje pomoću `throw OPSEG`. Skreće se pažnja na to da ako se dode do izraza `throw` metoda se prekida i ne stiže se do naredbe koja se nalazi iza naredbe `if`. Zato u naredbu `if` nije stavljena deo `else`. Prijavljanje neuspeha pri dinamičkoj do-

stvaranju kopije vektora u koji je pravi parametar tih metoda u tekućem objekat `*this`.

Ako kopirani vektor nije prazan potrebno je dodeliti memoriju za niz iste veličine kao što je veličina niza u kopiranom vektoru. Posle dodelje memorije ostaje samo da se prekopiraju elementi izvorišnog vektora. Sva prsta polja su već ranije popunjena odgovarajućim vrednostima kao bočni efekti odgovarajućih izraza.

Pošto se kopira postojeći niz ispravnost opsega indeksa ne treba da se proverava. Eventualni problem s dinamičkom dodelom memorije prijavaće operator `new`.

Ako je izvorišni vektor prazan samo je potrebno pokazivač `niz` postaviti na nulu radi kasnijeg bezbednog korišćenja i uništavanja vektora.

Prilikom uništavanja vektora u privatnoj metodi `brisi()` ne može da se desi greška. Čak i ako je vektor prazan, primena operatorka `delete` je bezopasna pošto je u tom slučaju `niz==0`. To je obezbeđeno postojanjem podrazumevanog konstruktora. Za dalji tok programa je važno da prethodna relacija važi po završetku funkcije `brisi()`. Tada je vektor, čak i ako ga pre toga nije bio, sigurno prazan.

Korišćenjem prethodnih pomoćnih metoda operatorska funkcija za dodelu vrednosti `operator=()` postala je krajne jednostavna. Svi eventualni izuzeci prijavljuju se u tim metodama, i na njih ne treba posebno obratiti pažnju. Ako se unutar tih pomoćnih metoda desi neka greška uopšte se ne vraća u operatorsku funkciju `operator=()`. Umesto toga, neposredno će se pozvati odgovarajući rukovalac izuzecima.

Prilikom pristupa datom elementu niza (indeksiranju), greška je ako je taj niz prazan ili ako je indeks traženog elementa izvan dozvoljenog opsega za dati primerak klase `Vekt`. Te situacije u operatorskoj funkciji `operator[]()` prijavljuju se kao izuzeci sa šiframa PRAZAN, odnosno INDEKS.

Ako nema greške, vrednost indeksa `ind` treba preslikati iz opsega od `min` do `max` (koji koristi klasa `Vekt`) na opseg od 0 do `duz-1` (koji koristi jezik C++ za nizovne tipove podataka).

Skreće se pažnja na to da je za tip rezultata funkcije za indeksiranje predviđen upućivač na realne brojeve (`double&`). To omogućava korišćenje indeksiranja i na levoj strani operatora za dodelu vrednosti (na primer: `v[6]=1.25`).

Kao i u ranijim zadacima, i ovde postoje dve varijante operatorske funkcije `operator[]()` radi sprečavanja promene vrednosti odabranog elementa u nepromenljivom vektoru posle povratka iz te metode. Prva varijanta se koristi za promenljive, a druga za nepromenljive vektore. Druga varijanta je ostvarena pozivanjem prve varijante.

Na kraju, prilikom izračunavanja skalarnog proizvoda dva vektora greška je ako je bar jedan od vektora prazan, ili ako vektori nisu iste dužine. Te situacije se u operatorskoj funkciji `operator*()` prijavljuju kao izuzeci sa šiframa PRAZAN, odnosno DUZINA. Pošto je funkcija `operator*()` prijateljska funkcija, a ne metoda klase, neophodno je navesti i klasu kojoj pripadaju te simboličke konstante (`Vekt::`).

6.6.1 Obrada vektora zadatih opsega indeksa

U priloženom rešenju je uzeto da nije bitan opseg vrednosti indeksa jednog i drugog vektora. Bitno je samo da su njihove dužine jednakе. Kao uslov za mogućnost izračunavanja skalarnog proizvoda dva vektora moglo je da se uzme da jednakost dužina nije dovoljna, već da i opsezi vrednosti indeksa moraju da budu isti.

Skreće se pažnja na to da, pošto su svi izuzeci u klasi `Vekt` međusobno jednakog tipa mogu da se prihvataju samo u jednom zajedničkom rukovaocu izuzecima. Šta je tačno problem, može da se zaključi na osnovu vrednosti podatka koji je kao parametar dostavljen rukovaocu.

Program 6.3 prikazuje primer za korišćenje klase `Vekt`.

Program za ispitivanje klase `Vekt`.

```
#include "vekt.h"
#include <iostream>
#include <new>
using namespace std;

int main () {
    while (true) {
        try {
            int min, max;
            cout << "Opseg indeksa prvog vektora: "; cin >> min >> max;
            if (min==0 & max==0) break;
            Vekt v1 (min, max); cout << "Elementi prvog vektora: ";
            for (int i=min; i<=max; i++) cin >> v1[i];
            cout << "Opseg indeksa drugog vektora: "; cin >> min >> max;
            Vekt v2 (min, max); cout << "Elementi drugog vektora: ";
            for (int i=min; i<=max; i++) cin >> v2[i];
            cout << "Skalarni proizvod = " << v1 * v2 << "\n\n";
        } catch (Vekt::Greska g) {
            char* poruke[] = { "Neispravan opseg indeksa",
                "Vektor je prazan",
                "Indeks je izvan opsega",
                "Neusaglasene duzine vektora" };
            cout << "\n*** " << poruke[g] << "!\a\n\n";
        } catch (bad_alloc) {
            cout << "\n*** Dodela memorije nije uspeila!\a\n\n";
        }
    }
}
```

Program 6.3 – Ispitivanje klase `Vekt` (vektt.C)

Program čita parove nizova, izračunava njihov skalarni proizvod i ispisuje dobijeni rezultat sve dok za obe granice opsega vrednosti indeksa prvog vektora ne pročita nulu.

Unutar ciklusa u programu 6.3 nalazi se naredba `try` s jednim rukovaocem izuzetaka. Koristan sadržaj nalazi se u prvom bloku naredbe. Ako je sve u redu, po dolasku na kraj bloka, rukovaoci izuzecima se preskaču.

Desi li se greška bilo gde unutar tog bloka, uključujući i unutrašnjost pozivanih funkcija, obrada se prekida i izvršava se jedan od dva rukovaoca izuzecima.

Prvi rukovalac je tipa `Vektor::Greska` i poziva se kada jedna od metoda klase `Vektor` prijavljuje izuzetak. Vrednost šifre izuzetka (parametra rukovaoca) koristi se kao indeks za odabiranje teksta čijim se ispisivanjem saopštava šta se desi. Ovakvo rešenje, kada korisnički program treba da zna tumačenje pojedinih šifara greške i sam da im pridružuje tekstove, nije baš preporučljivo. Nešto bolje rešenje bilo bi postojanje metode u klasi `Vektor` za preslikavanje šifre greške u odgovarajući tekst. Najbolje rešenje je, međutim, definisanje zasebne klase za greške čiji objekti u sebi sadrže tekst poruke koja odgovara dатој grešci.

Drugi rukovalac je tipa `bad_alloc` i poziva se kada operator `new` ne uspe da dodeli traženu količinu memorije u dinamičkoj zoni. Pošto se parametar rukovaoca ne koristi u telu rukovaoca, parametru nije dodeljeno ime.

Završetak izvršavanja bilo kojeg rukovaoca označava i kraj izvršavanja naredbe `try`. To znači, da bez obzira da li se desila greška ili ne, započinje se novi prolaz kroz ciklus programa 6.3. Izlazak iz tog ciklusa je moguć samo naredbom `break;`, koja se izvršava kada se u obe promenljive `min` i `max` pročita nula.

Rezultat 6.1 prikazuje primer rada programa 6.3.

```
% vektor
Opseg indeksa prvog vektora: 1 5
Elementi prvog vektora: 1 2 3 4 5
Opseg indeksa drugog vektora: -5 -1
Elementi drugog vektora: 5 4 3 2 1
Skalarni proizvod = 35

Opseg indeksa prvog vektora: 5 0
*** Neispravan opseg indeksa!

Opseg indeksa prvog vektora: 10 12
Elementi prvog vektora: 11 22 33
Opseg indeksa drugog vektora: -2 2
Elementi drugog vektora: 10 20 30 40 50

*** Neusaglasene duzine vektora!
Opseg indeksa prvog vektora: -2000000000 2000000000
*** Dodata memorije nije uspela!

Opseg indeksa prvog vektora: 0 0
```

Rezultat 6.1 – Obrada vektora programom 6.3

6.6.2 Izračunavanje određenog integrala

Zadatak:

- Napisati na jeziku C++ klase izračunavanje vrednosti funkcije i vrednosti neodređenog integrala funkcija $\sin x$, $e^{-0.1x} \sin x$, $\log x$ i polinoma $p(x)$. Za razrešavanje konfliktnih situacija napisati odgovarajuće klase. Napisati na jeziku C++ funkciju za računanje

6.6.2 Izračunavanje određenog integrala

određenog integrala realne funkcije s jednim realnim argumentom i program za prikazivanje mogućnosti projektovanog sistema klasa.

Rešenje:

Program 6.4 prikazuje definiciju klase opšte namene `Greska` za prijavljivanje izuzetaka, koja može da se koristi i samostalno, ali mogu iz nje da se izvode i neke složenije klase.

```
// Definicija klase za greške (Greska).

#ifndef _greska_h_
#define _greska_h_

#include <iostream>
#include <cstring>
using namespace std;

namespace Usluga {
    class Greska {
        char* poruka;
        void kopiraj (const char* p) {
            if (p) {
                poruka = new char [strlen(p)+1];
                strcpy (poruka, p);
            } else poruka = 0;
        }
        void brisi () { delete poruka; poruka = 0; } // Uništavanje objekta.
    public:
        Greska () { poruka = 0; } // podrazumevani,
        Greska (const char* por) { kopiraj (por); } // konverzije,
        Greska (const Greska& g) {kopiraj(g.poruka);} // kopije.
        virtual ~Greska () { brisi (); } // Destruktor.
        Greska& operator= (const Greska& g) { // Dodela vrednost.
            if (this != &g) { brisi (); kopiraj (g.poruka); }
            return *this;
        }
    protected:
        virtual void pisi (ostream& it) const { // Pisanje poruke o grešci.
            it << "\n*** " << (poruka ? poruka : "Greska") << " ***\n\a";
        }
        friend ostream& operator<< (ostream& it, const Greska& g)
            { g.pisi (it); return it; }
        bool ima_poruke() const {return poruka!=0;} // Ima li poruke u objektu?
    }; // class Greska
} // namespace Usluga

#endif
```

Program 6.4 – Definicija klase za greške (greska.h)

Pored osnovnog zadatka, u ovom rešenju prikazana je i upotreba prostora imena. Kako je klasa `Greska` upotrebljiva i u drugim oblastima, stavljena je u prostor imena `Usluga`. Sve ostale klase i funkcije, pošto su vezane za problematiku izračunavanja vrednosti funkcija, stavljene su u prostor imena `Funkcije` (videti preostale programe u ovom zadatku). Jedna od uloga prostora imena je i razvrstavanje klasa i globalnih funkcija prema

oblastima primene. Pošto prostori imena čine zasebne dosege, isti identifikator može da se koristi i u više prostora imena.

Jedino privatno polje klase `Greska` je pokazivač poruka na dinamički niz s tekstrom poruke kojom se opisuje nastali problem. Poruka može i da ne postoji, tada je vrednost pokazivača nula.

Privatne metode `kopiraj()` i `brisi()` su podrška konstruktoru konverzije, konstruktoru kopije, destruktoru i operatoru `=`. Skreće se pažnja na to da se u objekat greške uvek stavlja kopija izvorišnog teksta. Na taj način objekat greške postane nezavisan od okruženja, i kada se objekat bude uništavao moći će i tekst bez razmišljanja da se uništi. Destruktor je virtualan, da bi omogućio polimorfno uništavanje objekata tipa klasa koje su izvedene iz klase `Greska`.

U javnom delu klase `Greska` postoji još podrazumevani konstruktor koji omogućava stvaranje ispravnih i bezbednih praznih objekata. Ponekad ima smisla izuzetke prijavljivati i praznim objektima, koji ne sadrže nikakve dodatne podatke o problemu, osim da se problem pojavio.

Metoda `pisi()` u zaštićenom delu klase `Greska` omogućuje budućim izvedenim klasama da redefinišu način ispisivanja poruke o grešci. Ako se u izvedenoj klasi ništa ne preduzme posmatrana metoda će, uz dodatak ukrasnih zvezdica, proreda i zvučnog signala, ispisati poruku koja postoji u objektu, odnosno reč `Greska`, ako objekat ne sadrži poruku. Metoda `pisi()` je zaštićena, da bi iz izvedenih klasa mogla i neposredno da se poziva.

Za ispisivanje poruke o grešci iz bilo kog dela programa treba koristiti prijateljsku operatorsku funkciju `operator<<()`.

Kao usluga izvedenim klasama postoji još metoda `ima_poruke()` za ispitivanje da li u objektu postoji tekst poruke (pošto je pokazivač poruka privatno polje, njegova vrednost ne može neposredno da se ispituje iz izvedenih klasa).

Iz matematike se zna da neodređeni integral funkcije jednog realnog argumenta ne može uvek da se nađe u obliku algebarskog izraza. Kada takav izraz postoji određeni integral se računa vrlo jednostavno.

Program 6.5 prikazuje definiciju apstraktne klase za funkcije koja sadrži metode za računanje vrednosti funkcije i vrednosti neodređenog integrala (uzevši da je vrednost integracione konstante nula) za zadatu vrednost nezavisne promenljive.

Klasa `G_nema_integral` na početku programa 6.5 služi za prijavljivanje izuzetka ako ne postoji algebarski izraz za neodređeni integral date funkcije. Izvedena je iz klase `Greska` koja se nalazi u prostoru imena `Usluge`. Zbog toga identifikator `Greska` morala da se uveze u datotečki doseg naredbom `using`. Pošto je jedini identifikator iz tog prostora imena, nije morao da se uveze ceo prostor imena `Usluge`.

U klasi `G_nema_integral` postoji samo jedna metoda, podrazumevani konstruktor koji pomoću konstruktora osnovne klase `Greska`(`const char*`) inicijalizuje svaki objekat te klase istim tekstom. Ispisivanje te poruke će obavljati nasledena metoda `pisi()`, a posredstvom prijateljske operatorske funkcije `operator<<()`.

Apstraktna klasa `Fct` sadrži samo tri javne metode.

Virtuelni destruktur praznog tela je potreban zbog eventualnih izvedenih klasa koje koriste dinamičko dodeljivanje memorije.

Apstraktna metoda `operator()` predviđena je za izračunavanje vrednosti funkcije predstavljene datom klasom za vrednost nezavisne promenljive, koja je navedena kao parametar. Pošto je apstraktna metoda, mora da bude redefinisana u svim izvedenim klasa-

Definicija apstraktne klase za funkcije (`Fct.h`).

```
#ifndef _fct_h_
#define _fct_h_

#include "greska.h"
using Usluge::Greska;

namespace Funkcije {
    class G_nema_integral: public Greska { // KLASA ZA GREŠKU;
        public:
            G_nema_integral(): Greska ("Funkcija nema integral") {} // Konstruktor.

    }; // class G_nema_integral

    class Fct {
        public: // KLASA ZA FUNKCIJU;
            virtual ~Fct () {} // Virtuelni destruktur.
            virtual double operator() (double x) const =0; // Računanje funkcije.
            virtual double I (double) const // Računanje integrala.
            { throw G_nema_integral(); return 0; }
    }; // class Fct
} // namespace Funkcije

#endif
```

Program 6.5 – Definicija apstraktne klase za funkcije (`fct.h`)

ma iz klase `Fct`. Ako je `f` objekat klase koja je izvedena iz klase `Fct` ova metoda se poziva izrazom oblika `f(x)`, što je prirodan zapis za računanje vrednosti funkcije $f(x)$ u tački x .

Virtuelna metoda `I()` predviđena je za izračunavanje vrednosti neodređenog integrala funkcije predstavljene datom klasom, za vrednost nezavisne promenljive koja je navedena kao parametar. Da bi se olakšalo pisanje izvedenih klasa za funkcije za koje ne postoji algebarski izraz za neodredi integral, ova metoda nije apstraktna, već bezuslovno prijavljuje izuzetak tipa `G_nema_integral`. Ako za funkciju postoji algebarski izraz za neodređeni integral, u izvedenoj klasi potrebno je redefinisati metodu `I()`. Tada će se izrazom $f \cdot I(x)$ pozivati ta metoda za računanje vrednosti $I(x) = \int f(x)dx$. Ako za neku funkciju ne može da se računa vrednost neodređenog integrala nije potrebno redefinisati metodu `I()`, pa izrazom $f \cdot I(x)$ pozivач će metodu iz osnovne klase koja će da prijavi izuzetak tipa `G_nema_integral`.

Prisustvo naredbe `return` u metodi `I()` je potpuno nelogično. Ona sigurno nikada neće da se izvršva zbog bezuslovnog prijavljivanja izuzetka ispred nje. Prisutna je iz tehničkih razloga. Pošto tip vrednosti metode `I()` nije `void`, prema pravilima jezika C++ treba da sadrži naredbu `return` kojom se određuje vrednost funkcije. Prevodioci za jezik C++ prijavljuju grešku ili opomenu ako ta naredba ne postoji, čak i ako je, bar za čoveka jasno, da do njenog izvršavanja nikada neće doći.

Ako za funkciju $f(x)$ postoji algebarski oblik za neodređeni integral $I(x)$, određeni integral na intervalu od a do b računa se prema tačnoj formuli:

$$\int_a^b f(x)dx = I(b) - I(a)$$

Ako ne postoji funkcija $I(x)$ vrednost određenog integrala računa se pomoću neke od približnih postupaka. Najjednostavniji približni postupak je metoda pravougaonika. Interval od a do b se podeli na n jednakih delova širine $\Delta x = (b-a)/n$ i određeni integral se računa prema približnoj formuli:

$$\int_a^b f(x) dx \approx \Delta x \sum_{i=0}^{n-1} f(a + i\Delta x)$$

Program 6.6 prikazuje deklaraciju globalne funkcije `integral` u prostoru imena `Funkcije`, a program 6.7 definiciju te funkcije.

```
// Deklaracija funkcije za izračunavanje određenog integrala (integral).

#ifndef _integ_h_
#define _integ_h_

#include "fct.h"

namespace Funkcije {
    double integral (const Fct& f, double a, double b);
} // namespace Funkcije

#endif
```

Program 6.6 – Deklaracija funkcije za izračunavanje određenog integrala (`integ.h`)

```
// Definicija funkcije integral.

#include "integ.h"

double Funkcije::integral (const Fct& f, double a, double b) {
    try {                                // Pokušaj tačnog računanja.
        return f.I(b) - f.I(a);
    } catch (G_nema_integral) {           // Približno računanje.
        const int N = 1000;
        double dx = (b-a) / N;
        double y = 0;
        for (int i=0; i<N; y+=f(a+dx*i++));
        return y * dx;
    }
}
```

Program 6.7 – Definicija funkcije `integral` (`integ.C`)

Pošto funkcija pripada prostoru imena `Funkcije`, a definicija je izvan tog prostora, trebalo je sa `Funkcije::` naznačiti da se definisiše funkcija iz tog prostora.

Prvi parametar funkcije `integral` je objekat tipa `Fct` koji u sebi sadrži definiciju funkcije čiji integral se traži. Druga dva parametra određuju interval integraljenja.

- U osnovnom delu naredbe `try` pokušava se tačno izračunavanje određenog integrala na osnovu vrednosti neodređenog integrala u graničnim tačkama. Ukoliko za funkciju koju predstavlja objekat `f` ne postoji algebarski izraz za neodređeni integral, izraz `f.I(b)` ili

6.6.2 Izračunavanje određenog integrala

`f.I(a)` će dovesti do prijavljivanja izuzetka tipa `G_nema_integral`. Tada se u rukovajuću tim izuzetkom određeni integral izračunava prema ranije navedenoj približnoj formuli. Tačnosti rezultata nije posvećena posebna pažnja, jer svrha ovog zadatka je razmatranje sasvim drugih problema. Broj intervala N je odabran sasvim proizvoljno. Mnogo je važnije uočiti način kojim je pomoću izuzetaka omogućeno da funkcija `integral()` uvek daje neki rezultat, tačan ili približan, zavisno od funkcije `f`. Poslednja izjava je tačna samo pod uslovom da se u toku računanja vrednosti izraza `f(...)` nijednom ne prijavljuje izuzetak druge vrste (na primer, pokušaj računanja logaritma negativnog broja).

Funkcija `integral` u programu 6.7 napisana je za slučaj apstraktne klase `Fct`. Moći će da se primeni na bilo koje funkcije koje su ostvarene kao izvedene klase iz klase `Fct`. U svakoj od njih mora da se definisiše metoda `operator() (double)`, a metoda `I (double)` ako može. U nastavku su prikazana ostvarenja klase za konkretnе funkcije iz postavke zadatka.

Prva konkretna funkcija je funkcija $\sin x$ čiji neodređeni integral je $-\cos x$. Klasa `Sin` u programu 6.8 ostvaruje tu funkciju.

```
// Definicija klase za funkciju sinus (Sin).

#ifndef _sin2_h_
#define _sin2_h_

#include "fct.h"
#include <cmath>
using namespace std;

namespace Funkcije {
    class Sin: public Fct {
        public:
            double operator() (double x) const { return sin (x); } // Funkcija.
            double I (double x) const { return -cos (x); }          // Integral.
    }; // class Sin
} // namespace Funkcije

#endif
```

Program 6.8 – Definicija klase za funkciju sinus (`sin2.h`)

Druga konkretna funkcija je $e^{-0.1x} \sin x$ koja predstavlja prigušene oscilacije. Nalaženje algebarskog izraza za neodređeni integral ove funkcije prevaziđa doseg ove knjige, pa je uzeto kao da takvog izraza nema. Zbog toga klasa `Oscil` u programu 6.9 ne sadrži metodu `I ()`.

Funkcija $\log_e x$, pored toga što je i za nju uzeto da ne postoji algebarski izraz za njen neodređeni integral, definisana je samo za vrednosti nezavisne promenljive veće od nule. Zbog toga se na početku programa 6.10 prvo nalazi klasa `G_nema_logaritam` za prijavljivanje izuzetaka za slučaj nedozvoljene vrednosti parametra.

Privatno polje `x` omogućava dostavljanje nedozvoljene vrednosti parametra do rukovoca izuzecima.

Postoje dva konstruktora. Podrazumevani konstruktor postavlja nepromenljiv tekst poruke konstruktorom osnovne klase `Greska (const char*)`. Drugi konstruktor ne

// Definicija klase za prigušene oscilacije (Oscil).

```
#ifndef _oscil_h_
#define _oscil_h_

#include "fct.h"
#include <cmath>
using namespace std;

namespace Funkcije {
    class Oscil: public Fct {
        public:
            double operator() (double x) const { return exp(-0.1*x) * sin(x); }
    } // namespace Funkcije
}

#endif
```

Program 6.9 – Definicija klase za prigušene oscilacije (oscil.h)

stavlja poruku (koristi se podrazumevani konstruktor Greska() osnovne klase), ali zato vrednost svog parametra stavlja u polje x. Pošto polje klase i parametar konstruktora imaju isto ime x, samo x označava parametar, a za označavanje polja mora da se piše `this->x`.

Rukovalac može da dohvati vrednost polja pomoću javne metode `uzmiX()`. Ako u objektu postoji poruka, što se proverava pomoću zaštićene metode `ima_poruke()` osnovne klase, objekat je inicijalizovan podrazumevanim konstruktorm, pa polje x nije postavljano. U tom slučaju metoda `uzmiX()` vraća vrednost +1. Tako rukovalac može da sazna da ne može dobiti informaciju o vrednosti nedozvoljenog parametra za računanje logaritma (nedozvoljene vrednosti nisu veće od nule). Ako poruke nema znači da je korišćen drugi konstruktor, koji je postavio vrednost polja x, pa se ona vraća kao vrednost metode `uzmiX()`.

Privatna metoda `pisi()`, takođe, otkriva šta treba da radi pomoću metode `ima_poruke()`. Ako postoji poruka, treba da ispiše nju. Pošto je pokazivač na poruku u osnovnoj klasi privatno polje, jedini način za ispisivanje poruke je pozivanjem zaštićene metode `pisi()` osnovne klase. Pošto je ona sakrivena istoimenom metodom izvdene klase `G_nema_logaritam`, njeno pozivanje treba naznačiti sa `Greska::pisi(it)`. Ako u objektu nema poruke tekst za ispisivanje se sastavlja na licu mesta, ugrađujući u poruku i vrednost polja x.

Klasa Log u programu 6.10 sadrži samo metodu za računanje vrednosti funkcije. Ako vrednost parametra x nije prihvatljiva, prijavlji izuzetak objektom koji stvara konstruktorom `G_nema_logaritam(double)`, dostavljajući na taj način tu nedozvoljenu vrednost rukovaocu izuzecima. Ako je vrednost parametra prihvatljiva vrednost metode `veznost` bibliotečke funkcije `log()`. Pošto još nije uvezeno ništa iz prostora imena std, moralo je da se piše `std::log(x)`.

Njedna od razmatranih funkcija nema parametre, pa klase za njihova ostvarenja nemaju polja. Zbog toga svi objekti, na primer, tipa Sin su međusobno jednaki. Pošto nemaju polja, nisu potrebnii konstruktori ni destruktori.

// Definicija klase za funkciju logaritam (Log).

```
#ifndef _log_h_
#define _log_h_

#include "fct.h"
#include "greska.h"
#include <cmath>
using Usluge::Greska;

namespace Funkcije {
    class G_nema_logaritam: public Greska {
        public:
            G_nema_logaritam (): Greska ("Logaritam negativnog broja") {}
            G_nema_logaritam (double x): Greska () { this->x = x; }
            double uzmiX () const { return ima_poruke() ? 1 : x; } // Uzmi nedozv. vrednost.
            private:
                void pisi (ostream& it) const {
                    if (ima_poruke ()) Greska::pisi(it);
                    else it << "\n*** Ne postoji logaritam od " << x << " ***\n";
                }
    }; // class G_nema_logaritam

    class Log: public Fct {
        public:
            double operator() (double x) const {
                if (x <= 0) throw G_nema_logaritam (x);
                return std::log (x);
            }
    }; // class Log
} // namespace Funkcije
#endif
```

Program 6.10 – Definicija klase za funkciju logaritam (log.h)

Za razliku od njih, polinom kao funkcija određena je svojim redom i nizom koeficijenata. Za uskladištanje tih informacija potrebna su polja i mogućnost postavljanja vrednosti tih polja. Program 6.11 prikazuje definiciju klase za polinome realnih koeficijenata.

Klasa Poli je izvedena iz klase Fct i sadrži polje a tipa Vekt (klasa iz zadatka 6.6.1) za smeštanje niza koeficijenata. Nije se odlučilo za izvođenje klase Poli i iz klase Vekt, bez obzira što bi i takvo rešenje bilo tehnički prihvatljivo, iz semantičkih razloga. Opis „polinom je funkcija i vektor“ nije primeren. Sa vektorima se asocijiraju i radnje (na primer, uređivanje elemenata) što nije primereno polinomima. Opis „polinom je funkcija koja sadrži vektor“ je verodostojniji. Polinom jeste funkcija koja može da se izračunava za određenu vrednost nezavisne promenljive. Vektor za uskladištanje koeficijenata je unutrašnja stvar polinoma. Ako klasa za vektore i podržava uređivanje elemenata, klasa za polinome tu mogućnost neće da koristi, niti treba da omogući spoljnjem svetu da koristi. Kao opšte pravilo za odlučivanje da li se klasa B izvodi iz klase A ili bolje je da klasa B sadrži polje klase A, potrebno je zapitati se da li opis „B je A“ primereno u dатој situaciji. Ako da, treba izvoditi, inače ne.

```
// Definicija klase za polinome (Poli).

#ifndef _poli2_h_
#define _poli2_h_

#include "fct.h"
#include "vekt.h"

namespace Funkcije {
    class Poli: public Fct {
        Vekt a; // Vektor koeficijenata.
    public:
        Poli () : a () {} // Konstruktor.
        explicit Poli (int n) : a (0, n) {} // Dohvatanje koeficijenta.
        double& operator[] (int i) { return a[i]; } // Dohvatanje koeficijenta.
        const double& operator[] (int i) const { return a[i]; }
        int red () const { return a.max_ind (); } // Red polinoma.
        double operator() (double x) const; // Vrednost polinoma.
        double I (double x) const; // Vrednost integrala.
    }; // class Poli
} // namespace Funkcije

#endif
```

Program 6.11 – Definicija klase za polinome (poli2.h)

Na početku javnog dela klase Poli nalaze se dva konstruktora koji uz pomoć odgovarajućih konstruktora klase Vekt inicijalizuju polje a. Prvi konstruktor kao parzan vektor, a drugi kao vektor opsega indeksa od 0 do n, što odgovara polinomu reda n. Elementi vektora se inicijalizuju nulama (videti zadatak 6.6.1).

Pošto se u klasi Poli ne koristi dinamičko dodeljivanje memorije neposredno (to se dešava posredno u klasi Vekt) nije bilo potrebno definisati konstruktor kopije, destruktor i operator =. Te metode će se automatski generisati tako da će, pozivanjem odgovarajućih metoda klase Vekt, obezbediti ispravno kopiranje i uništavanje objekata tipa Poli u celini.

Preklopjeni operator [] omogućuje postavljanje koeficijenata polinoma na određene vrednosti. Preklapanje je uređeno u dve varijante, za promenljive i za nepromenljive polinome. I ovde se oslanja na odgovarajuće varijante preklopjenog operatora [] iz klase Vekt.

Metoda red() dohvata red polinoma. Red polinoma jednak je gornjoj granici opsega indeksa u sadržanom vektoru (a.max_ind()).

Skreće se pažnja, da korisnik klase Poli ne vidi mogućnosti klase Vekt koje nisu primerene polinomima. Na primer, skalarni proizvod polinoma nema smisla. Pošto je a privatno polje klase Poli, korisnik klase Poli ne može da pozove metodu Vekt::operator*().

Jedine dve metode koje su programirane u klasi Poli su metode koje čine polinom funkcijom: računanje vrednosti polinoma (operator()()) i neodređenog integrala (I()) u nekoj tački x. One izračunavaju vrednosti izraza:

$$p(x) = \sum_{i=0}^n a_i x^i, \quad \text{odnosno} \quad I(x) = \sum_{i=0}^n a_i \frac{x^{i+1}}{i+1} = x \sum_{i=0}^n \frac{a_i}{i+1} x^i$$

6.6.2 Izračunavanje određenog integrala

Ostvarenja tih metoda su prikazana u programu 6.12.

```
// Definicije metoda uz klasu Poli.

#include "poli2.h"

double Funkcije::Poli::operator() (double x) const { // Vrednost polinoma.
    double s = 0;
    for (int i=a.max_ind(); i>=0; (s+=x)+=a[i--]);
    return s;
}

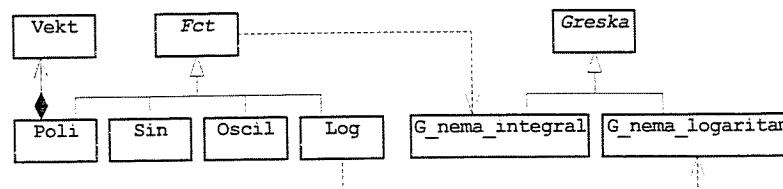
double Funkcije::Poli::I (double x) const { // Vrednost integrala.
    double s = 0;
    for (int i=a.max_ind(); i>=0; i--) (s+=x)+=a[i]/(i+1);
    return s * x;
}
```

Program 6.12 – Definicije metoda uz klasu Poli (poli2.C)

Pošto se definicije nalaze izvan klase kojoj pripadaju metode i izvan prostora imena kome pripada sama klasa, za označavanje metoda prvo treba navesti ime prostora imena, onda ime klase i na kraju ime metode, stavljajući operator za razrešenje dosega :: između pojedinih imena (Funkcije::Poli::...).

Skreće se pažnja na to da za pristup do reda (a.max_ind()) i koeficijenata (a[i]) polinoma se koriste metode klase Vekt.

Slika 6.3 prikazuje dijagram klasa opisanog sistema klasa.



Slika 6.3 – Klase za funkcije

Program 6.13 služi za ispitivanje klasa funkcija za računanje integrala.

U svakom prolazu kroz glavni ciklus, posle čitanja jednoslovne oznake vrste funkcije, napravi se dinamički objekat odgovarajućeg tipa čija se adresa stavlja u pokazivač fct tipa Fct*. Posle se pročitaju granice intervala za integraljenje i izračuna se određeni integral pozivanjem globalne funkcije integral (integral(*fct, a, b)). Na kraju ciklusa objekat funkcije se uništi (delete fct). Ovo se ponavlja sve dok se umesto oznake vrste funkcije ne pročita tačka.

Ako se odabere polinom, pre stvaranja objekta tipa Poli, prvo se pročita red polinoma n i tek onda se stvara objekat (new Poli(n)) koji će sadržati vektor dužine n+1 popunjeno nulama. Adresa novostvorenog objekta se stavlja u pomoćni pokazivač p tipa Poli*. Pomoću tog pokazivača mogu da se postave vrednosti koeficijenata ((*p)[i]=k). Indeksi

6.13 Program za ispitivanje klasa za računanja integrala.

```
#include "fct.h"
#include "sin2.h"
#include "oscil.h"
#include "poli2.h"
#include "log.h"
#include "integ.h"
#include "greska.h"
#include <iostream>
#include <new>
using namespace std;
using namespace Funkcije;

int main () {
    while (true) {
        Fct* fct = 0;
        try {
            cout << "\nFunkcija (S, O, L, P, .)? ";
            char izb; cin >> izb;
            switch (izb) {
                case 's': case 'S': fct = new Sin; break;
                case 'o': case 'O': fct = new Oscil; break;
                case 'l': case 'L': fct = new Log; break;
                case 'p': case 'P': {
                    int n; cout << "Red polinoma? "; cin >> n;
                    Poli* p = new Poli (n);
                    while (1) {
                        cout << "Indeks i vrednost koeficijenta? ";
                        int i; double k; cin >> i >> k;
                        if (i < 0) break;
                        (*p)(i) = k;
                    }
                    fct = p;
                    break;
                }
                case '.': break;
                default: throw "Nedozvoljen izbor";
            }
        }
        if (!fct) break;
        double a, b; cout << "Interval? "; cin >> a >> b;
        cout << "Integral= " << integral(*fct,a,b) << endl;
        catch (const Greska& g) {
            cout << g;
        }
        catch (Vekt::Greska g) {
            cout << "\n*** Greska Vektor::" << g << " ***\n\a";
        }
        catch (const char* g) {
            cout << "\n*** " << g << " ***\n\a";
        }
        catch (bad_alloc) {
            cout << "\n*** Nesupesna dodela memorije ***\n\a";
        }
        delete fct;
    }
}
```

Program 6.13 – Ispitivanje klasa za računanje integrala (integt.C)

6.6.2 Izračunavanje određenog integrala

i vrednosti koeficijenata koji nisu nula čitaju se u lokalnom ciklusu sve dok se ne pročita negativna vrednost za indeks. Na kraju se vrednost pokazivača p stavlja u pokazivač fct. Ova dodata vrednosti je ispravna, jer postoji automatska konverzija tipa pokazivača na izvedenu klasu u pokazivač na osnovnu klasu.

Mogla je adresa novog polinoma, kao i u slučaju drugih funkcija, neposredno da se stavi u pokazivač fct. Međutim, pošto u klasi Fct ne postoji operator [] , za postavljanje vrednosti koeficijenata ne bi moglo da se piše (*fct)[i], već moralo bi da se piše (*dynamic_cast<Polii>(fct))[i]. Rešenje primenjeno u programu 6.13 je, očigledno, mnogo elegantnije.

Pošto u toku prethodnih operacija na raznim mestima mogu da se pojave izuzeci, sve radnje za jedan prolaz kroz glavni ciklus programa, osim uništavanja objekta funkcije, nalaze se u osnovnom bloku naredbe try . Sama obrada je programirana za slučaj kada je sve u redu. Na kraju nijedne operacije se ne pita da li se desio izuzetak u toku prethodne operacije (izvršavanja nadležne funkcije). Jedino u grani default naredbe switch se konstatiuje da je unet nedozvoljen izbor. Problem se prijavljuje izuzetkom tipa char* , tj. običnim tekstom. Ovakav tip izuzetka je prihvatljiv samo kao priručno rešenje za lokalne potrebe. U slučaju klase i funkcija opšte namene ne bi trebalo ovako postupiti. Preporučuje se uvođenje specijalnih tipova za prijavljivanje grešaka: klase (kao što je urađeno u ovom zadatku), ili bar nabranja (kao što je rađeno u klasi Vekt u prethodnom zadatku).

U narebi try programa 6.13 postoji čak četiri rukovalaca izuzecima.

Prvi rukovalac je tipa const Greska& i namenjen je za prihvatanje izuzetaka razvijenih u okviru ovog zadatka. Ispisuje poruku o grešci pozivanjem prijateljske operatorske funkcije za operator << koja je pravljena uz klasu Greska. Parametar rukovaoca je deklarisan kao upućivač da bi se izbeglo kopiranje objekta koji se stvara na mestu prijavljivanja izuzetka pri inicijalizaciji parametra rukovaoca. Ovo je preporučljivo rešenje za izuzetke klasnih tipova, naročito ako se u njima koristi i dinamičko dodeljivanje memorije.

Drugi rukovalac je tipa Vekt::Greska nabranja koje je definisano u klasi Vekt (videti zadatak 6.6.1). Ispisuje poruku koja sadrži celobrojnu šifru prijavljenog izuzetka, uz naznaku da greška potiče iz klase Vekt (na primer: Greska Vekt::1). Nije baš lepo, ali u klasi Vekt nema podrške za nešto bolje.

Treći rukovalac je tipa const char* za prihvatanje priručnog izuzetka iz glavne funkcije.

Poslednji rukovalac je tipa bad_alloc . Prihvata izuzetke koje prijavljuje operator new . Skreće se pažnja na to da je uništavanje objekta funkcije (delete fct) stavljenovo iza svih rukovalaca izuzecima. Time je obezbedeno da, ako se posle stvaranja objekta desi neki izuzetak, taj objekat ipak bude uništen. Desi li se izuzetak pre stvaranja objekta, neće ništa loše da se desi, jer tada na ovom mestu pokazivač fct imaće svoju početnu vrednost 0, a primena operatora delete na pokazivač vrednosti nula je bezopasna.

Jedan od velikih problema kod primene izuzetaka je obezbeđivanje ispravnog oslobođanja dinamički dodeljene memorije.

Rezultat 6.2 prikazuje primer rada programa 6.13.

```

e CC integt integ poli2 vekt
e integt

Funkcija (S, O, L, P, .)? s
Interval? 0 3.14
Integral= 2

Funkcija (S, O, L, P, .)? o
Interval? 0 3.14
Integral= 1.71327

Funkcija (S, O, L, P, .)? l
Interval? 1 10
Integral= 14.0155

Funkcija (S, O, L, P, .)? p
Red polinoma? 3
Indeks i vrednost koeficijenta? 3 -1
Indeks i vrednost koeficijenta? 2 2
Indeks i vrednost koeficijenta? 1 -3
Indeks i vrednost koeficijenta? 0 4
Indeks i vrednost koeficijenta? -1 0
Interval? -5 5
Integral= 206.667

Funkcija (S, O, L, P, .)? k
*** Nedozvoljen izbor ***

Funkcija (S, O, L, P, .)? l
Interval? -1 1
*** Ne postoji logaritam od -1 ***

Funkcija (S, O, L, P, .)? p
Red polinoma? 3
Indeks i vrednost koeficijenta? 6 1
*** Greska Vektor::2 ***

Funkcija (S, C, L, P, .)? p
Red polinoma? 2000000000
*** Nesupesna dodela memorije ***

Funkcija (S, O, L, P, .)?

```

Rezultat 6.2 – Izračunavanje određenog integrala programom 6.13

7 Generičke funkcije i klase

Često je potrebno istu obradu primeniti na podatke različitih tipova. Na primer:

- naći veći od dva podatka,
- ostvariti rad sa stekom,
- urediti niz podataka po nekom kriterijumu,
- ...

Sa stanovišta algoritma za sve navedene obrade potpuno je svejedno da li se radi o celim brojevima, realnim brojevima ili pak o pravougaonima.

Do sada izloženim tehnikama programiranja bilo bi neophodno napisati po jednu funkciju za svaku kombinaciju tipova parametara za datu obradu. Zahvaljujući mogućnosti preklapanja imena funkcija, sve te funkcije mogu da imaju ista imena. Na primer:

```

char max (char i, char j) { return i>j ? i : j; }
int max (int i, int j) { return i>j ? i : j; }
float max (float i, float j) { return i>j ? i : j; }
...

```

Razlike između svih ovih funkcija svode se na sistematsko zamenjivanje oznake tipa unutar cele definicije funkcije. To je rutinski posao koji može i da se automatizuje.

U jeziku C++ postoji mogućnost definisanja **šablonu** za opisivanje date obrade za opšti slučaj, ne navodeći konkretnie tipove podataka. Na osnovu takvih šablonu mogu da se, po potrebi, automatski generišu konkretnie funkcije za konkretnie tipove podataka.

Šabloni mogu da se definišu i za klase u kojima je pitanje tipova nekih polja, ili parametara nekih metoda ostavljeno otvoreno u vreme pisanja šablonu.

Funkcije ili klase opisane pomoću šablonu nazivaju se i **generičke funkcije** ili **generičke klase** na osnovu kojih se kasnije generišu konkretne funkcije ili klase.

7.1 Definisanje šablonu

Opšti oblik za šablove generičkih funkcija ili klasa je:

```
template < parametar , parametar , ... , parametar > opis
```

Opis može da bude deklaracija (prototip) ili definicija generičke funkcije, odnosno definicija ili deklaracija generičke klase čiji se šablon opisuje.

Šabloni mogu da imaju *parametre* koji se redaju iza službene reči **template**, unutar zagrada obrazovanih od znakova manje i veće (<>). *Parametri* mogu da označavaju tipove ili konstante. Slično funkcijama, ovi (formalni) parametri biće zamenjeni (stvarnim) argumentima na mestima korišćenja šablonu. Opšti oblik *parametara* je:

```
class identifikator_tipa
typename identifikator_tipa
oznaka_tipa identifikator_konstante
ili
ili
```

Službene reči **class** ili **typename** označavaju parametre koji su obični (ne generički) tipovi podataka. *Identifikator tipa* može da se unutar generičke funkcije ili klase koristi na svim mestima gde se očekuju identifikatori tipova. Na mestima upotrebe šablonu argumenti mogu da budu proizvoljnih tipova, uključujući i standardne proste tipove. Upočinjenicu da se radi o bilo kakvom tipu (reč **class** sugerire klasne tipove). Reč **class** ima tu privlačnu osobinu da je kraća za 3 slova od reči **typename**. U ovoj knjizi reč **class** se koristi samo za parametre šablonu koji stvarno moraju da budu klase. Za parametre koji mogu da budu i prostih i klasnih tipova koristi se reč **typename**.

Parametri koji počinju *oznakom tipa* predstavljaju simboličke konstante. U obzir dolaze celobrojni tipovi (uključujući i nabranjanja) i pokazivači ili upućivači na objekte ili koristi svuda gde se očekuju konstante. Na mestima upotrebe šablonu, odgovarajući upućivač rezultat mora da pokazuje ili upućuje na objekat ili funkciju sa spoljašnjim povezivanjem, čije se adrese znaju još pre početka izvršavanja programa. To su globalni podaci i funkcije, kao i zajednička polja i metode klasa.

Šablone koristi prevodilac za generisanje konkretnih funkcija ili klasa. Zbog toga treba da se stavljuju u zaglavlja (datoteke .h). Odvojeno prevodenje šablonu nema smisla.

Osnovna mana šablonu generičkih funkcija i klase je što izlazi javnosti detalje razvijanja zatvorenih programskih sistema, koji se korisnicima isporučuju u prevedenom i povezanim obliku. Treba ih oprezno koristiti za razvijanje biblioteka funkcija opšte namene za podatke nepoznatih tipova u vreme pisanja tih funkcija.

Evo nekoliko primera definisanja šablonu:

```
template <class T> T max (T a, T b) { return a>b ? a : b; }
template <class T> void uređi (T af), int n);
template <typename T, int k> // Definicija klase.
class Vekt {
    T niz [k];
public:
    T& operator[] (int i) { return niz[i]; }
    void prazni ();
};

template <typename T, int k> // Definicija metode.
void Vekt<T,k>::prazni () {
    for (int i=0; i<k; niz[i++]=T());
}
```

Prvi primer predstavlja definiciju generičke funkcije za nalaženje većeg od dva podatka nekog, u ovom momentu nepoznatog, tipa T. Pošto konkretni tip može da označava objekte

7.1 Definisanje šablonu

proizvoljne složenosti, verovatno bi bilo povoljnije da su parametri deklarisani kao upućivači (**const T&**). Time bi se izbeglo kopiranje složene strukture argumenata u toku inicijalizacije parametra vrednostima argumenata funkcije.

Drugi primer predstavlja deklaraciju generičke funkcije za uredivanje niza podataka zadate dužine. Pitanje tipa elemenata niza ostavljeno je otvoreno do daljnog.

Sledeći primer je definicija generičke klase. Na osnovu priloženog šablonu mogu da se generišu klase čiji su primerci vektori (jednodimenzionalni nizovi) staticki određenog kapaciteta i određenog tipa elemenata. Kapacitet (**k**) i tip elemenata vektora (**T**) su parametri šablonu. Skreće se pažnja na korišćenje identifikatora **k** za označavanje dužine polja niza. U klasi su predviđene i dve metode: operator za dohvatanje datog elementa niza (indeksiranje) i pražnjenje sadržaja niza. Tip rezultata indeksiranja je upućivač na objekte (nekog) tipa T.

Poslednji primer prikazuje kako se odvojeno definišu metode generičkih klasa. Prilikom odvojenog definisanja metode obične klase trebalo je navesti identifikator klase radi proširenja dosega na tu definiciju. To je potrebno i u slučaju generičkih klasa, ali sada identifikator generičke klase ne određuje jednoznačno konkretnu klasu. Zato je potrebno iz identifikatora klase navesti i identifikatore parametara šablonu između para zagrada <>. U posmatranom slučaju to je **Vekt<T, k>::**. Skreće se još pažnja na to da metoda **prazni()** elementima niza dodeljuje vrednost podrazumevanog konstruktora **T()**. U slučaju prostih tipova to označava nulu odgovarajućeg tipa (videti i odeljak 3.4.7), dok za klasne tipove projektant klase treba da odredi šta to znači „prazan“ objekat.

7.2 Generisanje funkcija i klase

Konkretnе funkcije i klase na osnovu šablonu mogu da se generišu automatski i na zahtev programera.

Konkretna funkcija se automatski generiše na osnovu šablonu generičke funkcije kada se nađe na prvo pozivanje generičke funkcije datim skupom konkretnih tipova i konstanti.

Konkretna klasa (tip) se automatski generiše na osnovu šablonu generičke klase kada se nađe na prvu definiciju objekta datim skupom konkretnih tipova i konstanti. Metode generisane klase se generišu tek na mestima njihovih prvih pozivanja.

Oznaka generisane funkcije, odnosno oznaka generisane klase treba da sadrži i argumente šablonu:

funkcija < argument , argument , ... , argument > Klasa < argument , argument , ... , argument >	odnosno
---	---------

Sam identifikator generičke funkcije ili klase ne određuje jednoznačno konkretnu funkciju, odnosno klasu. Argumenti mogu da budu:

oznaka_tipa konstantan_izraz	ili
---------------------------------	-----

Oznake tipa mogu da stoje na mestima na kojima se u šablonu nalaze identifikatori tipa. Mogu da budu identifikatori standardnih ili nestandardnih tipova. Pored toga mogu da budu i izvedeni tipovi, na primer pokazivači na druge tipove.

Konstantni izrazi mogu da stoje na mestima na kojima se u šablonu nalaze identifikatori konstanti. Vrednosti tih izraza se izračunavaju u vreme prevođenja, zato

ne mogu da sadrže promenljive. Ukoliko izraz sadrži operator `>`, treba ga stavljati unutar oblih zagrada `()`. U suprotnom bi se taj znak shvatio kao kraj niza argumenata.

Prilikom pozivanja izgenerisane funkcije, po potrebi, vrše se uobičajene automatske konverzije tipova argumenata u tipove parametara funkcije.

Evo primera za generisanje funkcija i klasa na osnovu šablona iz odeljka 7.1:

```
int a = max<int> (1, 2);           // Generiše se int max(int,int).
char b = max<char> ('1', '2');    // Generiše se char max(char,char).
int c = max<int> (1, 2.f);        // Generiše se int max(int,int).
float d = max<float> (1, 2.f);    // Generiše se float max(float,float).

Vekt<int,10> vekt1;             // Sadrži niz od 10 elemenata tipa int.
Vekt<double,20> vekt2;           // Sadrži niz od 20 elemenata tipa double.
```

Prilikom generisanja funkcija nekoliko zadnjih argumenata šablona mogu da se izostave ako se tipovi koje treba da predstavljaju mogu jednoznačno da odrede na osnovu tipova argumenata funkcije. Uklapanje tipova mora da je moguće bez primene konverzije tipova na stvarne argumeante funkcije. Na primer:

```
int e = max (1, 2);              // Generiše se int max(int,int).
char f = max ('1', '2');         // Generiše se char max(char,char).
float g = max (1, 2.f);          // GREŠKA: tipovi argumenata se ne
                                // uklapaju.
```

Ne moraju svi parametri šablona generičke funkcije da se koriste za označavanje tipova parametara generičke funkcije. Odgovarajući argumenti šablona nikada ne mogu da se izostave prilikom pozivanja generičke funkcije. Na primer:

```
template <class A, class B> A fun (const B& b) { return b; }

int h = fun<int, float> ('a');   // A=int, B=float
int i = fun<int> ('a');          // A=int, B=char
int j = fun ('a');                // GREŠKA: A=??, B=char
```

Dozvoljeno je da istovremeno postoje obična (negenerička) funkcija s datim prototipom i generička funkcija na osnovu koje može da se generiše funkcija s istim prototipom. U tom slučaju, ako se funkcija poziva bez argumenata šablona, pozvaće se negenerička funkcija. Da bi se generisala funkcija na osnovu šablona potrebno je navesti i argumente šablona.

Na primer:

```
char* max (char* a, char* b); // Deklaracija negeneričke funkcije.
char* max (char* a, char* b) // Definicija negeneričke funkcije.
{ return strcmp(a,b)>0 ? a : b; }

char* k=max ("alfa", "beta"); // Poziva se negenerička verzija
                                // (upoređuje tekstove).
char* l=max<char*> ("alfa", "beta"); // Poziva se generička verzija
                                // (upoređuje pokazivače!).
```

Postoje, doduše retke, situacije kada je za neke konkretnе argumente šablona potrebno izgenerisati konkretnu funkciju ili klasu, ali bez pozivanja funkcije, odnosno bez definisanja objekata. To se postiže navodenjem prototipa generičke funkcije, odnosno deklaracije generičke klase sa željenim argumentima šablona i modifikatorom `template` na početku. Na primer:

```
template long max<long> (long a, long b); // Generisanje funkcije.
template class Vekt<short,55>;            // Generisanje klase.
```

Pošto pri svakom definisanju ili deklarisanju podataka tipa generičke klase moraju da se navedu i argumenti šablona, ako se to radi na mnogo mesta u programu, naredbom `typedef` opštег oblika:

```
typedef Klasa < argument , argument , ... , argument > Ident ;
typedef Vekt<float,100> V;           // Pridruživanje imena.
V vekt3;
```

7.3 Podrazumevane vrednosti parametara šablona Δ

Kao i kod običnih funkcija, i kod šablona generičkih klasa mogu da se predvide podrazumevane vrednosti za nekoliko poslednjih parametara. U tom slučaju na mestu korišćenja ne moraju da se navode argumenti za nekoliko zadnjih parametara s podrazumevanim vrednostima.

Šabloni generičkih funkcija ne mogu da imaju parametre s podrazumevanim vrednostima.

Podrazumevane vrednosti je dovoljno navesti samo u deklaraciji ili definiciji generičke klase. Prilikom odvojenog definisanja metoda generičke klase u zaglavlju šablona ne moraju da se navedu podrazumevane vrednosti. Koristiće se one iz definicije klase.

Ako je parametar tip podrazumevana vrednost može da bude naziv prostog tipa, ne-generičke klase ili generičke klase s argumentima šablona unutar para znakova `<>`. Ako je parametar konstanta podrazumevana vrednost može da bude konstantni celobrojan izraz. Ako izraz sadrži operator `>` mora da se stavi unutar para oblih zagrada, da se znak `>` ne bi tumačio kao kraj zaglavlja šablona.

Evo primera za definisanje generičke klase s podrazumevanim vrednostima parametara šablona:

```
template <typename T=int, int k=10> // Definicija klase.
class Niz {
    T niz [k];
public:
    T& operator[] (int i) { return niz[i]; }
    void prazni ();
}

template <typename T, int k> // Definicija metode.
void Niz<T,k>::prazni () {
    for (int i=0; i<k; niz[i++]=T());
}

Niz<char,55> niz1; // Sadrži niz od 55 elemenata tipa char.
Niz<float> niz2;   // Sadrži niz od 10 elemenata tipa float.
Niz<> niz3;        // Sadrži niz od 10 elemenata tipa int.
```

Raniji parametri šablona mogu da se koriste za određivanje vrednosti kasnijih parametara. Na primer:

```
template <typename T, int k, typename V = Niz<short>,
          typename V = Vekt<T,k> >
class Alfa { ... };
```

```

    Alfa<double,10> alfa1;
    Alfa<double,22,Niz<short,10>,Vektor<double,22>> alfa2; // isto što i ...

```

Posebno se skreće pažnja na dva uzastopna znaka > na kraju zaglavlja šablona. Oni moraju da se pišu razdvojeni nekom belinom da se ne bi tumačili kao operator >>.

7.4 Specijalizacija Δ

Dešava se da je za neke specifične vrednosti parametara šablona potrebna specifična definicija generičke funkcije ili klase. Pod specijalizacijom generičke funkcije ili klase podrazumeva se definisanje zasebnog šablona za neke posebne kombinacije parametara prvobitnog (opštег) šablona.

Definicije opštег šablona i svih njegovih specijalizacija moraju da se nalaze u istom prostoru imena.

7.4.1 Specijalizacija generičkih klasa Δ

Opšti oblik specijalizacije generičke klase je:

```

template <parametri> class GKlasa <argumenti>;           ili
template <parametri> class GKlasa <argumenti> { ... };

```

Prvi red predstavlja deklaraciju, a drugi red definiciju specijalizovane generičke klase. *GKlasa* je identifikator generičke klase koja se specijalizuje.

Parametri predstavljaju parametre specijalizovanog šablona, dok *argumenti* određuju varijantu opšte generičke klase koja se zamjenjuje posmatranom specijalizovanom generičkom klasom.

Vrednosti *argumentata* opšte generičke klase mogu da budu konkretni tipovi ili konstante, a mogu da budu izvedene iz *parametara* specijalizovanog šablona. Broj *parametara* može biti manji od broja *argumentata*. *Parametri* specijalizovanog šablona ne smiju imati podrazumevane vrednosti.

Specijalizacija može da bude delimična ili potpuna. Specijalizacija je **delimična** ako specijalizovani šablon ima bar jedan parametar. Na osnovu takvog šablona može da se generiše više različitih specifičnih klasa. Dakle, generičnost, doduše u suženom obimu, još uvek je prisutna. Specijalizacija je **potpuna** ako specijalizovani šablon nema nijedan parametar. Na osnovu takvog šablona može da se generiše samo jedna klasa.

Specijalizacija je moguća samo ako je prethodno navedena bar deklaracija opšte generičke klase. Specijalizacija određenim argumentima specijalizovanog šablona je dozvoljena samo ako još nije generisana nijedna klasa opšte generičke klase za odgovarajući skup argumentata opštег šablona.

Evo primera deklaracije specijalizovanih generičkih klasa na osnovu ranije definisane generičke klase *Vekt*:

```

template <typename T, int k> class Vekt<T*,k>; // Delimične
template <typename T> class Vekt<T,10>;          // specijalizacije.
template <int k> class Vekt<int,k>;
template <> class Vekt<void*,10>;                // Potpuna specijalizacija.

```

Prvi primer je specijalizacija za slučaj kada su elementi vektora pokazivači proizvoljnih tipova, drugi je za vektore od po deset elemenata, treći je za celobrojne vektore različitih dužina.

7.4.1 Specijalizacija generičkih klasa Δ

Poslednji primer je potpuna specijalizacija. Specijalizovani šablon nema argumente, a fiksirani su i tip elemenata vektora (*void**) i kapacitet (10).

Prilikom definisanja objekata (kada se i sama klasa generiše) treba navesti onoliko argumenta koliko ih ima opšti šablon. Prevodilac će da odabere najviše specijalizovani šablon (s najmanje argumentata) u koji može da uklopi te argumente, odnosno opšti šablon ako nijedan specijalizovani šablon ne odgovara. Greška je ako postoji više podjednako specijalizovanih šablona koji bi mogli da se koriste.

Na primer:

```

Vekt<int*,15> vekt4; // Koristi se prvi specijalizovani šablon.
Vekt<float*,10> vekt5; // Koristi se drugi specijalizovani šablon.
Vekt<int,20> vekt6; // Koristi se treći specijalizovani šablon.
Vekt<void*,10> vekt7; // Koristi se četvrti specijalizovani šablon.

```

Naravno, objekti mogu da se definišu tek posle navođenja definicije odgovarajuće generičke klase. Sama deklaracija nije dovoljna.

Vrednosti argumentata (tipovi i konstante) specijalizovanog šablona se izvode iz navedenih argumentata opštег šablona.

Na primer, za slučaj *Vekt<int*,15>* na osnovu prve specijalizacije sa zaključuje da *T** je *int**, dakle prvi argument *T* specijalizovanog šablona je *int*. Ovo treba imati na umu kada se definišu specijalizovane generičke klase.

Kao primer, evo definicije za specijalizaciju generičke klase *Vekt* za slučaj nizova pokazivača proizvoljnih tipova (prva specijalizacija od prethodna četiri primera):

```

template <typename T, int k>
class Vekt<T*,k> {
    T* niz [k];
public:
    T*& operator[](int i) { return niz[i]; }
    void prazni () ;
};

template <typename T, int k>
void Vekt<T*,k>::prazni () {
    for (int i=0; i<k; delete niz[i++]);
}
// Definicija klase.
// Definicija metode.

```

Skreće se pažnja na to da je tip elemenata niza označen sa *T**, jer prilikom definisanja objekata (i generisanja klase) pokazivački argument (na primer *int**) će se uvrštavati umesto navedenog parametra, a to je *T**.

7.4.2 Specijalizacija generičkih funkcija Δ

Za generičke funkcije moguća je samo potpuna specijalizacija. Opšti oblik specijalizacije generičke funkcije je:

```

template <> tip GFunkcija <argumenti> ( ... );
template <> tip GFunkcija <argumenti> ( ... ) { ... }

```

GFunkcija je identifikator generičke funkcije koja se specijalizuje, a *tip* je tip vrednosti funkcije. *Argumenti* određuju varijantu opšte generičke funkcije koja se zamjenjuje posmatranom specijalizovanom funkcijom. Ako na osnovu tipova argumentata funkcije (koji sada moraju biti konkretni tipovi) mogu da se odrede vrednosti (tipovi) argumentata šablona, *argumenti* (uključujući i <>) mogu da se izostave.

Evo primera specijalizacije ranije definisane generičke funkcije max():

```
template<> char* max<char*> (char* a, char* b); // Deklaracija.
template<> char* max (char* a, char* b);
template<> char* max<char*> (char* a, char* b) // Definicija.
    { return strcmp(a,b)>0 ? a : b; }
char* m=max<char*> ("alfa", "beta"); // Opšta char* max(char*,char*)
char* n=max ("alfa", "beta"); // bi uporedjivala pokazivače!
```

Prva dva primera predstavljaju deklaraciju specijalizovanog šablona sa, odnosno bez navođenja argumenata opštег šablona. Treći primer je definicija specijalizovanog šablona. I u ovom primeru je moglo <char*> da se izostavi.

Poslednje dve naredbe prikazuju pozivanje specijalizovane funkcije sa, odnosno bez argumenta šablona. Skreće se pažnja na to da, u slučaju da postoji i negenerička varijanta funkcije max(), u poslednjoj naredbi pozivaće se ta funkcija.

7.5 Generičke metode i unutrašnje generičke klase △

Metode klase mogu da budu generičke funkcije. Same klase mogu da budu obične (negeneričke) ili generičke klase. Kao i obične metode, i generičke metode mogu da se definišu unutar ili izvan klase kojoj pripadaju.

Pošto se generičke metode generišu tek ako se za njih ukaže potreba generičke metode ne mogu da budu virtualne. Broj virtualnih metoda mora unapred da se zna i ne može da zavisi od broja različitih konkretnih tipova argumenata šablona s kojima se pozivaju u toku izvršavanja programa.

Pošto svaka klasa može da ima samo jedan destruktorni, koji nema ni vrednost funkcije, destruktorni ne može da bude generička metoda.

Za konstruktoare nema smetnje da budu generički. Mogu da postoje više konstruktora koji se pozivaju (automatski ili na zahtev) s argumentima različitih tipova. Ponekad ima smisla te konstruktoare generisati na osnovu šablona.

Znak < u izrazima iza operatora ::.. i -> smatra se operatom. Ako znak < treba da predstavlja početak stvarnih arumenata šablona pozivane generičke funkcije, ispred imena metode treba navesti reč template:

```
Klasa :: template metoda < ... > ( ... )
objekat . template metoda < ... > ( ... )
pokazivač -> template metoda < ... > ( ... )
```

Postoje prevodioci koji ne zahtevaju navođenje reči template u ovakvim slučajevima.

Evo primera klase s dve generičke metode:

```
class A {
    double x;
public:
    template <typename T> // Definicija generičkog konstruktora.
        A (const T& t): x (t) {}
    template <typename T> // Deklaracija generičke metode.
        void m (const T& t);
};

template <typename T> // Definicija generičke metode izvan klase.
void A::m (const T& t) { x = t; }
```

```
void f1 () {
    A a1 (5);
    A a2 (1.2);
    A a3 ('q');
    al.template m<int> (1.2);
    al.template m<char> (3.4);
    al.m (3.4);
}
```

```
// Generiše se A::A(int&).
// Generiše se A::A(double&).
// Generiše se A::A(char&).
// Generiše se A::m(int&).
// Generiše se A::m(char&).
// Generiše se A::m(double&).
```

Ako se generička metoda generičke klase definiše izvan svoje klase, oznaka klase kojoj pripada metoda mora da sadrži i parametre šablona generičke klase. Na primer:

```
template <typename T> // Definicija generičke
class B { // klase.
    T t;
public:
    template <typename U> // Deklaracija generičke
        void m(const U&); // metode.
};

template <typename T> // Definicija generičke
template <typename U> // metode izvan
    void B<T>::m (const U& u) // generičke klase.
    { t = u; }

void f2 () {
    B<int> b1;
    B<float> b2;
    b1.m (55);
    b2.m (55);
    b2.template m<char> (55);
}
```

```
// Generiše se A<int>::m(int&).
// Generiše se A<float>::m(int&).
// Generiše se A<float>::m(char&).
```

Unutrašnje klase mogu da budu generičke klase. Sama spoljašnja klasa može da bude obična (negenerička) ili generička klasa. Unutrašnje generičke klase mogu da se koriste na isti način kao i obične (negeneričke) unutrašnje klase. To znači, da izvan spoljašnje klase mora operatorom za razrešenje dosega da se naznači pripadnost spoljašnjoj klasi. U slučaju generičke spoljašnje klase, naravno, oznaka klase mora da sadrži i parametre šablona.

```
template <typename T> // Definicija generičke
class C { // spoljašnje klase.
public:
    template <typename U> // Deklaracija generičke
        class D; // unutrašnje klase.
};

template <typename T> // Definicija generičke unutrašnje
template <typename U> // klase izvan spoljašnje klase.
    class C<T>::D {
public:
    void m (const U&); // Deklaracija metode unutrašnje
    }; // klase.
template <typename T> // Definicija metode unutrašnje
    template <typename U> // klase izvan obe klase.
        void C<T>::D<U>::m (const U& u) {}

void f3 () {
    C<int> c;
    C<double>::D<char> d;
    d.m (true); // Generiše se C<double>::D<char>::m(char&).
}
```

7.6 Zadaci

7.6.1 Fuzija nizova objekata

Zadatak:

Napisati na jeziku C++ generičku funkciju za nalaženje fuzije dva uređena niza objekata u treći, na isti način uređeni niz. Napisati na jeziku C++ program koji korišćenjem prethodne funkcije nalazi fuziju nizova:

- celih brojeva uređenih po vrednostima,
- tačaka u ravni uređenih po odstojanjima od koordinatnog početka,
- pravougaonika uređenih po površinama.

Rešenje:

Program 7.1 prikazuje definiciju generičke funkcije `fuzija()` koja od dva niza uređena po neopadajućim vrednostima elemenata obrazuje treći, na isti način uređeni niz.

```
// Generička funkcija za fuziju nizova objekata.

template <typename T>
void fuzija (const T* a, int na, const T* b, int nb, T*& c, int& nc) {
    c = new T [nc=na+nb];
    int ia=0, ib=0, ic=0;
    while (ia<na || ib<nb)
        c[ic++] = ia==na ? b[ib++] :
                    ib==nb ? a[ia++] :
                    a[ia]<b[ib] ? a[ia++] : b[ib++];
}
```

Program 7.1 – Generička funkcija za fuziju nizova objekata (`fuzija.h`)

Parametar šablona je tip elemenata obrađivanih nizova. Potpuno je nevažno kog su tipa elementi. Jedino je važna pretpostavka da za tip (možda klasu) T relacijski operator < ima neko tumačenje. To jest mora da bude definisano kada je neki objekat „ispred“ (ili „manji“ od) drugog objekta.

Za obradu celobrojnih nizova funkcijom iz programa 7.1 ništa više nije potrebno. Tip `int` je standardni tip i operator < ima vrlo precizno značenje. Apstraktni pojmovi tačka i pravougaonik nemaju unapred definisana značenja u jeziku C++, pa ih treba definisati potpuno odgovarajućih klasa.

Program 7.2 prikazuje jednu jednostavnu klasu tačaka u ravni s minimalnim brojem elemenata potrebnih za rešenje ovog zadatka.

Poseban problem je šta znači uređivanje tačaka? U ovom zadatku je uzeto da se tačke uređuju na osnovu njihovih odstojanja od koordinatnog početka. Jedna tačka je ispred druge ako je bliža koordinatnom početku. Operatorska funkcija `operator<()` ostvaruje ovo tumačenje. Tačnije, upoređuje kvadrate odstojanja, ali to se svodi na isto. U nekoj kompltnijoj klasi trebalo bi definisati operatorske funkcije i za preostale relacijske operatore.

Program 7.3 prikazuje sličnu jednostavnu klasu pravougaonika. Za kriterijum uređivanja uzete su veličine njihovih površina.

7.6.1 Fuzija nizova objekata

// Definicija klase tačaka (Tacka).

```
#include <iostream>
using namespace std;

class Tacka {
    double x, y;
public:
    friend istream& operator>> (istream& ut, Tacka& tt) // Koordinate.
        { return ut >> tt.x >> tt.y; }
    friend ostream& operator<< (ostream& it, const Tacka& tt) // Čitanje.
        { return it << "T(" << tt.x << ',' << tt.y << ')'; }
    friend int operator<(const Tacka& t1, const Tacka& t2) // Uporedovanje.
        { return t1.x*t1.x + t1.y*t1.y < t2.x*t2.x + t2.y*t2.y; }
};
```

Program 7.2 – Definicija klase tačaka (`tacka3.h`)

// Definicija klase pravougaonika (Pravoug).

```
#include <iostream>
using namespace std;

class Pravoug {
    double a, b;
public:
    friend istream& operator>> (istream& ut, Pravoug& pp) // Dužine ivica.
        { return ut >> pp.a >> pp.b; }
    friend ostream& operator<< (ostream& it, const Pravoug& pp) // Čitanje.
        { return it << "P[" << pp.a << ',' << pp.b << ']'; }
    friend int operator<(const Pravoug& p1, const Pravoug& p2) // Uporedi-
        { return p1.a*p1.b < p2.a*p2.b; } // vanje.
};
```

Program 7.3 – Definicija klase pravougaonika (`pravoug1.h`)

Na kraju, program 7.4 prikazuje primer korišćenja generičke funkcije `fuzija()`.

Sadržaj glavne funkcije podeljen je u tri skoro istovetna bloka. Razlika je samo u oznakama tipa elemenata nizova.

Unutar svakog bloka prvo se čita dužina prvog niza, dodeli se prostor za niz u dinamičkoj zoni memorije i niz se napuni vrednostima čitajući s glavnog ulaza računara. Zatim se isto uradi i za drugi niz. Posle definisanja pokazivača na rezultujući niz poziva se funkcija `fuzija()` za obrazovanje fuzije pročitanih nizova. Tu je mesto na kojem prevodilac za vreme prevođenja programa 7.4 generiše konkretnu funkciju za korišćeni tip elemenata nizova. Na kraju bloka, posle prikazivanja rezultujućeg niza, sva tri niza uniše se primenom operatora `delete`.

```
// Ispitivanje generičke funkcije za fuziju nizova.

#include <iostream>
#include "fuzija.h"
#include "tacka3.h"
#include "pravoug1.h"
using namespace std;

int main () {
    int na, nb, nc, i;

    cout << "\nFuzija nizova celih brojeva:\n\n";
    cin >> na; int* a = new int [na]; cout << "Prvi niz:";
    for (i=0; i<na; i++) {cin >> a[i]; cout << ' ' << a[i];} cout << endl;
    cin >> nb; int* b = new int [nb]; cout << "Drugi niz:";
    for (i=0; i<nb; i++) {cin >> b[i]; cout << ' ' << b[i];} cout << endl;
    int* c; fuzija<int> (a, na, b, nb, c, nc);
    cout << "Fuzija :";
    for (i=0; i<nc; i++) {cout << ' ' << c[i];} cout << endl;
    delete [] a; delete [] b; delete [] c;
}

cout << "\nFuzija nizova tacaka:\n\n";
cin >> na; Tacka* a = new Tacka [na]; cout << "Prvi niz:";
for (i=0; i<na; i++) {cin >> a[i]; cout << ' ' << a[i];} cout << endl;
cin >> nb; Tacka* b = new Tacka [nb]; cout << "Drugi niz:";
for (i=0; i<nb; i++) {cin >> b[i]; cout << ' ' << b[i];} cout << endl;
Tacka* c; fuzija<Tacka> (a, na, b, nb, c, nc);
cout << "Fuzija :";
for (i=0; i<nc; i++) {cout << ' ' << c[i];} cout << endl;
delete [] a; delete [] b; delete [] c;
}

cout << "\nFuzija nizova pravougaonika:\n\n";
cin >> na; Pravoug* a = new Pravoug [na]; cout << "Prvi niz:";
for (i=0; i<na; i++) {cin >> a[i]; cout << ' ' << a[i];} cout << endl;
cin >> nb; Pravoug* b = new Pravoug [nb]; cout << "Drugi niz:";
for (i=0; i<nb; i++) {cin >> b[i]; cout << ' ' << b[i];} cout << endl;
Pravoug* c; fuzija<Pravoug> (a, na, b, nb, c, nc);
cout << "Fuzija :";
for (i=0; i<nc; i++) {cout << ' ' << c[i];} cout << endl;
delete [] a; delete [] b; delete [] c;
}
```

Program 7.4 – Primer fuzije nizova objekata (fuzijat.C)

Rezultat 7.1 prikazuje primer rada programa 7.4.

% CC fuzijat -o fuzijat
% fuzijat <fuzijat.pod

Fuzija nizova celih brojeva:

Prvi niz: -3 -1 0 2 2 5 6 6
Drugi niz: -2 2 6 7 8
Fuzija : -3 -2 -1 0 2 2 2 5 6 6 6 7 8

Fuzija nizova tacaka:

Prvi niz: T(0,0) T(-2,-2) T(3,2)
Drugi niz: T(1,1) T(-1,2) T(2,-3) T(3,3)
Fuzija : T(0,0) T(1,1) T(-1,2) T(-2,-2) T(2,-3) T(3,2) T(3,3)

Fuzija nizova pravougaonika:

Prvi niz: P[1,4] P[2,3] P[4,2] P[3,3]
Drugi niz: P[3,1] P[2,3] P[4,3]
Fuzija : P[3,1] P[1,4] P[2,3] P[2,3] P[4,2] P[3,3] P[4,3]

Rezultat 7.1 – Fuzija nizova podataka programom 7.4

7.6.2 Obrada stekova objekata

Zadatak:

Napisati na jeziku C++ generičku klasu za statičke stekove zadatih kapaciteta. Napisati na jeziku C++ program za prikazivanje mogućnosti te klase.

Rešenje:

Stekovi su strukture podataka kod kojih se podatak koji se zadnji stavlja na vrh steka uzima prvi iz steka. Kod statičkih stekova ne koristi se memorija u dinamičkoj zoni i kapacitet steka se određuje u vreme prevodenja programa.

Osnovne radnje čine stavljanje jednog podatka na vrh steka i uzimanje jednog podatka s vrha steka. Greška je pokušati stavljati podatke u pun stek ili uzimati podatke iz praznog steka. Kao pomoćne radnje mogu da se predvide ispitivanja da li je stek pun i da li je prazan.

Sve nabrojane radnje su očigledno nezavisne od tipa objekata koji se smeštaju na stek. Tako se prirodno nameće korišćenje generičke klase za obradu gore opisanih stekova.

Program 7.5 predstavlja definiciju generičke klase za obradu gore opisanih stekova.

Polja klase Stek<> su statički dimenzionisan niz stek od kap elemenata tipa Pod. Parametri šablona su upravo dužina niza i tip elemenata.

Pošto se ne koristi memorija u dinamičkoj zoni inicijalizacija podrazumevanim konstruktorm je krajnje jednostavna. Samo treba naznačiti da je stek prazan. Pražnjenje steka (metoda brisi ()) treba da uradi isto to. Sadržaj samog niza stek uopšte ne mora da se menja.

Greška može da nastane samo u metodama stavi() i uzmi(). Prilikom smeštanja podatka na stek, greška je ako je vrh==kap na početku metode. Prilikom uzimanja podatka sa steka, greška je ako je vrh==0 na početku metode. U oba slučaja greška se prijavljuje izuzetkom tipa char'.

Definicija generičke klase za obradu stekova (*Stek<>*).

```
template <class Pod, int kap>
class Stek {
    Pod stek[kap];
    int vrh;
public:
    Stek () { vrh = 0; } // Inicijalizacija.
    void stavi (const Pod& pod); // Smeštanje podatka.
    void uzmi (Pod& pod);
    int vel () const { return vrh; } // Broj podataka na steku.
    void brisi () { vrh = 0; } // Pražnjenje steka.
    bool prazan () const { return vrh == 0; } // Indikator praznog steka.
    bool pun () const { return vrh == kap; } // Indikator punog steka.
};

// Definicije metoda uz klase Stek<>.

template <class Pod, int kap>
void Stek<Pod,kap>::stavi (const Pod& pod) { // Smeštanje podatka.
    if (vrh == kap) throw "Stek je pun!";
    stek[vrh++] = pod;
}

template <class Pod, int kap>
void Stek<Pod,kap>::uzmi (Pod& pod) { // Uzimanje podatka.
    if (vrh == 0) throw "Stek je prazan!";
    pod = stek[--vrh];
}
```

Program 7.5 – Definisanje generičke klase za stekove (*stek2.h*)

Preostale metode daju razne informacije o stanju steka. Vrednost metode *vel()* je broj podataka na steku. Vrednosti logičkih metoda *prazan()* i *pun()* označavaju da li je stek prazan ili pun.

Metode *stavi()* i *uzmi()* definišu se odvojeno od same klase, ali pošto se radi o metodama generičke klase te definicije moraju da budu u zaglavlju. U ovom slučaju u datoteci *stek2.h*.

Program 7.6 prikazuje primer korišćenja generičke klase *Stek<>*.

U početnom delu glavne funkcije, na osnovu šablonu iz programa 7.5, generišu se dve klase za potrebe dva objekta *clb_stek* i *dbl_stek*. Prva je klasa *Stek<int,CVEL>* (gde je *CVEL* jednak 10) za stekove kapaciteta 10 celobrojnih (*int*) podataka. Druga je klasa *Stek<double,DVEL>* (gde je *DVEL* jednak 3) za stekove kapaciteta tri realna podatka dvostrukе tačnosti (*double*).

U nastavku programa prvo se čitaju celi brojevi s glavnog ulaza računara, sve dok se celobrojan stek (primerak *clb_stek*) ne napuni. Posle se uzimaju podaci s tog steka i ispisuju se na glavnom izlazu računara, sve dok se stek ne isprazni. U poslednjem delu programa isto se uradi i s realnim stekom (primerak *dbl_stek*).

Rezultat 7.2 prikazuje primer rada programa 7.6.

// Program za ispitivanje generičke klase za stekove (*Stek<>*).

```
#include "stek2.h"
#include <iostream>
using namespace std;

const int CVEL=10, DVEL=3;

void main () {
    int i; double d;
    Stek<int,CVEL> clb_stek;
    Stek<double,DVEL> dbl_stek;

    cout << "\nPunjene celobrojne stekove:";
    while (! clb_stek.prazan())
        { cin >> i; clb_stek.stavi (i); cout << ' ' << i; }
    cout << "\nPražnjenje celobrojnog steka:";
    while (! clb_stek.prazan())
        { clb_stek.uzmi (i); cout << ' ' << i; }

    cout << "\nPunjene realne stekove:";
    while (! dbl_stek.prazan())
        { cin >> d; dbl_stek.stavi (d); cout << ' ' << d; }
    cout << "\nPražnjenje realnog steka:";
    while (! dbl_stek.prazan())
        { dbl_stek.uzmi (d); cout << ' ' << d; }
    cout << endl;
}
```

Program 7.6 – Ispitivanje generičke klase za stekove (*stek2t.C*)

* stek2t <stek2t.pod

```
Punjene celobrojne stekove: 0 1 2 3 4 5 6 7 8 9
Pražnjenje celobrojnog steka: 9 8 7 6 5 4 3 2 1 0
Punjene realne stekove: 1.11 2.22 3.33
Pražnjenje realnog steka: 3.33 2.22 1.11
```

Rezultat 7.2 – Obrada stekova programom 7.6

Treba jasno razlikovati stekove koji su primerci klase generisanih na osnovu generičke klase iz programa 7.5 od stekova koji bi bili primerci klase za stekove objekata različitih tipova sa zajedničkom osnovnom klasom.

U prvom slučaju svaki stek tipa *Stek<>* može da sadrži samo podatke međusobno istih tipova. Druga je stvar da na jednom steku mogu da budu celi brojevi, na drugom realni brojevi, a na trećem podaci nekog trećeg (možda klasnog) tipa. Svi ti stekovi su primerci različitih klasa koje su generisane iz iste generičke klase.

U drugom slučaju isti stek može, u isto vreme, da sadrži objekte različitih, ali srodnih, klasnih tipova.

7.6.3 Uređivanje nizova podataka △

Zadatak:

Napisati na jeziku C++ generičke klase za upoređivanje parova podataka po rastućem i po opadajućem poretku. Napisati na jeziku C++ generičku klasu za uređivanje nizova podataka po odabranom redosledu (rastućem ili opadajućem). Napisati na jeziku C++ program koji korišćenjem prethodnih klasa uređuje niz:

- a) celih brojeva po vrednostima,
- b) tačaka u ravni po odstojanjima od koordinatnog početka, i
- c) pravougaonika po površinama

po rastućem i po opadajućem poretku.

Rešenje:

U ovom zadatku, pored ostalog, pokazano je kako mogu klase da se iskoriste za definisanje strategije rada.

Program 7.7 prikazuje jednostavnu generičku klasu Raste<> za upoređivanje dva podatka po rastućem poretku. Sadrži samo jednu zajedničku metodu ispred() za ispitivanje da li se prvi podatak (parametar) nalazi ispred drugog podatka. Po rastućem poretku, naravno, manji podatak se nalazi ispred većeg. Upoređivanje se vrši operatom <, koji je za sve proste tipove definisan u samom jeziku C++. Ako parametar T šablona bude klasa, u toj klasi mora operator < da bude preklopjen na odgovarajući način.

```
// Generička klasa za rastuće upoređivanje podataka.

template <typename T>
class Raste {
public:
    static bool ispred (const T& a, const T& b) { return a < b; }
};
```

Program 7.7 – Generička klasa za rastuće upoređivanje podataka (raste.h)

Slično programu 7.7, program 7.8 prikazuje jednostavnu generičku klasu Opada<> za upoređivanje dva podatka po opadajućem poretku. Po opadajućem poretku veći podatak se nalazi ispred manjeg. Izraz u metodi ispred() i u ovoj klasi je napisan korišćenjem operatora <, da u bi eventualnoj klasi koja bude korišćena kao argument šablona bilo dovoljno preklopiti samo operator <.

```
// Generička klasa za opadajuće upoređivanje podataka.

template <typename T>
class Opada {
public:
    static bool ispred (const T& a, const T& b) { return b < a; }
};
```

Program 7.8 – Generička klasa za opadajuće upoređivanje podataka (opada.h)

7.6.3 Uređivanje nizova podataka △

Program 7.9 sadrži definiciju generičke klase Uredi<> za uređivanje niza podataka tipa T, koji je parametar šablona. Drugi parametar U predstavlja klasu koja određuje strategiju upoređivanja podataka. Kao podrazumevana vrednost navedena je generička klasa Raste<> sa stvarnim parametrom T koji je prvi parametar klase Uredi<>. Pošto je zajednička metoda uređi () relativno složena (sadrži čak dva uklopljena ciklusa), njena definicija je navedena izvan definicije klase kojoj pripada.

```
// Generička klasa za uređivanje niza podataka.

#include "raste.h"

template <typename T, class U=Raste<T> > // Definicija klase.
class Uredi {
public:
    static void uređi (T* a, int n);
};

template <typename T, class U > // Definicija metode.
void Uredi<T,U>::uređi (T* a, int n) {
    for (int i=0; i<n-1; i++)
        for (int j=i+1; j<n; j++)
            if (U::ispred(a[j],a[i]))
                { T b = a[i]; a[i] = a[j]; a[j] = b; }
}
```

Program 7.9 – Generička klasa za uređivanje niza podataka (uredi.h)

Na kraju, program 7.10 prikazuje traženi program za ispitivanje prethodnih generičkih klasa. Za definisanje klasa za tačke i pravougaonike iskorišćeni su programi 7.2 i 7.3 iz zadatka 7.6.1. Slično programu 7.4 u zadatu 7.6.1, postoje tri logički istovetna bloka za rad s pojedinim tipovima podataka.

U prvom bloku se obrađuje niz celih brojeva. Uređivanje niza po rastućem poretku postiže se izrazom Uredi<int>::uređi(a,n). Na tom mestu se generiše zajednička metoda uređi () s prvim argumentom šablona int i podrazumevanim drugim argumentom Raste<int>. Da bi se postiglo uređivanje niza po opadajućem poretku mora da se navede i drugi argument šablona Uredi<> da bi se koristila klasa Opada<>. Tako se dobija izraz Uredi<int, Opada<int> >::uređi(a,n).

Drugi i treći blok se razlikuju od prvog samo po tome što je tip int na svakom mestu gde predstavlja tip elemenata niza, zamenjen tipom Tacka, odnosno Pravoug.

Rezultat 7.3 prikazuje primer rada programa 7.10.

// Ispitivanje generičke klase za uređivanje nizova podataka.

```
#include "uredi.h"
#include "opada.h"
#include "tacka3.h"
#include "pravougl.h"
#include <iostream>
using namespace std;

int main () {

    cout << "\nUredjivanje niza celih brojeva:\n\n";
    int n; cin >> n; int* a = new int [n];
    cout << "Pocetni niz:";
    for (int i=0;i<n;i++) { cin>>a[i]; cout<< ' '<<a[i]; } cout<<endl;
    Uredi<int>::uredi (a, n);
    cout << "Rastuci niz:";
    for (int i=0; i<n; i++) cout << ' ' << a[i]; cout << endl;
    Uredi<int,Opada<int> >::uredi (a, n);
    cout << "Opadajuci niz:";
    for (int i=0; i<n; i++) cout << ' ' << a[i]; cout << endl;
    delete [] a;
}

cout << "\nUredjivanje niza tacaka:\n\n";
int n; cin >> n; Tacka* a = new Tacka [n];
cout << "Pocetni niz:";
for (int i=0;i<n;i++) { cin>>a[i]; cout<< ' '<<a[i]; } cout<<endl;
Uredi<Tacka>::uredi (a, n);
cout << "Rastuci niz:";
for (int i=0; i<n; i++) cout << ' ' << a[i]; cout << endl;
Uredi<Tacka,Opada<Tacka> >::uredi (a, n);
cout << "Opadajuci niz:";
for (int i=0; i<n; i++) cout << ' ' << a[i]; cout << endl;
delete [] a;
}

cout << "\nUredjivanje niza pravougaonika:\n\n";
int n; cin >> n; Pravoug* a = new Pravoug [n];
cout << "Pocetni niz:";
for (int i=0;i<n;i++) { cin>>a[i]; cout<< ' '<<a[i]; } cout<<endl;
Uredi<Pravoug>::uredi (a, n);
cout << "Rastuci niz:";
for (int i=0; i<n; i++) cout << ' ' << a[i]; cout << endl;
Uredi<Pravoug,Opada<Pravoug> >::uredi (a, n);
cout << "Opadajuci niz:";
for (int i=0; i<n; i++) cout << ' ' << a[i]; cout << endl;
delete [] a;
}
```

Program 7.10 – Ispitivanje generičke klase za uređivanje nizova (uredit.C)

7.6.3 Uredjivanje nizova podataka

```
CC uredit.C -o uredit
uredit <uredit.pod
```

Uredjivanje niza celih brojeva:

```
Pocetni niz: 2 5 -3 7 -1 6 0 8 2 6 -2 2 6
Rastuci niz: -3 -2 -1 0 2 2 2 5 6 6 6 7 8
Opadajuci niz: 8 7 6 6 6 5 2 2 2 0 -1 -2 -3
```

Uredjivanje niza tacaka:

```
Pocetni niz: T(0,0) T(-2,-2) T(2,-3) T(3,3) T(3,2) T(1,1) T(-1,2)
Rastuci niz: T(0,0) T(1,1) T(-1,2) T(-2,-2) T(2,-3) T(3,2) T(3,3)
Opadajuci niz: T(3,3) T(3,2) T(2,-3) T(-2,-2) T(-1,2) T(1,1) T(0,0)
```

Uredjivanje niza pravougaonika:

```
Pocetni niz: P[1,4] P[2,3] P[3,3] P[1,2] P[3,4] P[2,3] P[3,4]
Rastuci niz: P[1,2] P[1,4] P[2,3] P[2,3] P[3,3] P[3,4] P[3,4]
Opadajuci niz: P[3,4] P[3,4] P[3,3] P[2,3] P[2,3] P[1,4] P[1,2]
```

Rezultat 7.3 – Uredjivanje nizova podataka programom 7.10

8 Standardna biblioteka Δ

Standard za jezik C++ definiše bogatu biblioteku gotovih klasa i funkcija za često korišćene složene strukture podataka (nizove, liste, skupove itd) i postupke (pretraživanje, uređivanje itd.). Većina tih klasa i funkcija su generičke klase, odnosno funkcije. Zbog toga se često koristi i naziv *standardna biblioteka šablonu (STL – Standard Template Library)*. Svrha te biblioteke je da pruža efikasna ostvarenja za često korišćene strukture podataka i postupke, a koji ne zavise od tipova podataka od kojih se struktura sastoji ili koji se obrađuju. Programer za svoje konkretnе tipove podatka može vrlo lako da dobije konkretnе klase i funkcije koje odgovarajuće radnje ostvaruju isto tako efikasno kao da su pisane upravo za te tipove podataka.

Sve bibliotečke klase mogu da se podele u nekoliko grupa:

- uslužne klase čije primerke koriste ostale klase u biblioteci,
- rad sa zbirkama različitih struktura (vektori, liste, skupovi, ...),
- rad s tekstovima,
- rad s brojevima (kompleksni brojevi, nizovi brojeva),
- ostvarenja često korišćenih postupaka (pretraživanje, uređivanje, ...),
- rad s datotekama,
- ...

U ovom poglavlju prikazane su važnije mogućnosti najčešće korišćenih funkcija i klasa iz veoma obimne standardne biblioteke.

Svi definisani identifikatori u standardnoj biblioteci stavljuju se u prostor imena std.

8.1 Usluge Δ

Standardna biblioteka jezika C++ sadrži veći broj jednostavnih generičkih funkcija i klasa za elementarne radnje koje omogućavaju veću fleksibilnost upotrebe složenijih klasa. U ovom odeljku su prikazane neke od njih. Pored njih u ovom odeljku je prikazana i generička klasa za rad s kompleksnim brojevima.

8.1.1 Relacioni operatori Δ

Postoje četiri generičke operatorske funkcije koje ostvaruju operatorske funkcije za operatorе !=, <=, > i >= pomoću operatora == i <. Njihov prototip u zaglavljу <utility> je:

template <typename T> bool operator rop (const T&, const T&);
gde je *rop* jedan od relacionih operatora !=, <=, > ili >=. Identifikatori ovih funkcija stavljeni su u unutrašnji prostor imena *rel_ops* unutar prostora imena *std*.

Ove funkcije omogućuju da programer za svoje klase treba da definiše samo operator-ske funkcije za operatore == i <. Za ostale relacije automatski će se izgenerisati odgovarajuće funkcije ako se za njih ukaže potreba.

Evo primera s jednostavnom klasom za pravougaonike koji se upoređuju na osnovu vrednosti njihovih površina:

```
class Pravoug {
    double a, b;
public:
    Pravoug (double aa=1, double bb=1) { a = aa; b = bb; }
    double P () const { return a * b; }
};

bool operator==(Pravoug x, Pravoug y) { return x.P() == y.P(); }
bool operator< (Pravoug x, Pravoug y) { return x.P() < y.P(); }
#include <utility>
using namespace std::rel_ops;
void f () {
    Pravoug a(1,5), b(2,2);
    if (a > b) ... // Generiše se operator>(Pravoug,Pravoug).
```

8.1.2 Parovi podataka Δ

Par podataka se sastoji od dva podatka međusobno proizvoljnih tipova. Definicija generičke klase za parove podataka u zaglavlju <utility> je:

```
template <typename A, typename B> struct pair {
    A first; B second;
    pair () : first(A()), second(B()) {}
    pair (const A&, const B&);
    template <typename C, typename D> pair (const pair<C,D>&);

    template <typename A, typename B>
    bool operator rop (const pair<A,B>&, const pair<A,B>&);

    template <typename A, typename B>
    pair<A,B> make_pair (const A&, const B&);
```

Struktura (klasa čiji su svi članovi javni) *pair<A,B>* sadrži po jedno javno polje tipa *A* i *B*. Tipovi *A* i *B* mogu da budu proizvoljnog, prostog i klasnog tipa, ali ne mogu biti nizovi.

Prvi konstruktor je podrazumevani konstruktor koji polja inicijalizuje njihovim podrazumevanim konstruktorima. Drugi konstruktor polja inicijalizuje kopijama svojih parametara. Treći konstruktor je generički konstruktor konverzije koji polja inicijalizuje kopijama odgovarajućih polja para koji je parametar konstruktora. Naravno, neophodno je da postoje konvezije *A(C)* i *B(D)* koje će se koristiti za pretvaranje tipova polja para koji se navodi kao argument u tipove polja para koji se stvara.

Postoje šest globalnih generičkih funkcija za svih šest relacijskih operatora (*rop* može da bude jedan od operatora ==, !=, <, <=, > ili >=). Dva para (a,b) i (c,d) su jednakia ako je *a==c&&b==d*. Par (a,b) je manji od para (c,d) ako je *a<c || a==c&&b<d*. Ostale relacije se izvode iz ove dve. U klasama *A* i *B* dovoljno je da su definisani operatori == i <.

Poslednja globalna generička funkcija *make_pair()* kao vrednost funkcije daje par sastavljen od kopija svojih parametara.

Evo primera korišćenja generičke klase *pair<>*:

```
#include <iostream>
#include <utility>
using namespace std;
int main () {
    pair<float,int> p;
    pair<int,char> q(1,'a');
    cout << p.first << ' ' << p.second << endl;
    cout << q.first << ' ' << q.second << endl;
}
```

8.1.3 Funkcijske klase i objekti Δ

Funkcijske klase su klase u kojima postoji operatorska funkcija za pozivanje funkcije () i čija je osnovna namena da se pozivanjem te funkcije obavi neka radnja s jednim ili dva argumentima. Na primer objekat *f* klase *F*:

```
class F {
    ...
public:
    ...
    T1 operator() (T2); // Unarna operacija.
    T3 operator() (T4, T5); // Binarna operacija.
    ...
};
```

može da se koristi u izrazima oblika *f(t2)* i *f(t4,t5)*.

Funkcijske klase su često generičke klase i tipovi parametara i rezultata metode *f()* su parametri šablona te klase.

U standardnoj biblioteci jezika C++ postoje generičke klase i funkcije čiji parametri su funkcijske klase. Te funkcijske klase omogućavaju korisniku da utiče na ponašanje bibliičkih klasa i funkcija.

U standardnoj biblioteci postoji veći broj generičkih funkcijskih klasa koje ostvaruju unarne i binarne aritmetičke i logičke operacije. Opšti oblici definicije tih klasa u zaglavlju <functional>, u zavisnosti od vrste operacija i broja operanada, su:

```
template <typename T> struct ime { // Unarna aritmetička operacija.
    T operator() (const T&);
}

template <typename T> struct ime { // Binarna aritmetička operacija.
    T operator() (const T&, const T&);
}

template <typename T> struct ime { // Unarna logička operacija.
    bool operator() (const T&);
}

template <typename T> struct ime { // Binarna logička operacija.
    bool operator() (const T&, const T&);
}
```

Klase plus, minus, multiplies, divides i modulus ostvaruju odgovarajuće binarne aritmetičke operacije pomoću operatora +, -, /, * i %. Klasa negate ostvaruje unarnu aritmetičku operaciju -.

Klase `equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal` i `less_equal` ostvaruju binarne relacione operacije pomoću operatora `==`, `!=`, `>`, `<`, `>=` i `<=` dajući logičke rezultate.

Klase `logical_and`, `logical_or` ostvaruju binarne logičke operacije pomoću operatora `&&` i `||`, a `logical_not` unarnu logičku operaciju pomoću operatora `~`.

Sve navedene klase mogu da se koriste za sve standardne proste tipove podataka. Za klasne tipove neophodno je da su korišćeni operatori preklopljeni operatorskim funkcijama.

U svim navedenim klasama operatorska funkcija `operator()` je ugrađena funkcija, pa se ne gubi ništa u efikasnosti ako se koriste te klase, a dobija se u fleksibilnosti.

Kao podrazumevana vrednost parametra bibliotečkih generičkih klasa često se navodi jedna od gore navedenih bibliotečkih funkcijskih klasa. Programer za svoje specifične potrebe može da definiše svoje funkcione klase.

Evo primera generičke funkcije koja stvara treći niz primenjujući neku operaciju na parove elemenata druga dva niza:

```
#include <iostream>
#include <functional>
using namespace std;

template <typename T, class Op>
void radi (const T a[], const T b[], T c[], int n) {
    for (int i=0; i<n; i++) c[i] = Op()(a[i], b[i]);
}

template <typename T> struct Zbir_kvadrata {
    T operator() (const T& a, const T& b) { return a*a+b*b; }
};

int main () {
    int a[10], b[10], c[10], n, i;
    cout << "n? "; cin >> n;
    cout << "A? "; for (int i=0; i<n; cin>>a[i++]);
    cout << "B? "; for (int i=0; i<n; cin>>b[i++]);
    radi<int,plus<int> > (a, b, c, n);
    for (int i=0; i<n; cout << c[i] << ' '); cout << endl;
    radi<int,Zbir_kvadrata<int> > (a, b, c, n);
    for (int i=0; i<n; cout << c[i] << ' '); cout << endl;
}
```

Šablon generičke funkcije `radi()` ima dva parametra, tip elemenata niza `T` i klasa `Op` koja definiše operaciju među elementima niza.

Generička klasa `Zbir_kvadrata` predstavlja primer nestandardne operacije, računanje zbirka kvadrata dva podatka tipa `T`.

U glavnoj funkciji, posle čitanja podataka, prvo se računaju zbirovi elemenata celobrojnih nizova `a` i `b`. Kao argumenti šablonu navedeni su tip `int` i klasa `plus <int>` koja se dobija iz bibliotečke generičke klase `plus<>`. Posle toga se računaju zbirovi kvadrata elemenata nizova `a` i `b` korišćenjem ranije navedene generičke klase `Zbir_kvadrata<>`.

8.1.4 Kompleksni brojevi (`complex`)

Za rad s kompleksnim brojevima u standardnoj biblioteci jezika C++ postoji generička klasa `complex<>`, koja je definisana u zaglavju `<complex>`, a čija deklaracija je:

```
template <typename T> class complex;
```

gde je `T` tip realnog i imaginarnog dela kompleksnog broja. Kao argumenti šablona u obzir dolaze tipovi `float`, `double` i `long double`.

Klase je projektovana tako da se rad s kompleksnim brojevima ne razlikuje od rada s bilo kojim numeričkim tipom podataka.

Kompleksni brojevi mogu da se stvaraju na osnovu pravouglih koordinata konstruktorom:

```
complex (const T& re=T(), const T& im=T());
```

Delovi kompleksnog broja mogu da se dohvate metodama:

```
T real () const; // Realni deo.
T imag () const; // Imaginarni deo.
```

Aritmetičke operacije `(+, -, *, /, =, +=, -=, *=, /=)` i operacije upoređivanja `(==, !=)` ostvarene su operatorskim funkcijama (metodama ili prijateljskim funkcijama).

Kompleksni brojevi prilikom čitanja operatorom `>>` mogu da se unesu u obliku `re`, `(re)` ili `(re, im)`, gde je `re` realni, a `im` imaginarni deo broja (u prva dva slučaja imaginarni deo biće nula). Kompleksni brojevi se operatorom `<<` pišu u obliku `(re, im)`.

Pored sledećih elementarnih funkcija za rad s kompleksnim brojevima:

```
T real (const complex<T>& z); // Realni deo.
T imag (const complex<T>& z); // Imaginarni deo.
T abs (const complex<T>& z); // Apsolutna vrednost (poteg)
T arg (const complex<T>& z); // Argument (nagib).
T norm (const complex<T>& z); // Norma (kvadrat potega).
complex<T> conj (const complex<T>& z); // Konjugovano kompleksan br.
complex<T> polar (const T& r, // Kompleksan broj na osnovu
                  const T& fi); // polarnih koordinata.
```

postoje i funkcije za uobičajene matematičke funkcije s kompleksnim parametrom `z` i kompleksnim rezultatom: `sqrt(z)`, `exp(z)`, `log(z)`, `log10(z)`, `sin(z)`, `cos(z)`, `tan(z)`, `sinh(z)`, `cosh(z)` i `tanh(z)`. Pored njih postoji i funkcija za stepenovanje `pow(a, b)`. Ako je parametar `a` kompleksan broj, argument `b` može da bude ceo broj (tip `int`), realan broj (tip `T`) ili kompleksan broj. Ako je argument `b` kompleksan broj argument `a` može da bude i realan broj (tip `T`).

8.2 Zbirke podataka

Zbirke su objekti koji sadrže izvestan broj podataka nekog tipa. Omogućuju dodavanje novih elemenata zbirci, dohvatanje postojećih elemenata u zbirci i izbacivanje nepotrebnih elemenata iz zbirke.

Zbirke mogu da imaju različite unutrašnje strukture. Unutrašnja struktura zbirke može da utiče na brzinu pristupanja pojedinim elementima, brzinu izvođenja pojedinih radnji nad zbirkom (dodavanje i izbacivanje elemenata) i na utrošak memorije za uskladištanje zbirke. Na primer jednodimenzionalni nizovi (vektori) omogućuju efikasan pristup elementima niza po proizvoljnom redosledu, ali nisu pogodni za umetanje ili izbacivanje elemenata u

sredini niza. S druge strane kod listi upravo umetanje i izbacivanje elemenata se izvodi efikasno, ali liste troše više memorije za uskladištanje podataka i nisu pogodne za pristup elementima po proizvoljnom redosledu.

Pored podrške zbirkama za pristupanje pojedinim elementima, zbirkama mogu da se pridruže i specijalni objekti za obilazak elemenata zbirke, najčešće redom, radi izvođenja neke radnje nad svakim elementom pojedinačno. Takvi objekti se nazivaju **iteratori**. Njihove klase u standardnoj biblioteci su ostvarene kao unutrašnje klase odgovarajućih klasa za zbirke.

Klase za zbirke čine skup nezavisnih klasa u smislu da one nisu izvedene iz neke zajedničke osnovne klase, uprkos vrlo ujednačenom spoljašnjem izgledu (većina radnji postoji u svima njima i metode za njihovo ostvarenje imaju ista imena i iste parametre). Ovo isto važi i za iteratorske klase.

Sve zbirke u standardnoj biblioteci mogu da se podele u dve grupe:

- **sekvenčne zbirke** kod kojih redosled elemenata u zbirci korisnik određuje neposredno;
- **asocijativne zbirke** kod kojih su elementi zbirke uređeni po nekom kriterijumu i zbog toga na redosled elemenata korisnik može da utiče samo posredno, navođenjem odredene strategije uređivanja.

Od podataka koji se stavljuju u zbirke očekuje se da mogu da se inicijalizuju kao prazni objekti, da se inicijalizuju kopijama sadržaja drugih podataka istog tipa, da se pri dodeljivanju vrednosti kopira sadržaj izvorišnog podatka u odredišni podatak i da podaci mogu da se uništavaju. U slučaju asocijativnih zbirki očekuje se još da može da se ispita da li su dva podatka jednaka ili da li je jedan podatak manji od drugog.

Za standardne tipove podataka svi ti uslovi su ispunjeni.

Za klasne tipove, projektant za klase čije primerke stavlja u zbirku mora da obezbedi podrazumevani konstruktor, konstruktor kopije, destruktorni i operatorske funkcije `=`, `==` i `<`. Skreće se pažnja na to da su od relacijskih operatora potrebna samo navedena dva. Ostale četiri relacije u svim zbirkama ostvaruju se pomoću njih.

Ako su elementi zbirke klasnih tipova prilikom stavljanja podataka u zbirke uvek se stavlja kopija podatka stvorena pozivanjem konstruktora kopije. Prilikom izbacivanja elemenata iz zbirke uvek se pozove destruktorni za izbacivane elemente. Konstruktori kopije se pozivaju i prilikom kopiranja sadržaja zbirke ili dela zbirke u drugu zbirku. Destruktorni se pozivaju i prilikom uništavanja zbirki.

Skreće se pažnja na to da ako su elementi zbirke pokazivači prilikom kopiranja kopiraju se samo pokazivači, a ne i pokazivani podaci. Ako su pokazivani podaci klasnih tipova, neće se pozivati konstruktori kopije. Isto tako, prilikom uništavanja zbirke koja sadrži pokazivače na objekte, ne pozivaju se destruktorni za pokazivane objekte.

8.2.1 Iteratori Δ

Elementi u zbirkama, bez obzira na strukturu zbirke, nalaze se po određenom redosledu. Za svaki element, osim prvog i poslednjeg, postoji prethodni i naredni element. Elementima zbirke pridružuju se redni brojevi počev od nula.

Primerak iteratorske klase, skraćeno **iterator**, svakog momenta omogućava pristup jednom elementu zbirke koji se naziva **tekući element**. Radnje nad iteratorima obuhvataju pomeranje iteratora s jednog elementa na drugi (redom ili po slučajnom redosledu), pristup tekućem elementu i upoređivanje dva iteratora (da li pokazuju na isti element zbirke).

8.2.1.1 Iteratori Δ

Iteratori mogu da se zamisle kao pokazivači koji pokazuju na elemente nizova. Sve operacije nad iteratorima su ostvarene preklapanjem operatora tako da nema razlike između izraza s običnim pokazivačima i izraza s iteratorima. Tako, na primer, ako iterator i pokazuje na neki element zbirke, $i+1$ pokazuje na naredni, a $i-1$ na prethodni element zbirke (bez obzira da li zbirka ima strukturu niza, liste ili neku drugu strukturu). Takođe $*i$ predstavlja vrednost tekućeg elementa zbirke.

Pored pokazivanja na pojedine elemente zbirke, kao specijalna vrednost, iteratori mogu da pokazuju i iza poslednjeg elementa zbirke. U tom slučaju nije dozvoljen pristup elementu, ali nad zbirkama mogu postojati radnje koje i tada mogu da se izvode. Na primer, dodavanje elementa ispred tekućeg elementa u tom slučaju, u stvari, označava dodavanje novog elementa iza poslednjeg.

Prema mogućim operacijama nad iteratorima, iteratori mogu da se podele u sledeće vrste:

- **izlazni iteratori** omogućuju samo postavljanje vrednosti tekućeg elementa i kretnjanje od početka ka kraju zbirke,
- **ulazni iteratori** omogućuju samo dohvatanje vrednosti tekućeg elementa i kretnjanje od početka ka kraju zbirke,
- **jednosmerni iteratori** omogućuju sve radnje koje postoje kod izlaznih i ulaznih iteratora zajedno,
- **dvosmerni iteratori** pored mogućnosti jednosmernih iteratora, omogućuju i kretnjanje od kraja ka početku zbirke,
- **iteratori s proizvoljnim pristupom** pored mogućnosti dvosmernih iteratora omogućuju pristup elementima zbirke po proizvoljnom redosledu.

Iteratori s više mogućnosti uvek mogu da se koriste za radnje nad zbirkama za koje su dovoljni iteratori s manje mogućnosti.

Tablica 8.1 prikazuje operatore koji mogu da se koriste za pojedine vrste iteratora.

Izlazni iteratori	Ulazni iteratori	Jednosmerni iteratori	Dvosmerni iteratori	Iteratori s proizvoljnim pristupom
$I(i) =$	$I(i) =$	$I(i) =$	$I(i) =$	$I(i) =$
$*i =$	$= *i$	$*i$	$*i$	$*i$
				$[]$
		$->$	$->$	$->$
	$++$	$++$	$++ --$	$++ -- + - += -=$
	$== !=$	$== !=$	$== !=$	$== != < > <= >=$

Tablica 8.1 – Operatori nad iteratorima

Iteratori svih vrsta mogu da se inicijalizuju drugim iteratorima ($I(j(i))$) i da im se dodeljuju vrednosti drugih iteratora ($j=i$). Posle toga oba iteratora pokazuju na isti element iste zbirke.

Vrednost izraza $*i$ je tekući element zbirke. Kod izlaznog iteratora sme da se koristi samo s leve strane (postavljanje vrednosti tekućeg elementa), a kod ulaznog iteratora samo s desne strane (dohvatanje vrednosti) operatora $=$. Kod svih ostalih vrsta iteratora pomoću izraza $*i$ može da se postavlja i da se dohvata vrednost tekućeg elementa.

Indeksiranje je relativno u odnosu na tekući element ($i[1]$ je vrednost narednog, a $i[-1]$ vrednost prethodnog elementa u odnosu na tekući element $i[0]$, odnosno $*i$).

Operatorom \rightarrow može da se pristupa članovima tekućeg elementa zbirke kada su oni klasnih tipova.

Operatorima $++ i$ – naredni, odnosno prethodni element zbirke će postati novi tekući element.

Vrednost izraza $i+k$, gde je k ceo broj, pokazuje na k -ti element od tekućeg elementa prema kraju zbirke. Izrazom $i+=k$ iterator se pomera za k mesta prema kraju zbirke i element na tom mestu će postati novi tekući element. Ako je k negativno pomeranje se dešava ka početku zbirke. Naravno, operatorima $- i$ – postiže se pomeranje u suprotnom smjeru nego s operatorima $+ i +=$.

Operatori za uporedivanje daju rezultate tipa **bool**. Vrednost izraza $i < j$ je **true** ako iterator i pokazuje na element koji je bliže početku zbirke od elementa na koji pokazuje iterator j . Rezultat nema smisla ako i i j ne pokazuju na elemente iste zbirke.

Iteratori, bez obzira na vrstu, mogu da budu jednog od sledećih tipova:

- **iterator** – direktni iterator za promenljive zbirke,
- **const_iterator** – direktni iterator za nepromenljive zbirke,
- **reverse_iterator** – obrnuti iterator za promenljive zbirke,
- **const_reverse_iterator** – obrnuti iterator za nepromenljive zbirke.

Kod direktnih iteratora vrednost iteratora raste od početka do kraja zbirke ($i+1$ pokazuje na sledeći element u odnosu na tekući). Kod obrnutih iteratora vrednost iteratora raste od kraja ka početku zbirke ($i+1$ pokazuje na prethodni element u odnosu na tekući).

Iteratori za promenljive zbirke omogućuju menjanje sadržaja zbirke bilo umetanjem ili izbacivanjem elemenata, bilo menjanjem vrednosti elemenata. Pomoću iteratora za nepromenljive zbirke ne može da se promeni sadržaj zbirke ni umetanjem ili izbacivanjem elemenata, ni menjanjem vrednosti elemenata.

Skreće se pažnja na to da su iteratorske klase definisane kao unutrašnje klase u klasama zbirki. Zbog toga prilikom navođenja njihovog tipa neophodno je, pomoću operatora za razrešenje dosega $(::)$, navesti ime odgovarajuće klase za zbirke. Na primer generička klasa **vector<E>** (videti odeljak 8.2.2) je najčešće korišćena klasa za zbirke elemenata tipa *E*. Oznaka tipa direktnog iteratora za tu klasu je **vector<E>::iterator**. Koja je to vrsta iteratora zavisi od same zbirke. Na primer, kod vektora to je iterator s proizvoljnim pristupom.

Evo primera koji korišćenjem iteratora napuni vektor celih brojeva čitajući podatke preko tastature i potom ih ispiše na ekranu po suprotnom redosledu (za detalje korišćenja klase **vector<E>** videti odeljak 8.2.2):

```
vector<int> v(10);
for (vector<int>::iterator i = v.begin();
     i != v.end();
     cin << *i++);
for (vector<int>::reverse_iterator i = v.rbegin();
     i != v.rend();
     cout << *i++ << ' ');
cout << endl;
```

Prvom naredbom se definiše vektor (jednodimenzionalni niz) celih brojeva od 10 elemenata jednakih nuli.

8.2.1 Iteratori Δ

Vrednost metode **begin()** je direktni iterator koji pokazuje na prvi element, a metode **end()** iterator koji pokazuje iza poslednjeg elementa tekuće zbirke.

Slično tome, vrednost metode **rbegin()** je obrnuti iterator koji pokazuje na prvi element od kraja (tj. na poslednji element), a metode **rend()** iterator koji pokazuje iza poslednjeg elementa od kraja (tj. ispred prvog elementa) tekuće zbirke.

Skreće se pažnja na to da se vrednost iteratora u oba ciklusa povećava ($i++$). Kod direktnog iteratora time se kreće ka kraju, a kod obrnutog iteratora ka početku zbirke.

8.2.2 Vektori (**vector**) Δ

Vektori su jednodimenzionalni nizovi podataka. Osnovna osobina vektora je efikasan pristup elementima po proizvoljnom redosledu i efikasno dodavanje i izbacivanje elemenata na kraju vektora. Dodavanje i izbacivanje elemenata na početku i u unutrašnjosti vektora je neefikasno.

Vektori se mogu efikasno koristiti i kao stekovi. Kraj vektora predstavlja vrh steka gde se podaci stave na stek i odakle se podaci uzimaju sa steka.

Generička klasa **vector<E>** u standardnoj biblioteci jezika C++ vektore ostvaruje u obliku dinamičkih nizova koji imaju određeni kapacitet za uskladištanje elemenata vektora. Stvaran broj elemenata u vektoru nikada nije veći od trenutnog kapaciteta. Prilikom dodavanja i izbacivanja elemenata kapacitet vektora se menja automatski. Standard ne propisuje strategiju dodeljivanja memorije vektoru. Promena kapaciteta je dugotrajan zahvat pa Standard dozvoljava da automatska promena kapaciteta ne bude samo za jedno mesto, već tako da bude i nekoliko rezervnih slobodnih mesta za eventualne buduće promene broja elemenata vektora.

Vektori spadaju u grupu sekvenčnih zbirki i za pristup elementima koriste iteratore s proizvoljnim pristupom.

Deklaracija generičke klase **vector<E>**, koja je definisana u zaglavlju **<vector>**, je:

```
template <typename E> class vector;
```

gde je *E* tip elemenata vektora.

Klasa **vector<E>** poseduje podrazumevani konstruktor koji stvara prazan vektor, konstruktor kopije, operatori za dodelu vrednosti i destruktör. U nastavku su navedene preostale metode klase **vector<E>** i globalne funkcije za rad s vektorima.

Za obeležavanje nekoliko uzastopnih elemenata zbirke koriste se dva iteratora **prvi** i **posl**, koji uvek označavaju poluzatvoreni interval $[prvi, posl]$. Iterator **prvi** pokazuje na prvi element koji pripada intervalu, a iterator **posl** na prvi element koji ne pripada intervalu. Posledice su nepredvidljive ako iteratori **prvi** i **posl** ne pokazuju na elemente iste zbirke, ili ako **posl** pokazuje na raniji element nego **prvi**.

```
vector (unsigned vel, const E& x=E());
```

Ovaj konstruktor stvara vektor koji sadrži *vel* elemenata inicijalizovanih kopijama parametra *x*. Podrazumevana vrednost elemenata je nula za proste tipove podataka, odnosno rezultat podrazumevanog konstruktora za klasne tipove podataka.

```
template <class It> vector (It prvi, It posl);
```

Ovaj generički konstruktor stvara vektor kopirajući sve uzastopne elemente iz neke zbirke iz intervala $[prvi, posl]$. Izvorišna zbirka je određena vrednostima korišćenih iteratora i ne mora da bude vektor, već može da bude zbirka bilo koje vrste. Šta više,

izvorišna zbirka može da sadrži podatke različitog tipa od tipa elemenata stvaranog vektora, pod uslovom da postoji konverzija tipa elemenata izvorišne zbirke u tip elemenata vektora.

```
void assign (unsigned n, const E& x=E());
template <class It> void assign (It prvi, It posl);
```

Ove metode zamenjuju sadržaj vektora novim vrednostima. To podrazumeva prethodno uništavanje zatečenog sadržaja vektora.

Kod prve metode novi sadržaj vektora biće *n* kopija parametra *x*.

Kod druge metode novi sadržaj vektora biće kopije uzastopnih elemenata druge zbirke iz intervala *[prvi, posl]*.

```
unsigned size () const;
unsigned max_size () const;
bool empty () const;
```

Vrednost prve metode je trenutni broj elemenata vektora, druge najveći mogući broj elemenata vektora, a treće indikator da li je vektor prazan.

```
void resize (unsigned vel, const E& x=E());
```

Ova metoda menja veličinu vektora u *vel* elemenata. Ako vektor ima više od *vel* elemenata uništice se potreban broj zadnjih elemenata. Ako vektor ima manje od *vel* elemenata dodaće se potreban broj elemenata iza poslednjeg elementa vektora, čije vrednosti će biti jednake kopiji vrednosti parametra *x*. Ako je *vel*>*max_size()*, prijavljuje se izuzetak tipa *length_error*.

```
unsigned capacity () const;
```

Vrednost ove metode je trenutni kapacitet vektora.

```
void reserve (unsigned n);
```

Ova metoda obezbeđuje da vektor ima mesta za usklađivanje najmanje *n* elemenata. Ako je *n* manji od trenutnog kapaciteta novi kapacitet neće biti manji od broja elemenata vektora (*size()*).

Ova metoda je korisna kada se очekuje dodavanje većeg broja elemenata na kraj niza. Povećanjem kapaciteta odjednom na očekivani broj elemenata može znatno da se ubrza izvršavanje programa.

```
iterator begin ();
const_iterator begin () const;
iterator end ();
const_iterator end () const;
```

Vrednost prve dve metode je direktni iterator koji pokazuje na prvi element, a druge dve metode iza poslednjeg elementa vektora u smeru od početka ka kraju vektora. Prva i treća metoda se koriste za promenljive, a druga i četvrta za nepromenljive vektore.

```
reverse_iterator rbegin ();
const_reverse_iterator rbegin () const;
reverse_iterator rend ();
const_reverse_iterator rend () const;
```

Vrednost prve dve metode je obrnut iterator koji pokazuje na prvi element, a druge dve iza poslednjeg elementa vektora u smeru od kraja ka početku vektora (što znači, na

8.2.2 Vektori (vector) ▾

poslednji element i ispred prvog elementa vektora). Prva i treća metoda se koriste za promenljive, a druga i četvrta za nepromenljive vektore.

```
E& operator[] (unsigned ind);
const E& operator[] (unsigned ind) const;
E& at (unsigned ind);
const E& at (unsigned ind) const;
```

Vrednost ovih metoda je upućivač na element vektora s rednim brojem *ind*. Prva i treća metoda se koriste za promenljive, a druga i četvrta za nepromenljive vektore. Operator za indeksiranje [] ne proverava da li je vrednost indeksa *ind* u dozvoljenim granicama ($0 \leq ind < size()$) i zato je efikasniji od metode *at()* koja to proverava i prijavljuje izuzetak tipa *out_of_range* ako vrednost indeksa nije dozvoljena. Indeksiranje treba koristiti kada je sigurno da je indeks ispravan, a metodu *at()* ako to nije slučaj. Na primer:

```
vector<int> v(10);
for (unsigned i=0; i<size(), cin>>v[i++]);
unsigned ind; cin >> ind; cout << v.at(ind);
E& front();
const E& front() const;
E& back();
const E& back() const;
```

Vrednost prve dve metode je upućivač na prvi element, a druge dve na zadnji element vektora. Prva i treća metoda se koriste za promenljive, a druga i četvrta za nepromenljive vektore.

```
void push_back (const E& x);
void pop_back ();
```

Prva metoda stavlja kopiju parametra *x* iza poslednjeg elementa vektora. Time se veličina vektora povećava za jedan.

Druga metoda izbacuje poslednji element iz vektora. Time se veličina vektora smanjuje za jedan.

```
iterator insert (iterator poz, const E& x=E());
void insert (iterator poz, unsigned n, const E& x);
template <class It> void insert(iterator poz, It prvi, It posl);
```

Ove metode umeću jedan ili više podataka ispred elementa vektora na koji pokazuje iterator *poz*.

Prva metoda umeće kopiju parametra *x*. Vrednost metode je iterator koji pokazuje na novi element vektora.

Druga metoda umeće *n* primeraka kopije parametra *x*.

Treća metoda umeće sve uzastopne elemente neke zbirke iz intervala *[prvi, posl]*.

```
iterator erase (iterator poz);
iterator erase (iterator prvi, iterator posl);
```

Prva metoda izbacuje element vektora na koji pokazuje iterator *poz*.

Druga metoda izbacuje sve uzastopne elemente vektora iz intervala [*prvi*, *posl*].

Vrednost obe metode je iterator koji pokazuje na element neposredno iza poslednjeg izbačenog elementa.

```
void clear ();
```

Ova metoda izbacuje sve elemente vektora. Posle toga veličina vektora je nula.

```
bool operator rop (const vector<E>& v1, const vector<E>& v2);
```

Ova grupa operatorskih funkcija leksikografski upoređuje sadržaj vektora *v1* i *v2*.

Relacioni operator *rop* može da bude ==, !=, <, <=, > ili >=.

Dva vektora su jednaka (*v1*==*v2*) ako imaju isti broj elemenata i ako su u njima elementi s istim rednim brojevima jednakci.

Prvi vektor je manji od drugog vektora (*v1*<*v2*) ako je kod prvog para različitih elemenata element u prvom vektoru manji od elementa u drugom vektoru. Ako je prvi vektor kraći od drugog i svi njegovi elementi su jednaki odgovarajućim elementima drugog vektora, prvi vektor se smatra manjim.

Ostale relacije između dva vektora izvode se iz prethodne dve relacije.

```
void swap (vector<E>& v);
```

Ova metoda međusobno zamenjuje sadržaj tekućeg vektora (**this*) i vektora *v*.

Evo primera za korišćenje vektora:

```
#include <iostream>
#include <vector>
using namespace std;

int main () {
    vector<int> v; int k;
    cout << "niz? ";
    while (cin>>k, k!=9999) {
        if (v.size()==v.capacity()) v.reserve(v.size()+10);
        v.push_back(k);
    }
    v.reserve(v.size());
    int s = 0; for (unsigned i=0; i<v.size(); s+=v[i++]);
    cout << "s= " << s << endl;
}
```

Prvo se definiše prazan vektor celih brojeva (*vector<int>*).

Zatim se čita niz celih brojeva nepoznate dužine. Pročitani brojevi se stavljuju u vektor celih brojeva jedan po jedan. Da bi se izbeglo povećavanje kapaciteta vektora pri svakom dodavanju broja, kad god se vektor napuni (*v.size()*==*v.capacity()*), traži se dodata memorije za još 10 elemenata (*v.size()*+10). Po izlasku iz ciklusa čitanja brojeva kapacitet vektora se smanjuje na tačno onoliko mesta koliko ima elemenata u vektoru.

U drugom delu programa izračunava se zbir elemenata vektora. Za pristup elementima koristi se indeksiranje (bez provere vrednosti indeksa).

8.2.3 Redovi s dva kraja (deque) △

Redovi s dva kraja su dinamički nizovi koji pored dobrih osobina vektora iz prethodne tačke omogućuju efikasno dodavanje i izbacivanje elemenata i na početku reda. Mogu da se koriste kao opšti jednodimenzionalni nizovi, stekovi i redovi kroz koje podaci mogu da prolaze u dva smera: od početka ka kraju ili od kraja ka početku. Iz ove dvosmernosti potiče i njihovo ime.

Redovi s dva kraja u standardnoj biblioteci jezika C++ ostvaruju se generičkom klasom *deque*<> (*double-ended queue*), čija deklaracija u zagлавju <*deque*> je:

```
template <typename E> class deque;
```

gde je *E* tip elementa reda.

Redovi s dva kraja spadaju u grupu sekvenčijalnih zbirk i koriste iteratore s proizvoljnim pristupom.

Redovi s dva kraja mogu da se koriste na isti način kao i vektori. Jedino ne može da se utiče na određivanje kapaciteta redova. To znači da postoje sve metode, izuzev metoda *capacity()* i *reserve()*, i sve globalne funkcije kao i u slučaju vektora iz prethodne tačke. Pored toga u klasi *deque*<> postoje i sledeće dve metode:

```
void push_front (const E& x);
void pop_front();
```

Prva metoda stavlja kopiju parametra *x* ispred prvog elementa reda s dva kraja. Druga metoda izbacuje prvi element iz reda s dva kraja.

8.2.4 Liste (list) △

Osnovna osobina liste je da, za razliku od vektora, omogućuju efikasno umetanje i izbacivanje elemenata i u unutrašnjosti liste, a ne samo na krajevima. S druge strane, pristupanje elementima liste po proizvoljnem redosledu je vrlo neefikasno.

Liste u standardnoj biblioteci jezika C++ ostvaruju se generičkom klasom *list*<>, koja je definisana u zaglavju <*list*>, a čija deklaracija je:

```
template <typename E> class list;
```

gde je *E* tip elemenata liste.

Liste spadaju u grupu sekvenčijalnih zbirk i koriste dvosmerne iteratore. Za liste postoje sve metode i globalne funkcije kao za redove s dva kraja, izuzev operatora za indeksiranje ([] i metode *at()*). S druge strane postoji veći broj metoda za složenije zahvate nad listama.

```
void splice (iterator poz, list<E>& lst);
void splice (iterator poz, list<E>& lst, iterator i);
void splice (iterator poz, list<E>& lst,
             iterator prvi, iterator posl);
```

Ove metode ubacuju ispred elementa liste na koji pokazuje iterator *poz*. Elemente druge liste *lst*. Ne prave se kopije elemenata liste *lst*, već se odabrani elementi prebacuju u određenu listu.

Prva metoda prebacuje sve elemente liste *lst*. Posle toga *lst* je prazna lista.

Druga metoda prebacuje element liste *lst* na koji pokazuje iterator *i*.

Treća metoda prebacuje sve elemente liste *lst* iz intervala [*prvi*, *posl*].

8.2.5 Specijalne zbirke (`queue`, `priority_queue` i `stack`)

Generičke klase `queue<E>` i `priority_queue<E>` su definisane u zaglavlju `<queue.h>`, a generička klasa `stack<E>` u zaglavlju `<stack.h>`. Deklaracije tih klasa su:

```
template <typename E, class Z=deque<E>> class queue;
template <typename E, class Z=vector<E>> class priority_queue;
template <typename E, class Z=stack<E>> class stack;
```

`E` je tip elemenata zbirki.

`Z` je tip sekvenčalne zbirke čije usluge se koriste. Podrazumevani tip uslužne zbirke za redove je `deque<E>`, a u obzir dolazi još tip `list<E>`. Podrazumevani tip uslužne zbirke za prioritetne redove je `vector<E>`, a u obzir dolazi još tip `deque<E>`. Podrazumevani tip uslužne zbirke za stekove je `deque<E>`, a u obzir dolaze još tipovi `vector<E>` i `list<E>`.

`U` je tip funkcijskog objekta koji se koristi za upoređivanje elemenata prioritetnog reda. Podrazumevano se koristi bibliotečka klasa `less<E>` čiji primerci za upoređivanje koriste operator `<` (videti odeljak 8.1.3). U obzir dolaze klase za koje je vrednost izraza $u(e_1, e_2)$ jednaka `true` ako je $e_1 < e_2$, gde je u primerak klase `U`.

Od metoda i funkcija prikazanih u odeljku 8.2.2 u sve tri klase postoje samo metode `size()`, `empty()`. U klasi `queue<E>` postoje još i metode `back()`, `front()` za dohvatanje elementa na kraju i na početku reda. Pored njih, uz klase `queue<E>` i `stack<E>` postoje i globalne operatorske funkcije za upoređivanje zbirki (`==`, `!=`, `<`, `<=`, `>` i `>=`). U nastavku su navedene preostale metode klase posmatranih specijalnih zbirki.

```
explicit queue (const Z& z=Z());
explicit priority_queue (const U& u=U(), const Z& z=Z());
explicit stack (const Z& z=Z());
```

Ovi konstruktori stvarani objekat inicijalizuju kopijom sadržaja zbirke `z` tipa `Z` (`Z` je parametar šablona), podrazumevano praznom zbirkom. Parametar `u` za prioritetne redove predstavlja objekat tipa `U` (`U` je parametar šablona) koji se koristi za upoređivanje elemenata reda.

```
E& top();
const E& top() const;
```

Vrednost ovih metoda je upućivač na element na početku prioritetnog reda, odnosno na vrhu steka. Za prioritetne redove postoji samo druga metoda.

```
void push (const E& x);
```

Ova metoda stavlja kopiju parametra `x` u zbirku. Zavisno od vrste zbirke to znači na kraj reda, na mesto koji odgovara prioritetu kod prioritetnog reda, odnosno na vrh steka.

```
void pop();
```

Ova metoda izbacuje element s početka reda ili prioritetnog reda, odnosno s vrha steka.

Kod druge dve metode izvorišna lista (`lst`) i odredišna lista (`*this`) mogu biti ista lista. Tada se jedan ili više elemenata liste premeštaju na drugo mesto unutar iste liste. Posledice su nepredviđljive ako iterator `pos` pokazuje na element liste koji se nalazi između elemenata na koje pokazuju iteratori `prvi` i `pos1`.

```
void remove (const E& x);
```

Ova metoda izbacuje iz liste sve elemente koji su jednaki parametru `x`.

```
template <class U> void remove_if (U u);
```

Ova metoda izbacuje iz liste sve elemente `e` za koje je vrednost izraza `u(e)` jednaka `true`.

```
void unique ();
```

```
template <class U> void unique (U u);
```

Ove metode redukuju listu tako da u svakoj sekvenci uzastopno jednakih elemenata zadrže samo prvi element i izbacuju preostale.

Prva metoda za upoređivanje elemenata koristi operator `==`. Druga metoda za svaki par uzastopnih elemenata `e1` i `e2` izbacuje jedan ako je vrednost izraza `u(e1, e2)` jednaka `true`.

```
void merge (list<E>& lst);
```

```
template <class U> void merge (list<E>& lst, U u);
```

Ove metode ubacuju između elemenata uređene tekuće liste (`*this`) elemente druge, na isti način uređene liste `lst` tako da tekuća lista i dalje bude uređena. Elementi liste `lst` se ne kopiraju i na kraju lista `lst` je prazna.

Prva metoda za upoređivanje elemenata koristi operator `<` (neopadajući poredak).

Dруга metoda smatra da je element `e1` ispred elementa `e2` ako je vrednost izraza `u(e1, e2)` jednaka `true`.

```
void sort ();
```

```
template <class U> void sort(U u);
```

Ove metode uređuju elemente liste po određenom poretku. Prva metoda za upoređivanje elemenata koristi operator `<` (neopadajući poredak). Druga metoda smatra da je element `e1` ispred elementa `e2` ako je vrednost izraza `u(e1, e2)` jednaka `true`.

```
void reverse ();
```

Ova metoda obrne redosled elemenata liste.

8.2.5 Specijalne zbirke (`queue`, `priority_queue` i `stack`)

U standardnoj biblioteci jezika C++ postoje tri specijalne generičke klase koje svoje funkcionalnosti ostvaruju korišćenjem usluga jedne od sekvenčalnih zbirki iz prethodnih tačaka. To su:

- `queue<E>` – redovi kod kojih se podaci dodaju na kraju reda, a uzimaju se s početka reda,
- `priority_queue<E>` – prioritetni redovi kod kojih se podaci uzimaju po opadajućem redosledu prioriteta (za elemente jednakih prioriteta Standard ne propisuje redosled dolaženja na početak reda),
- `stack<E>` – stekovi kod kojih se podaci stavljaju na vrh i uzimaju s vrha.

8.2.6 Preslikavanja (map i multimap) △

Preslikavanja u standardnoj biblioteci jezika C++ su zbirke čiji su elementi parovi vrednosti (*ključ, podatak*) i koje omogućuju brzo pronađenje vrednosti podatka na osnovu vrednosti ključa. Elementi se drže uređeno po vrednostima ključeva i zbog toga spadaju u grupu asocijativnih zbirki.

Kaže se da ovakve zbirke vrše preslikavanje ključeva u pridružene podatke. Preslikavanje može da bude jednoznačno i višežnačno. U prvom slučaju svakom ključu odgovara jedna slika, što drugim rečima znači da se isti ključ ne pojavljuje u više parova. Kod višežnačnog preslikavanja isti ključ može da se preslika u više podataka, tj. isti ključ može da se pojavljuje u više parova (ključ, podatak).

Preslikavanja koriste dvostrane iteratore što omogućuje njihov obilazak po rastućem i po opadajućem poretku ključeva. Iteratori pokazuju na parove (ključ, podatak).

Jednoznačna preslikavanja se ostvaruju generičkom klasom `map<T, U>`, a višežnačna generičkom klasom `multimap<T, U>`. Obe klase su definisane u zagлавljima `<map>` i `<multimap>`, a njihove deklaracije su:

```
template <typename K, typename P, class U=less<K>> class map;
template <typename K, typename P, class U=less<K>> class multimap;
```

K je tip ključa, a *P* tip pridruženog podatka. Elementi zbirki su primerci generičke klase `pair<const K, P>` (videti odeljak 8.1.2).

U je tip funkcijskog objekta koji se koristi za upoređivanje ključeva. Podrazumevano se koristi bibliotečka klasa `less<K>` čiji primerci za upoređivanje koriste operator < (videti odeljak 8.1.3). U obzir dolaze klase za koje je vrednost izraza *u(k1, k2)* jednaka `true` ako je *k1* ispred *k2*, gde je *u* primerak klase *U*.

Generičke klase `map<T, U>` i `multimap<T, U>` sadrže podrazumevani konstruktor koji stvara praznu zbirku, konstruktor kopije, operator za dodelu vrednosti i destruktork. Sadrže metode `size()`, `max_size()`, `empty()`, `begin()`, `end()`, `rbegin()`, `rend()`, `swap()` i `clear()`, a postoje i globalne operatorske funkcije za operatore `==`, `!=`, `<`, `<=`, `>`, `>=` s istim značenjima kao i kod vektora (videti odeljak 8.2.2).

U nastavku su navedene preostale metode generičkih klasa `map<T, U>` i `multimap<T, U>`. Tipovi *K*, *P* i *U* svuda označavaju parametre šablonata generičke klase. Tip *E* označava tip elemenata zbirke, a to je `pair<K, P>`.

```
explicit map (const U& u=U());
explicit multimap (const U& u=U());
```

Ovi konstruktori stvaraju praznu zbirku preslikavanja kojoj pridružuju funkcijski objekat *u* za upoređivanje ključeva.

```
template <class It> map (It prvi, It posl, const U& u=U());
template <class It> multimap (It prvi, It posl, const U& u=U());
```

Ovi konstruktori stvaraju zbirku preslikavanja inicijalizuju kopijama svih elemenata tipa *E* druge zbirke iz intervala `[prvi, posl]`. Stvorenoj zbirci pridružuju funkcijski objekat *u* za upoređivanje ključeva.

```
unsigned count (const K& k) const;
```

Vrednost ove metode je broj elemenata zbirke s ključem *k*.

```
iterator           (const K& k);
find              (const K& k) const;
const_iterator    (const K& k) const;
lower_bound       (const K& k) const;
upper_bound       (const K& k) const;
const_iterator    (const K& k) const;
pair<iterator, iterator> equal_range (const K& k);
pair<const_iterator, const_iterator>
                     equal_range (const K& k) const;
```

Vrednost prvog para metoda je iterator koji pokazuje na prvi element zbirke čiji ključ je jednak *k*.

Vrednost drugog para metoda je iterator koji pokazuje na prvi element zbirke čiji ključ je veći ili jednak *k*.

Vrednost trećeg para metoda je iterator koji pokazuje na prvi element zbirke čiji ključ je veći od *k*.

Prvi član vrednosti poslednjeg para metoda je iterator koji pokazuje na prvi element zbirke čiji ključ je veći ili jednak *k*, a drugi član je iterator koji pokazuje na prvi element zbirke čiji ključ je veći od *k*.

Kod svih metoda rezultat je jednak `end()` ako ne postoji nijedan element zbirke koji ispunjava odgovarajući uslov.

```
P& operator[] (const K& k); // map
```

Vrednost ove metode je upućivač na podatak u elementu s ključem *k*. Ako takav element ne postoji zbirci se dodaje element u kome je ključ jednak *k*, a podatak jednak `P()` i rezultat je upućivač na podatak u novododatom elementu. Ova metoda postoji samo za jednoznačno preslikavanje (map).

```
pair<iterator, bool> insert (const E& x); // map
iterator             insert (const E& x); // multimap
```

Prva metoda umeće u jednoznačno preslikavanje (map) novi element kao kopiju parametra *x*, pod uslovom da u zbirci još ne postoji element s ključem *x.first*. Ako takav element već postoji sadržaj zbirke se ne menja. Prvi član vrednosti metode je iterator koji pokazuje na element s ključem *x.first*, bez obzira da li je to novododati element ili element koji je postojao od ranije. Drugi član vrednosti metode je jednak `true` ako je dodavanje novog elementa bilo uspešno.

Druga metoda uvek umeće u višežnačno preslikavanje (multimap) novi element kao kopiju parametra *x*. Vrednost metode je iterator koji pokazuje na novoumetnuti element.

```
template <class It> void insert (It prvi, It posl);
```

Ova metoda umeće u tekuću zbirku (`*this`) kopije svih elemenata tipa *E* druge zbirke iz intervala `[prvi, posl]`.

```
void   erase (iterator poz);
void   erase (iterator prvi, iterator posl);
unsigned erase (const K& k);
```

Prva metoda izbacuje element zbirke na koji pokazuje iterator *poz*.

Druga metoda izbacuje sve elemente zbirke iz intervala `[prvi, posl]`.

Treća metoda izbacuje sve elemente zbirke s ključem *k*. Vrednost metode je broj izbačenih elemenata.

8.2.7 Skupovi (*set* i *multiset*) △

Skupovi u standardnoj biblioteci jezika C++ su zbirke koje omogućuju brzo pronalaženje elemenata na osnovu njihovih vrednosti. Mogu da se smatraju preslikavanjima koja ključevi preslikavaju u same sebe, tj. preslikavanjima čiji elementi sadrže samo ključ, a ne i podatke. Ovako definisani skupovi su, u stvari, uređeni skupovi, što znači da su elementi uređeni prema njihovim vrednostima. Zbog toga spadaju u grupu asocijativnih zbirki.

Slično preslikavanjima iz prethodnog odeljka i skupovi mogu da imaju sve različite elemente, ili mogu da imaju i više elemenata jednakih vrednosti.

Skupovi koriste dvostrane iteratore što omogućuje njihov obilazak po rastućem i po opadajućem poretku vrednosti elemenata.

Skupovi s različitim elementima ostvaruju se generičkom klasom *set*<>, a skupovi koji mogu da imaju i jednake elemente generičkom klasom *multiset*<>. Obe klase su definisane u zaglavlju <set>, a njihove deklaracije su:

```
template <typename K, class U=less<K>> class set;
template <typename K, class U=less<K>> class multiset;
```

K je tip elemenata skupa. Korišćena je oznaka *K* jer se koristi slično kao i ključevi kod zbirki preslikavanja. Tumačenje parametra *U* je isto kao kod preslikavanja u odeljku 8.2.6.

Generičke klase *set*<> i *multiset*<> sadrže sve metode klase *map*<>, odnosno *multimap*<> (videti odeljak 8.2.6). Jedino operator za indeksiranje ([]) ne postoji ni za jednu vrstu skupova. Postoje i sve globalne funkcije za leksikografsko upoređivanje skupova.

8.2.8 Tekstovi (*string*) △

Tekstovi se sastoje od nizova znakova nekog tipa. Za obradu nizova podataka sa semantikom tekstova promenljivih dužina u standardnoj biblioteci jezika C++ postoji generička klasa *basic_string*<>. To omogućava generisanje klase za razne tipove znakova. Na primer, pored uobičajenog tipa *char* koji koristi neki 8-bitni (najčešće ASCII ili UTF-8) kôd, mogu da se koriste i znakovi kodirani 16-bitnim kodom.

Deklaracija generičke klase *basic_string*<>, u zaglavlju <string>, je:

```
template <typename Z> class basic_string;
```

gde je *Z* tip korišćenih znakova.

Posebno se skreće pažnja da ne treba pomešati zaglavljje <string> sa <string.h>. Ovo drugo je zaglavljje koja sadrži deklaracije funkcija za obradu niski u stilu jezika C.

U praksi tekstovi se vrlo često sastoje od znakova tipa *char*. Zbog toga je u zaglavljiju <string> pomoću:

```
typedef basic_string<char> string;
```

Uveden identifikator tipa *string* koji predstavlja klasu generisanu iz generičke klase *basic_string*<> argumentom šablona jednakim *char*. Skreće se pažnja na to da se ovde ne radi o specijalizaciji generičke klase za datu vrednost parametra (što podrazumeva pisanje nove klase), već o generisanju klase na osnovu gotove generičke klase.

8.2.8 Tekstovi (*string*) △

Mada tekstovi ne asociraju na zbirku znakova, generička klasa *basic_string*<> je, ipak, zbirka u smislu definicije iz odeljka 8.2. Poseduje sve mogućnosti generičke klase *vector*<> (videti odeljak 8.2.2) izuzev metoda *push_back()* i *pop_back()*. Pored toga, postoji veliki broj operacija specifičnih za obradu tekstova. Jedino ograničenje u odnosu na vektore je u ograničenom broju tipova podataka koji dolaze u obzir kao tipovi elemenata tekstova.

Od operacija koje postoje kod svih zbirki posebno se skreće pažnja na operator indeksiranja ([]) i na postojanje iteratora s proizvoljnim pristupom. Pomoću njih je moguće pristupati pojedinačnim znakovima u tekstu sadržanom u datom objektu.

U nastavku je dat pregled metoda i globalnih funkcija specifičnih za generičku klasu *basic_string*<>.

Radi lakšeg razumevanja za označavanje tipa za tekstualne podatke u ovoj knjizi koristi se oznaka *string*, a za pojedinačne znakove tip *char*. Tip *char** označava pokazivač na nisku u stilu jezika C (niz znakova sa znakom '\0' na kraju).

Parametar *poz* označava poziciju (redni broj znaka) u tekstu gde treba nešto učiniti. Ako je vrednost tog parametra izvan dozvoljenog opsega (od nule do *dužina*-1), prijavljuje se izuzetak tipa *out_of_range*. Parametar *poz* uvek ima podrazumevanu vrednost 0 sa značenjem „od početka”.

Parametar *max* označava najveći broj znakova koje treba uzeti u obzir. Ako od početka ili počev od pozicije *poz* do kraja teksta nema toliko znakova, koristiće se preostali znakovi do kraja teksta. Parametar *max* uvek ima podrazumevanu vrednost -1 sa značenjem „svi preostali znakovi”. Skreće se pažnja na to da je *max* neoznačenog celobrojnog tipa, tako da vrednost -1, koja u binarnom zapisu sadrži sve bitove jednak jedan, u stvari predstavlja najveću moguću (pozitivnu) vrednost.

IZV označava izvoriste podataka za datu operaciju i može da se sastoji od jednog, dva ili tri parametra. Može da predstavlja jedan objekat (*const string& s*), deo objekta (*const string& s, unsigned poz, unsigned max*), nisku u stilu jezika C (*const char* niz*), početak niske (*const char* niz, unsigned max*) ili dati broj međusobno jednakih znakova (*unsigned n, char z*).

```
string ();
string ( IZV );
```

Prvi konstruktor stvara prazan objekat. Drugi red predstavlja grupu konstruktora koji novi objekat inicijalizuju kopijom navedenog izvorista.

```
string& operator= (T x);
string& assign ( IZV );
```

Operator = levom operandu dodeljuje vrednost kopije parametra *x* (desnog operanda). Tip *T* može da bude *const string&*, *const char** ili *char*. Metode *assign()* tekućem objektu dodeljuju vrednost kopije izvorista. Vrednost svih metoda je tekući objekat s novim sadržajem.

```
string& operator+= (T x);
string& append ( IZV );
```

Operator += dopisuje na kraj levog operanda kopiju parametra *x* (desnog operanda). Tip *T* može da bude *const string&*, *const char** ili *char*. Metode *append()* dopisuju na kraj tekućeg objekta kopije izvorista. Vrednost svih metoda je tekući objekat promjenjenog sadržaja.

```
string operator+ (T1 x, T2 y);
```

Vrednost operatora + je kopija sadržaja desnog operanad (drugog parametra) dopisan na kraj kopije levog operanda (prvog parametra). Jedan od parametara x i y mora da bude tipa `const string&`, a drugi parametar može da bude tipa `const string&` ili `const char*`.

```
istream& operator>> (istream& ut, string& s);
ostream& operator<< (ostream& it, const string& s);
```

Operator >> čita jednu reč (niz znakova između dva bela znaka) iz ulaznog toka ut i smešta u parametar s. Operator << piše sadržaj parametra s u izlazni tok it. Vrednost ovih funkcija je parametar ut, odnosno it u izmenjenom stanju.

```
istream& getline (istream& ut, string& s, char kraj='\'\n');
```

Ova metoda čita tekst do navedenog završnog znaka kraj iz ulaznog toka ut i smešta u parametar s.

```
bool operator rop (T1 x, T2 y);
```

Relacijski operatori rop (==, !=, <, <=, >, >=) vrše leksikografsko upoređivanje svojih operanada. Jedan od parametara x i y mora da bude tipa `const string&`, a drugi parametar može da bude tipa `const string&` ili `const char*`.

```
int compare (const string& s) const;
int compare (const char* niz) const;
int compare (unsigned poz1, unsigned n1,
             const string& s, unsigned poz=0, unsigned n=-1) const;
int compare (unsigned poz1, unsigned n1,
             const char* niz, unsigned n=-1) const;
```

Ove metode leksikografski upoređuju sadržaj tekućeg objekta s tekstrom koji je određen njihovim parametrima.

Prve dve metode upoređuju ceo tekući objekat s drugim objektom, odnosno tekstrom u stilu jezika C.

Druge dve metode upoređuju deo tekućeg objekta, najveće dužine n1 znakova počev od znaka rednog broja poz1, s celim drugim objektom ili delom drugog objekta, odnosno s tekstrom u stilu jezika C. Parametar poz predstavlja početnu poziciju u drugom tekstu, a n najveći broj znakova koje treba uzeti u ozir iz drugog teksta.

Vrednost svih metoda je manja od nule ako je sadržaj tekućeg objekta ispred teksta koji je određen parametrima, veća od nule ako je iza tog teksta i jednaka je nuli ako su uporedivani tekstovi jednakci.

```
string substr (unsigned poz=0, unsigned n=-1) const;
```

Vrednost ove metode je kopija dela tekućeg objekta od najveće n znakova počev od znaka rednog broja poz.

```
unsigned copy (char* niz, unsigned n, unsigned poz=0) const;
```

Ova metoda kopira najveće n znakova počev od znaka rednog broja poz u niz znakova niz bez dodavanja završnog znaka '\0'. Vrednost metode je broj kopiranih znakova.

```
const char* c_str () const;
const char* data () const;
```

Vrednost ove dve metode je pokazivač na dinamičku kopiju sadržaja tekućeg objekta. Metoda `c_str()` dodaje završni znak '\0' na kraj teksta, a metoda `data()` ne. Sadržaj teksta ne sme da se menja. Oslobadanje dodeljene memorije je u nadležnosti objekta čija kopija se nalazi u nizu. Dobijeni pokazivač postaje nevažeći pri prvom pozivanju bilo koje promenljive (ne-`const`) metode za taj objekat.

```
string& insert (unsigned poz1, IZV );
```

Metode iz ove grupe umeću kopiju sadržaja izvorišta IZV ispred znaka rednog broja poz1. Vrednost metoda je tekući objekat izmenjenog sadržaja.

```
string& erase (unsigned poz=0, unsigned n=-1);
```

Ova metoda iz tekućeg objekta izbacuje najviše n znakova počev od pozicije poz. Vrednost metode je tekući objekat izmenjenog sadržaja.

```
string& replace (unsigned poz1, unsigned n1, IZV );
```

```
string& replace (iterator prvi, iterator posl, IZV );
```

Metode iz obe grupe zamenjuju deo sadržaja tekućeg objekta kopijom sadržaja izvorišta IZV. Metode iz prve grupe zamenjuju n1 znakova počev od znaka rednog broja poz1.

Metode iz druge grupe zamenjuju sve znakove iz intervala [prvi, posl]. Vrednost svih metoda je tekući objekat izmenjenog sadržaja.

```
unsigned find (const string& s, unsigned poz=0) const;
```

```
unsigned find (const char* niz,
```

```
           unsigned poz=0, unsigned n=-1 ) const;
```

```
unsigned find (char z,           unsigned poz=0) const;
```

```
unsigned rfind(const string& s, unsigned poz=0) const;
```

```
unsigned rfind(const char* s, unsigned poz=0) const;
```

```
unsigned rfind(const char* niz,
```

```
           unsigned poz=0, unsigned n=-1 ) const;
```

```
unsigned rfind(char z,           unsigned poz=0) const;
```

Vrednost ovih metoda je redni broj mesta prvog (`find()`), odnosno poslednjeg (`rfind()`) pojavljivanja prvog znaka teksta koji je određen parametrima u tekućem objektu počev od mesta poz. Traženi tekst može da bude sadržaj objekta s, najviše n znakova (podrazumevano do znaka '\0') iz teksta niz u stilu jezika C, ili jedan znak z. Vrednost svih metoda u slučaju neuspeha je -1 (tačnije, najveći mogući pozitivan broj).

```
unsigned find_first_of ( ... ) const;
```

```
unsigned find_last_of ( ... ) const;
```

```
unsigned find_first_not_of ( ... ) const;
```

```
unsigned find_last_not_of ( ... ) const;
```

Vrednost metoda u prvoj grupi metoda je redni broj mesta prvog pojavljivanja u tekućem objektu bilo kog znaka koji se nalazi u tekstu koji je određen parametrima.

Vrednost metoda u drugoj grupi metoda je redni broj mesta poslednjeg pojavljivanja u tekućem objektu bilo kog znaka koji se nalazi u tekstu koji je određen parametrima.

Vrednost metoda u trećoj grupi metoda je redni broj mesta prvog pojavljivanja u tekućem objektu bilo kog znaka koji se nalazi u tekstu koji je određen parametrima.

Vrednost metoda u trećoj grupi metoda je redni broj mesta poslednjeg pojavljivanja u tekućem objektu bilo kog znaka koji se ne nalazi u tekstu koji je određen parametrima.

Mogući parametri su isti kao i kod metoda `find()`, odnosno `rfind()`. Vrednost svih metoda u slučaju neuspeha je `-1`.

8.2.9 Nizovi bitova (`vector<bool>::bitset`)

Za rad s nizovima bitova standardna biblioteka jezika C++ nudi dva rešenja.

Prvo rešenje je specijalizacija generičke klase `vector<>` za slučaj elemenata tipa `bool`. Primeri specijalizovane generičke klase `vector<bool>` su dinamički nizovi promenljivih dužina s efikasnim uskladištanjem bitova s obzirom na utrošak memorije. Drugačije rečeno, bitovi se pakuju, više njih u jednu memoriju lokaciju. Pakovanje bitova je nevidljivo za korisnika klase. Pojedinim bitovima može da se pristupa na isti način (indeksiranjem ili iteratorima) kao i u slučaju vektora podataka drugačijih tipova. Na primer:

```
vector<bool> v(20);
for (unsigned i=0; i<v.size(); cin>>v[i++]);
for (vector<bool>::iterator i=v.begin(); i!=v.end(); cout<<*i++);
```

Druge rešenje je generička klasa `bitset<>` koja ostvaruje nizove bitova fiksnih dužina na način kako to rade operatori za rad s bitovima (`~, &, |, ^, <<, >>`) s operandima celobrojnih tipova. Pošto je primena tih operatora na celobrojne tipove podataka znatno efikasnija od primene na primerke klase `bitset<>`, tu klasu treba koristiti samo za nizove bitova koji ne mogu da stanu ni u jedan od celobrojnih tipova podataka.

Generička klasa `bitset<>` mada ima neke osobine zbirk i to nije u smislu definicije zbirk u odeljku 8.2. Posebno se ističe da ne podržava upotrebu iteratora.

Generička klasa `bitset<>` je definisana u zaglavlju `<bitset>` i njena deklaracija je:

```
template <unsigned N> class bitset;
```

gde `N` predstavlja broj bitova u primercima klase `bitset<N>`.

Kao što je uobičajeno kod celobrojnih tipova podataka, bitovi unutar nizova bitova se numerišu zdesna ulevo rednim brojevima počev od nule. Svi operatori po bitovima `~, &, |, ^, <<, >>` deluju na nizove bitova na isti način kao na celobrojne tipove. Pomeranja ulevo i udesno su logička pomeranja, tj. upražnjeni bitovi se popunjavaju nulama. Operatori `==` i `!=` ispituju da li su dva niza bitova jednak ili različita.

Postoji podrazumevani konstruktor koji sve bitove postavlja na nulu, konstruktor kopije i operator za dodelu vrednosti. Operatori `>>` i `<<` su preklopjeni i za potrebe čitanja i pisanja nizova bitova.

U nastavku su navedene preostale metode klase `bitset<>`. Sve metode koje imaju parametar `poz` (redni broj bita) prijavljuju izuzetak tipa `out_of_range` ako je `poz` izvan dozvoljenog opsega (od 0 do `N-1`).

```
bitset <unsigned long x>;
```

Ovaj konstruktor novi objekat inicijalizuje sadržajem desnih `N` bitova parametra `x`. Ako u parametru `x` nema `N` bitova dopuniće se nulama s leve strane.

8.2.9 Nizovi bitova (`vector<bool>::bitset`)

```
explicit bitset (const string& s,
                 unsigned prvi=0, unsigned n=-1);
```

Ovaj konstruktor novi objekat inicijalizuje na osnovu najviše `n` znakova počev od pozicije `prvi` ulevo u tekstu `s` (za ovu primenu, izuzetno, poslednji znak teksta ima redni broj 0). Ako od pozicije `prvi` do početka teksta nema `n` znakova koristiće se znakovи do početka teksta. Podrazumevana vrednost `-1` za `n` označava „do početka teksta”. Ako je `prvi>s.size()`, prijavljuje se izuzetak tipa `out_of_range`. Ako u korišćenom delu teksta `s` postoji bar jedan znak koji nije jednak '0' ili '1', prijavljuje se izuzetak tipa `invalid_argument`. Izdvojeni bitovi se oslanjamaju uz desnu ivicu stvorenog niza bitova. Eventualni višak bitova se odseca na levom kraju. U slučaju manjka bitova niz se dopunjaje nulama na levoj strani.

```
bitset<N>& set ();
bitset<N>& set (unsigned poz, unsigned b=1);
bitset<N>& reset ();
bitset<N>& reset (unsigned poz);
bitset<N>& flip ();
bitset<N>& flip (unsigned poz);
```

Prva metoda postavlja vrednost svih bitova na 1. Druga metoda postavlja vrednost bita rednog broja `poz` na `b`.

Treća metoda postavlja vrednost svih bitova na 0. Treća metoda postavlja vrednost bita rednog broja `poz` na 0.

Peta metoda komplementira vrednosti svih bitova. Šesta metoda komplementira vrednost bita rednog broja `poz`.

Vrednost svih metoda je upućivač na tekući objekat, što omogućava lančanje operacija u jednom izrazu.

```
reference operator[] (unsigned poz);
```

Ova metoda omogućava pristup bitu rednog broja `poz`. Vrednost metode je specijalan tip `reference` koji omogućava kako dohvatanje, tako i menjanje vrednosti odabranog bita unutar niza bitova. Drugim rečima, sme da se napiše `niz[i]=1`, bez obzira na činjenicu da odabrani bit nije samostalni podatak.

```
unsigned long to_ulong () const;
```

Vrednost ove metode je celobrojan podatak koji se dobija od bitova tekućeg niza bitova. Ako dobijena celobrojna vrednost ne može da stane u podatak tipa `unsigned long`, prijavljuje se izuzetak tipa `overflow_error`.

```
unsigned count () const;
unsigned size () const;
```

Vrednost prve metode je broj bitova jednakih 1 u nizu bitova. Vrednost druge metode je ukupan broj bitova u nizu bitova (tj. `N`).

```
bool test (unsigned poz) const;
bool any () const;
bool none () const;
```

Vrednost prve metode je `true` ako je bit rednog broja `poz` jednak 1. Vrednost druge metode je `true` ako je bar jedan bit jednak 1. Vrednost treće metode je `true` ako nijedan bit nije jednak 1.

8.3 Algoritmi

Standardna biblioteka sadrži veći broj generičkih funkcija za često korišćene postupke nad zbirkama podataka. Njihove deklaracije se nalaze u zaglavlju `<algorithm>`.

Većina algoritama deluje na elemente neke zbirke koji su određeni intervalom `[prvi, posl]` vrednosti iteratora `prvi` i `posl`. U obzir se uzimaju svi elementi zbirke počev od elementa na koji pokazuje iterator `prvi` do ispred elementa na koji pokazuje iterator `posl`. Da bi se obradivali svi elementi zbirke `z`, za granice intervala treba navesti `z.begin()` i `z.end()`.

Za neke obrade postoje po dve generičke funkcije. Jedna od njih poseduje parametar koji je funkcionska klasa kojom se određuje uslov pod kojim se biraju ili uređuju elementi zbirke. Druga od njih ne poseduje taj parametar već primenjuje unapred definisan uslov za odabiranje ili upoređivanje. Bez obzira što šabloni generičkih funkcija ne mogu da imaju parametre s podrazumevanim vrednostima, radi sažetijeg izražavanja u ovom odeljku podrazumevani uslovi su navedeni kao podrazumevane vrednosti parametara šablonu. Kao podrazumevane vrednosti navedene su odgovarajuće bibliotečke funkcionske klase iz odeljka 8.1.3.

8.3.1 Algoritmi nad pojedinačnim podacima

U ovom odeljku su prikazane generičke funkcije koje izvode elementarne operacije nad pojedinačnim podacima (koji nisu elementi neke zbirke).

```
template <typename T, class U=less<T> >
    const T& min (const T& x, const T& y, U u);
template <typename T, class U=less<T> >
    const T& max (const T& x, const T& y, U u);
```

Ove generičke funkcije odabiraju parametar manje, odnosno veće vrednosti.

Vrednost prve generičke funkcije je upućivač na parametar `y` ako je vrednost izraza `u(y, x)` jednaka `true`, a inače upućivač na parametar `x` (podrazumevani uslov je `y < x`).

Vrednost druge generičke funkcije je upućivač na parametar `y` ako je vrednost izraza `u(x, y)` jednaka `true`, a inače upućivač na parametar `x` (podrazumevani uslov je `x < y`).

```
template <typename T> void swap (T& x, T& y);
```

Ova generička funkcija međusobno zamenjuje vrednosti parametara `x` i `y`.

8.3.2 Obrada pojedinačnih elemenata zbirki

U ovom odeljku su prikazane generičke funkcije koje određenu operaciju primenjuju na svaki element zbirke, nezavisno od ostalih elemenata.

```
template <class It, class Op>
    Op for_each (It prvi, It posl, Op f);
```

Ova generička funkcija primenjuje funkcionski objekat `f` na uzastopne elemente zbirke, koja je određena iteratorima, obavljajući određenu radnju zavisno od vrednosti tih elemenata. Vrednosti elemenata ne smeju da se promene, već rezultati obrade treba da se odlazu u objektu `f`.

Za svaku vrednost iteratora `i` tipa `It` u intervalu `[prvi, posl]` izvršava se naredba `f(*i);`. Bočni efekti te naredbe mogu da utiču na sadržaj objekta `f`. Eventualna vrednost izraza `f(*i)` se ne koristi.

Vrednost generičke funkcije `for_each()` je funkcionski objekat `f` koji može da sadrži rezultate obrade elemenata zbirke.

```
template <class It, typename E>
    void fill (It prvi, It posl, const E& x);
template <class It, typename Vel, typename E>
    void fill_n (It prvi, Vel n, const E& x);
template <class It, class G>
    void generate (It prvi, It posl, G g);
template <class It, typename Vel, class G>
    void generate_n (It prvi, Vel n, G g);
```

Ove generičke funkcije popunjavaju elemente zbirke nekom vrednošću.

Prva i treća generička funkcija popunjavaju elemente zbirke u intervalu `[prvi, posl]`. Druga i četvrta generička funkcija popunjavaju `n` elemenata zbirke počev od elementa na koji pokazuje iterator `prvi`. Parametar šablonu `Vel` treba da predstavlja neki celobrojan tip.

Prve dve metode odabrane elemente popunjavaju kopijama parametra `x`.

Druge dve metode elemente popunjavaju vrednostima izraza `g()` koji se iznova izračunava za svaki element. Klasa `G` je funkcionska klasa koja sadrži operatorsku funkciju za operator `()` čiji je prototip `E operator () ()`, gde je `E` tip elemenata zbirke. Skreće se pažnja na to da vrednost izraza `g()` ne zavisi od sadržaja elementa zbirke na koji se primenjuje.

```
template <class It1, class ItR, class Op1>
    ItR transform (It1 prvi1, It1 posl1, ItR rez, Op1 f1);
template <class It1, class It2, class ItR, class Op2>
    ItR transform (It1 prvi1, It1 posl1,
                    It2 prvi2, ItR rez, Op2 f2);
```

Prva generička funkcija primenjuje funkcionski objekat `f1` na uzastopne elemente izvořišne zbirke stavljući rezultate u uzastopne elemente određene zbirke. Slično tome, druga generička funkcija primenjuje funkcionski objekat `f2` na uzastopne parove elemenata iz dve izvořišne zbirke stavljući rezultate u uzastopne elemente određene zbirke.

Za svaku vrednost iteratora `i1` tipa `It1` u intervalu `[prvi1, posl1]` prva funkcija izvršava naredbu `*ir=f1(*i1);`, a druga naredbu `*ir=f2(*i1, *i2);`. Iterator `i2` poprima uzastopne vrednosti počev od `prvi2`, a iterator `ir` poprima uzastopne vrednosti počev od `rez`. Vrednosti elemenata izvořišnih zbirki `(*i1 i *i2)` ne smeju da se promene. Intervali elemenata izvořišnih zbirki i određene zbirke mogu da se preklapaju.

Vrednost obe generičke funkcije je iterator koji pokazuje iza poslednje obradenog elementa određene zbirke (tj. završna vrednost iteratora `ir`).

Skreće se pažnja na to da ove funkcije ne stvaraju nove elemente u određenoj zbirci, već samo menjaju vrednosti postojećih elemenata.

Program 8.1 prikazuje primer korišćenja nekih od prethodnih funkcija kojim se izračunava zbir kvadrata niza celih brojeva.

```

#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

class Pravi {
    // Generisanje uzastopnih celih brojeva.
public:
    Pravi (int p=1, int d=1): br(p-d), kor(d) {}
    int operator() () { return br += kor; }

    struct Kvadrat { // Izračunavanje kvadrata celog broja.
        int operator() (const int& k) { return k * k; }
    };

    struct Pisi {
        // Ispisivanje celog broja.
        void operator() (const int& k) { cout << k << ' '; }
    };

    class Zbir { // Kumulativno sabiranje celih brojeva.
public:
    Zbir () : rez() {}
    void operator() (const int& k) { rez += k; }
    int operator+ () const { return rez; }
};

int main () {
    cout << "Broj sabiraka? "; int n; cin >> n;
    cout << "Početni broj? "; int p; cin >> p;
    cout << "Korak? "; int d; cin >> d;
    vector<int> v (n);
    vector<int>::iterator prvi=v.begin(), posl=v.end();
    generate (prvi, posl, Pravi(p,d));
    cout << "Brojevi= "; for_each (prvi, posl, Pisi()); cout << endl;
    transform (prvi, posl, prvi, Kvadrat());
    cout << "Kvadrati= "; for_each (prvi, posl, Pisi()); cout << endl;
    cout << "Zbir kvadrata= "
        << + for_each (prvi, posl, Zbir()) << endl;
}

```

Program 8.1 – Izračunavanje zbiru kvadrata celih brojeva (zbirkvad.c)

Na početku programa nalaze se četiri jednostavne funkcionske klase čiji primerci će se primjenjivati na uzastopne elemente niza brojeva.

Funkcionska klasa `Pravi` služi za generisanje uzastopnih celih brojeva od početne vrednosti `p` s korakom `d` (`p` i `d` su parametri konstruktora). Privatno polje `br` predstavlja vrednost poslednjeg generisanog broja. Pri svakom pozivanju operatorske funkcije za operator `()` vrednost polja `br` se povećava za iznos koraka `kor`.

Funkcionska klasa `Kvadrat` izračunava kvadrat parametra operatorske funkcije za operator `()`. Deklarisana je kao struktura (`struct`) da bi njena jedina metoda podrazumevano bila javna. Slično ovoj klasi, funkcionska klasa `Pisi` ispisuje parametar svoje operatorske funkcije za operator `()`.

8.3.2 Obrada pojedinačnih elemenata zbirki

Funkcionska klasa `Zbir` služi za kumulativno izračunavanje zbiru celih brojeva. Privatno polje `rez` se podrazumevanim konstruktorom postavlja na nulu. Posle toga pri svakom pozivanju operatorske funkcije za operator `()` vrednost parametra te metode se dodaje na vrednost polja `rez`. Akumulirani rezultat može da se dohvati operatorskom funkcijom za unarni operator `+`.

U glavnoj funkciji, posle čitanja potrebnih podataka, stvara se vektor celih brojeva `v` potrebne veličine i definišu se iteratori `prvi` i `posl` tako da pokazuju na početak i na kraj vektora `v`. Tačnije, `prvi` pokazuje na prvi element, a `posl` iza poslednjeg elementa.

Za generisanje niza uzastopnih celih brojeva od broja `p` s korakom `d` koristi se generička funkcija `generate<>()`. Kao funkcionski objekat naveden je bezimeni primerak klase `Pravi` stvoren pomoću konstruktora te klase. Generička funkcija `generate<>()`, poslovanjem operatorske funkcije `operator()()` tog funkcionskog objekta za svaki element vektora `v`, popunjava vektor sa uzastopno obrazovanim celim brojevima. Interval obradivanih elemenata vektora `v` je određen vrednostima iteratora `prvi` i `posl`.

Slično tome, generisani niz se ispisuje pozivanjem generičke funkcije `for_each<>()` primerkom funkcionske klase `Pisi` kao funkcionskim argumentom.

Računanje kvadrata elemenata vektora `v`, stavljajući rezultate u isti vektor `v`, postignuto je pozivanjem generičke funkcije `transform<>()` time što je kao početak odredišne zbirke (treći argument funkcije) naveden iterator `prvi`, koji označava i početak izvorišne zbirke (prvi argument funkcije).

Na kraju, posle ispisivanja sadržaja promjenjenog vektora `v` na već opisan način, zbir elemenata vektora `v` računa se pozivanjem generičke funkcije `for_each<>()`. Sada je funkcionski objekat bezimeni primerak funkcionske klase `Zbir`. Vrednost funkcije `for_each<>()` je taj funkcionski objekat u izmenjenom stanju. Zbir elemenata vektora `v` dobija se primenom unarnog operatara `+` klase `Zbir` na vrednost funkcije `for_each<>()`.

8.3.3 Pretraživanje zbirki

U ovom odeljku prikazane su generičke funkcije kojima se ispituje da li se u zbirci nalaze elementi određenih osobina.

```

template <class It, typename E>
It find (It prvi, It posl, const E& x);
template <class It, class U>
It find_if (It prvi, It posl, U u);

```

Ove generičke funkcije pronalaze prvi element zbirke koji zadovoljava zadati uslov.

Vrednost prve funkcije je najmanja vrednost iteratora `i` tipa `It` u intervalu `[prvi, posl]` za koji je `*i==x`. Vrednost druge funkcije je najmanja vrednost iteratora `i` tipa `It` u intervalu `[prvi, posl]` za koji je vrednost izraza `u(*i)` jednaka `true`. Vrednost obe funkcije je vrednost iteratora `posl` ako ne postoji nijedan element koji zadovoljava dati uslov.

```
template <class It1, class It2, class U=equal_to<E> >
    It find_first_of (It1 prvi1, It1 pos1,
                      It2 prvi2, It2 pos12, U u);
```

Ova generička funkcija pronalazi prvi element prve zbirke koji je u zadatoj relaciji s bilo kojim elementom druge zbirke.

Vrednost funkcije je iterator *i1* tipa *It1* u intervalu *[prvi1, pos1]* koji pokazuje na prvi element zbirke za koji postoji element druge zbirke takav da je vrednost izraza *u(*i1, *i2)* jednaka **true**, gde je *i2* iterator tipa *It2* u intervalu *[prvi2, pos12]* koji pokazuje na elemente druge zbirke (podrazumevani uslov je **i1==*i2*). Vrednost funkcije je vrednost iteratora *pos1* ako ne postoji nijedan element koji zadovoljava dati uslov.

```
template <class It, class U=less<E> >
    It min_element (It prvi, It pos1, U u);
template <class It, class U=less<E> >
    It max_element (It prvi, It pos1, U u);
```

Ove generičke funkcije pronalaze prvo pojavljivanje najmanjeg, odnosno najvećeg elementa zbirke.

Vrednost prve funkcije je najmanja vrednost iteratora *i* tipa *It* u intervalu *[prvi, pos1]* za koju ne postoji nijedan element *e* u navedenom intervalu za koju je vrednost izraza *u(e, *i)* jednaka **true** (podrazumevani uslov je *e<*i*).

Vrednost druge funkcije je najmanja vrednost iteratora *i* tipa *It* u intervalu *[prvi, pos1]* za koju ne postoji nijedan element *e* u navedenom intervalu za koju je vrednost izraza *u(*i, e)* jednaka **true** (podrazumevani uslov je **i<e*).

```
template <class It, typename E>
    unsigned count (It prvi, It pos1, const E& x);
template <class It, class U>
    unsigned count_if (It prvi, It pos1, U u);
```

Ove generičke funkcije određuju broj elemenata zbirke koji zadovoljavaju zadati uslov.

Vrednost prve funkcije je broj elemenata tipa *E* u intervalu *[prvi, pos1]* čije su vrednosti jednake *x*. Vrednost druge funkcije je broj elemenata *e* tipa *E* u intervalu *[prvi, pos1]* za koje je vrednost izraza *u(e)* jednaka **true**.

```
template <class It1, class It2, class U=equal_to<E> >
    It1 search (It1 prvi1, It1 pos11, It2 prvi2, It2 pos12, U u);
template <class It1, class It2, class U=equal_to<E> >
    It1 find_end(It1 prvi1, It1 pos11, It2 prvi2, It2 pos12, U u);
```

Ove generičke funkcije pronalaze prvo, odnosno poslednje pojavljivanje niza elemenata druge zbirke kao podniz niza elemenata prve zbirke pod zadatim uslovom.

Vrednosti funkcija su najmanja, odnosno najveća vrednost iteratora *i1* tipa *It1* u intervalu *[prvi1, pos11]* takvo da za svaki par elemenata *(el1, el2)*, gde je *el2* element druge zbirke počev od mesta *prvi2* do ispred mesta *pos12* i *el1* odgovarajući element prve zbirke počev od mesta *i1*, vrednost izraza *u(el1, el2)* je jednaka **true** (podrazumevani uslov je *el1==el2*). Vrednost obe funkcije je vrednost iteratora *pos11* ako ne postoji traženi podniz.

8.3.4 Obrada zbirki

U ovom odeljku prikazane su generičke funkcije koje vrše obradu zbirki u celini.

Neke od tih funkcija rezultat stavljuju u neku odredišnu zbirku. U tim slučajevima neophodno je pre pozivanja funkcije obezbediti da odredišna zbirka ima dovoljan broj elemenata za prihvatanje rezultata. Same funkcije ne menjaju veličinu odredišne zbirke, već samo menjaju sadržaje postojećih elemenata.

```
template <class It, class ItR>
    ItR copy (It prvi, It pos1, ItR rez);
template <class It, class ItR>
    ItR copy_backward (It prvi, It pos1, ItR rez);
```

Ove generičke funkcije kopiraju sve elemente izvorišne zbirke u intervalu *[prvi, pos1]* u odredišnu zbirku.

Prva generička funkcija elemente kopira po redosledu elemenata u izvorišnoj zbirci. Odredišnu zbirku popunjava od elementa na koji pokazuje iterator *rez*. Vrednost funkcije je iterator koji pokazuje iza poslednjeg elementa rezultata.

Druga generička funkcija elemente kopira po obrnutom redosledu u odnosu na redosled elemenata u izvorišnoj zbirci. Odredišna zbirka se popunjava od elementa iza kojeg pokazuje iterator *rez* prema početku zbirke. Vrednost funkcije je iterator koji pokazuje na prvi element rezultata.

```
template <class It, typename E>
    void replace (It prvi, It pos1, const E& x, const E& novo);
template <class It, class U, typename E>
    void replace_if (It prvi, It pos1, U u, const E& novo);
template <class It, class ItR, typename E>
    ItR replace_copy (It prvi, It pos1,
                      ItR rez, const E& x, const E& novo);
template <class It, class ItR, class U, typename E>
    ItR replace_copy_if (It prvi, It pos1,
                         ItR rez, U u, const E& novo);
```

Ove generičke funkcije zamjenjuju vrednosti elemenata zbirke u intervalu *[prvi, pos1]* koji zadovoljavaju dati uslov kopijom parametra *novo*.

Prve dve generičke funkcije menjaju vrednosti elemenata zbirke koju obrađuju.

Druge dve generičke funkcije kopiraju elemente izvorišne zbirke u odredišnu zbirku uz zamenu vrednosti elemenata koji zadovoljavaju uslov za zamenu. Odredišna zbirka se popunjava počev od elementa na koji pokazuje iterator *rez*. Vrednost obe funkcije je iterator koji pokazuje iza poslednjeg elementa rezultata.

Prva i treća generička funkcija zamjenjuju vrednosti elemenata koji su jednakim vrednostima parametra *x*. Druga i četvrta metoda zamjenjuju vrednosti elemenata *z* izvorišne zbirke za koje je vrednost izraza *u(z)* jednaka **true**.

```

template <class It, typename E>
    It remove (It prvi, It posl, const E& x);
template <class It, class U>
    It remove_if (It prvi, It posl, U u);
template <class It, class ItR, typename E>
    ItR remove_copy (It prvi, It posl, ItR rez, const E& x);
template <class It, class ItR, class U>
    ItR remove_copy_if (It prvi, It posl, ItR rez, U u);
Ove generičke funkcije izbacuju elemente zbirke u intervalu [prvi, posl) koji zadovoljavaju dati uslov.

```

Prve dve generičke funkcije izbacuju elemente zbirke koju obradjuju. Vrednost obe funkcije je iterator koji pokazuje iza poslednjeg elementa koji je ostao u zbirci.

Druge dve generičke funkcije kopiraju elemente izvorišne zbirke u odredišnu zbirku uz preskakanje elemenata koji zadovoljavaju uslov za izbacivanje. Izvorišna zbirka ostaje neizmenjena. Odredišna zbirka se popunjava počev od elementa na koji pokazuje iterator rez. Vrednost obe funkcije je iterator koji pokazuje iza poslednjeg elementa rezultata.

Prva i treća generička funkcija izostavljaju elemente koji su jednaki vrednosti parametra x. Druga i četvrta metoda izostavljaju elemente e izvorišne zbirke za koje je vrednost izraza $u(e)$ jednaka true.

```

template <class It>
void reverse (It prvi, It posl);
template <class It, class ItR>
ItR reverse_copy (It prvi, It posl, ItR rez);

```

Ove generičke funkcije obrću redosled elemenat zbirke u intervalu [prvi, posl).

Prva generička funkcija menja redosled elemenata zbirke koju obraduje.

Druga generička funkcija kopira elemente izvorišne zbirke u odredišnu zbirku uz izmenu redosleda elemenata. Odredišna zbirka se popunjava počev od elementa na koji pokazuje iterator rez. Vrednost funkcije je iterator koji pokazuje iza poslednjeg elemenata rezultata.

```

template <class It, class U=equal_to<E>>
It unique (It prvi, It posl, U u);
template <class It, class ItR, class U=equal_to<E>>
ItR unique_copy (It prvi, It posl, ItR rez, U u);

```

Ove generičke funkcije redukuju zbirku u intervalu [prvi, posl) tako da u svakoj sekvenци uzastopno jednakih elemenata zadrže samo prvi element i izbacuju preostale.

Elementi e1 i e2 se smatraju jednakim ako je vrednost izraza $u(e1, e2)$ jednaka true (podrazumevani uslov je $e1==e2$).

Prva generička funkcija izbacuje elemente zbirke koju obradjuje. Vrednost funkcije je iterator koji pokazuje iza poslednjeg elementa koji je ostao u zbirci.

Druga generička funkcija kopira elemente izvorišne zbirke u odredišnu zbirku uz preskakanje elemenata koji zadovoljavaju uslov za izbacivanje. Izvorišna zbirka ostaje neizmenjena. Odredišna zbirka se popunjava počev od elementa na koji pokazuje iterator rez. Vrednost funkcije je iterator koji pokazuje iza poslednjeg elementa rezultata.

8.3.4 Obrada zbirki △

```

template <class It1, class It2, class U=equal_to<E>>
bool equal (It1 prvi1, It1 posl1, It2 prvi2, U u);

```

Ova generička funkcija ispituje da li je niz elemenata prve zbirke jednak nizu elemenata druge zbirke.

Vrednost funkcije je jednaka true ako za svaki par elemenata (e1, e2), gde je e1 element prve zbirke počev od mesta prvi1 do mesta posl1 i e2 odgovarajući element druge zbirke počev od mesta prvi2, vrednost izraza u(e1, e2) je jednaka true (podrazumevani uslov je e1==e2).

```

template <class It1, class It2, class U=less<E>>
bool lexicographical_compare (It1 prvi1, It1 posl1,
                             It2 prvi2, It2 posl2, U u);

```

Ova generička funkcija ispituje da li je niz elemenata prve zbirke leksikografski ispred niza elemenata druge zbirke. Prvi niz je ispred drugog niza ako kod prvog para različitih elemenata element u prvom nizu je ispred elementa u drugom nizu. Ako je prvi niz kraći od drugog i svi njegovi elementi su jednaki odgovarajućim elementima drugog niza, prvi niz se smatra manjim.

Vrednost funkcije je jednaka true ako za prvi par različitih elemenata (e1, e2), gde je e1 element prve zbirke počev od mesta prvi1 do mesta posl1 i e2 odgovarajući element druge zbirke počev od mesta prvi2 do mesta posl2, vrednost izraza u(e1, e2) je jednaka true (podrazumevani uslov je e1<e2). Dva elementa e1 i e2 se smatraju jednakim ako je vrednost izraza !u(e1, e2) &!u(e2, e1) jednaka true (neposredna relacija jednakosti ne mora da bude definisana).

```

template <class It, class U=less<E>>
bool next_permutation (It prvi, It posl, U u);
template <class It, class U=less<E>>
bool prev_permutation (It prvi, It posl, U u);

```

Ove generičke funkcije menjaju redosled elemenata zbirke u intervalu [prvi, posl) tako da predstavljaju prvu sledeću, odnosno prethodnu permutaciju u odnosu na početni redosled prema leksikografskom poretku (videti i generičku funkciju lexicographical_compare()).

Vrednost obe generičke funkcije je jednaka true ako takva permutacija postoji, odnosno false ako takva permutacija ne postoji. U ovom drugom slučaju sadržaj zbirke kod prve funkcije biće permutacija koja je ispred, a kod druge funkcije koja je iza svih permutacija. U slučaju podrazumevanog upoređivanja operatorom < elementi zbirke u permutaciji koja je ispred svih permutacija poredani su po neopadajućem redosledu. U permutaciji koja je iza svih permutacija elementi su poredani po nerastućem redosledu.

8.3.5 Obrada uređenih sekvenčijalnih zbirki △

Mada sekvenčijalne zbirke ne podrazumevaju uređenost elemenata po nekom kriterijumu, u praksi je često potrebno da se elementi takvih zbirki uređuju i da se vrše neke operacije nad takvim zbirkama. U ovom odeljku prikazane su važnije generičke funkcije za rad s uređenim sekvenčijalnim zbirkama.

Za upoređivanje elemenata koristi se funkcionska klasa *U* koja treba da određuje kada je jedan element ispred drugog elementa.

Smatra se da je element *e1* ispred elementa *e2* ako je vrednost izraza *u(e1, e2)* jednaka **true**, gde je *u* funkcionski objekat tipa *U*. Podrazumevani uslov je *e1 < e2*, što daje uređivanje po neopadajućem poretku (medusobno jednakim elementima su dozvoljeni).

Jednakost elemenata se izvodi iz prethodne relacije. Dva elementa *e1* i *e2* smatraju se jednakim ako su vrednosti oba izraza *u(e1, e2)* i *u(e2, e1)* jednake **false**. Za podrazumevanu relaciju *<*, to znači da *e1* nije manji od *e2* i *e2* nije manji od *e1*.

```
template <class It, class U=less<E> >
void sort (It prvi, It posl, U u);

template <class It, class U=less<E> >
void stable_sort (It prvi, It posl, U u);
```

Ove generičke funkcije uređuju elemente zbirke u intervalu *[prvi, posl]*. Moguće ih je primeniti samo na vektore (*vector<>*) i redove s dva kraja (*deque<>*).

Prva funkcija koristi vrlo efikasnu metodu, dok druga funkcija manje efikasnu metodu ali koja čuva relativan redosled elemenata koji se smatraju jednakim.

```
template <class It1, class It2, class ItR, class U=less<E> >
ItR merge (It1 prvil, It1 pos1,
           It2 prvi2, It2 pos12, ItR rez, U u);
```

Ova generička funkcija na osnovu dve uređene zbirke odredene intervalima *[prvil, pos1]* i *[prvi2, pos12]* obrazuje treću uređenu zbirku.

Rezultat se smešta u određenu zbirku počev od mesta koje je određeno iteratorom *rez*. Vrednost funkcije je iterator *ir* takav da se elementi rezultata nalaze u intervalu *[rez, ir]*. Skreće se pažnja na to da određena zbirka pre početka operacije treba da ima dovoljan broj elemenata za smeštanje rezultata. Naime, ne dodaju se novi elementi određenoj zbirci, već se samo menjaju vrednosti postojećih elemenata.

```
template <class It, typename E, class U=less<E> >
bool binary_search (It prvi, It posl, const E& x, U u);
```

Ova generička funkcija ispituje da li uređena zbirka u intervalu *[prvi, posl]* sadrži element jednak parametru *x*. Moguće je primeniti samo na vektore (*vector<>*) i redove s dva kraja (*deque<>*).

```
template <class It, typename E, class U=less<E> >
It lower_bound (It prvi, It posl, const E& x, U u);

template <class It, typename E, class U=less<E> >
It upper_bound (It prvi, It posl, const E& x, U u);

template <class It, typename E, class U=less<E> >
pair<It, It> equal_range (It prvi, It posl, const E& x, U u);
```

Ove generičke funkcije pronalaze granice opsega u uređenoj zbirci određenog vrednošću parametra *x*. Zbirka se pretražuje u intervalu *[prvi, posl]*.

Vrednost prve generičke funkcije je iterator koji pokazuje na prvi element zbirke koji nije ispred parametra *x*.

Vrednost druge generičke funkcije je iterator koji pokazuje na prvi element zbirke koji je iza parametra *x*.

Vrednost treće generičke funkcije je par iteratora *p* koji određuju interval *[p.first, p.second]* elemenata zbirke koji su jednakim parametru *x*.

8.4 Zadaci △

8.4.1 Obrada obojenih geometrijskih figura u ravni △

Zadatak:

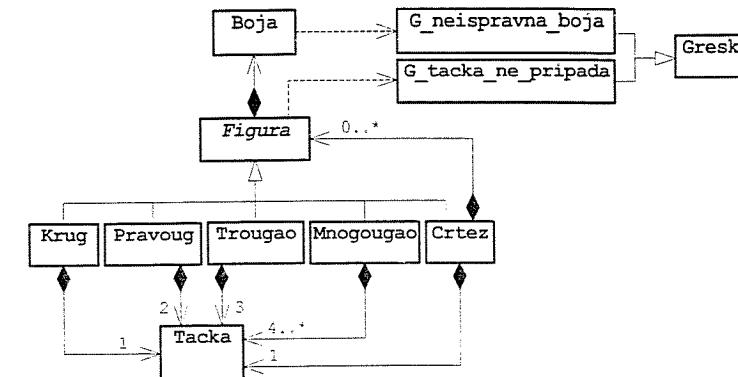
Napisati na jeziku C++ sistem klase za obradu obojenih geometrijskih figura. Predviđeti jednostavne figure (kao što su krugovi, pravougaonici, trouglovi, mnogouglovi) i složene figure koje mogu da sadrže druge figure. Jednostavne figure imaju svoju boju, a složene figure osnovnu boju, dok boju date tačke određuje komponentna figura na tom mestu.

Napisati na jeziku C++ program za prikazivanje mogućnosti projektovanih klasa.

Rešenje:

Na slici prikazan je dijagram klase za rešenje ovog zadatka. Sve te klase se nalaze u prostoru imena *Figure*.

Na centralnom mestu se nalazi apstraktna klasa *Figura* koja obuhvata zajedničke osobine svih klasa za geometrijske figure. To je pre svega boja figure predstavljena klasom *Boja*.



Slika 8.1 – Klase za obojene geometrijske figure

Od jednostavnih figura postoje klase, izvedene iz klase *Figura*, za krugove, pravougaonike, trouglove i mnogouglove. Za određivanje njihovih položaja u ravni koristi se jedan ili više primeraka klase *Tacka*.

Jedina klasa za složene figure je klasa *Crtez*. Ona, pored toga što je izvedena iz klase *Figura*, u sebi može da sadrži i proizvoljan broj drugih figura. Među tim figurama mogu da budu drugi crteži. To omogućava sastavljanje proizvoljno složenih figura. Za označavanje položaja crteža u ravni, kao celine, koristi se jedan objekat tipa *Tacka*.

Za prijavljivanje grešaka izuzećima predviđene su dve klase izvedene iz klase *Greska*, koja je prikazana u rešenju zadatka 6.6.2 kao program 6.4. Klasa *Greska* nalazi se u prostoru imena *Usluge*.

Program 8.2 prikazuje definiciju klase Tacka. Pored stvaranja objekata omogućava dohvatanje koordinata, izračunavanje rastojanja između dve tačke i čitanje i pisanje podataka o tačkama.

```
// Definicija klase tacaka (Tacka).

#ifndef _tacka4_h_
#define _tacka4_h_

#include <iostream>
#include <cmath>
using namespace std;

namespace Figure {
    class Tacka {
        double x, y;
    public:
        Tacka () { x = y = 0; } // Koordinate.
        Tacka (double xx, double yy) { x = xx; y = yy; } // Konstruktor.
        double uzmi_x () const { return x; } // Dohvatanje koordinata.
        double uzmi_y () const { return y; }
        double operator- (const Tacka& T) const // Rastojanje do druge tačke.
            { return sqrt (pow(x-T.x,2) + pow(y-T.y,2)); }
        friend istream& operator>> (istream& ut, Tacka& T) // Čitanje.
            { return ut >> T.x >> T.y; }
        friend ostream& operator<< (ostream& it, const Tacka& T) // Pisanje.
            { return it << '(' << T.x << ',' << T.y << ')'; }
    }; // class Tacka
} // namespace Figure

#endif
```

Program 8.2 – Definicija klase tačaka (tacka4.h)

Boje se obično predstavljaju kao mešavina crvene, zelene i plave boje. Ovde je usvojeno da se intenziteti komponentnih boja prikazuju realnim brojevima u opsegu od 0 do 1. Tako trojka (0,0,0) predstavlja crnu boju, (1,1,1) belu boju, (1,0,0) crvenu boju itd.

Na početku programa 8.3 nalazi se pomoćna klasa G_neispravna_boja (izvedena iz klase Usluge::Greska) za prijavljivanje izuzetaka u slučajevima pokušaja stvaranja boje nedozvoljenih vrednosti komponenata. Predviđena je mogućnost dostavljanja nedozvoljenih vrednosti komponenata do rukovalaca izuzecima.

Sama klasa Boja je vrlo jednostavna. Omogućava stvaranje objekata, dohvatanje komponenata boje i čitanje i pisanje podataka o bojama. Podrazumevani konstruktor novi objekat inicijalizuje belom bojom. Skreće se pažnja na to da drugi konstruktor proverava ispravnost navedenih vrednosti komponenata boje i prijavljuje izuzetak ako je bar jedna komponenta izvan dozvoljenog opsega. Drugo potencijalno mesto pojave neispravnih vrednosti je prilikom čitanja komponenata boje operatorom >>. Pročitane vrednosti se posredno proveravaju pozivanjem odgovarajućeg konstruktora.

Centralnu klasu u ovom zadatku čini apstraktna klasa Figura koja obuhvata zajedničke osobine svih vrsta obojenih figura. Definicija te klase nalazi se u programu 8.4 iza jednostavne klase G_tacka_ne_pripada za prijavljivanje izuzetaka kada se konstatuje

8.4.1 Obrada obojenih geometrijskih figura u ravni △

```
// Definicija klase za boje (Boja).

#ifndef _boja_h_
#define _boja_h_

#include "greska.h"
using Usluge::Greska;
#include <iostream>
using namespace std;

namespace Figure {
    class G_neispravna_boja: public Greska { // KLASA ZA GREŠKU:
        double c, z, p; // Nedozvoljene komponente.
    public:
        G_neispravna_boja (): Greska ("Nedozvoljene komponente boje") {}
        G_neispravna_boja (double cc, double zz, double pp): Greska () { c = cc; z = zz; p = pp; }
    private:
        void pisi (ostream& it) const { // Pisanje poruke.
            if (ima_poruke ()) Greska::pisi(it);
            else it << "\n*** Neispravna boja: (" << c << ',' << z << ',' << p << ") ***\a\n";
        } // Dohv. nedozv. komp.
        double uzmi_c () const { return ima_poruke () ? 0 : c; }
        double uzmi_z () const { return ima_poruke () ? 0 : z; }
        double uzmi_p () const { return ima_poruke () ? 0 : p; }
    }; // class G_neispravna_boja

    class Boja { // KLASA ZA BOJU:
        double c, z, p; // Komponente boje.
    public:
        Boja () { c = z = p = 1; } // Konstruktor.
        Boja (double cc, double zz, double pp) {
            if (cc<0 || cc>1 || zz<0 || zz>1 || pp<0 || pp>1)
                throw G_neispravna_boja (cc, zz, pp);
            c = cc; z = zz; p = pp;
        }
        double uzmi_c () const { return c; } // Dohvatanje komponenata boje.
        double uzmi_z () const { return z; }
        double uzmi_p () const { return p; }
        friend istream& operator>> (istream& ut, Boja& b) { // Čitanje.
            double c, z, p; ut >> c >> z >> p;
            b = Boja (c, z, p); return ut;
        }
        friend ostream& operator<< (ostream& it, const Boja& b) // Pisanje.
            { return it << '(' << b.c << ',' << b.z << ',' << b.p << ')'; }
    }; // class Boja
} // namespace Figure

#endif
```

Program 8.3 – Definicija klase za boje (boja.h)

da tačka, koja za datu operaciju mora da pripada figuri, nije u unutrašnjosti ili na ivici figure.

```
// Definicija klase obojenih geometrijskih figura (Figura).
#ifndef _figura2_h_
#define _figura2_h_

#include "greska.h"
using Usluge::Greska;
#include "boja.h"
#include "tacka4.h"
#include <iostream>
#include <string>
using namespace std;

namespace Figure {
    class G_tacka_ne_pripada: public Greska { // KLASA ZA GREŠKU:
public:
    G_tacka_ne_pripada (): Greska ("Tacka ne pripada figuri") {}
}; // class G_tacka_ne_pripada

class Figura { // KLASA ZA FIGURU:
private:
    static int uk_id; // Poslednje korisćeni identifikator.
    int id; // Identifikator figure.
    Boja b; // Osnovna boja figure.
public:
    explicit Figura (Boja bb=Boja()): b(bb), id(++uk_id) {}
    Figura (const Figura& fig): b(fig.b), id(++uk_id) {}
    Figura& operator= (const Figura& fig) // Dodela vrednosti.
    { b = fig.b; return *this; }
    virtual ~Figura () {} // Virtuelni destruktur.
    virtual Figura* kopija () const =0; // Dinamička kopija.
    int uzmi_id () const { return id; }
    Boja boja () const { return b; } // Osnovna boja figure.
    virtual Boja boja (const Tacka& T) const { // Boja tačke.
        if (! pripada (T)) throw G_tacka_ne_pripada ();
        return b;
    }
private:
    virtual bool pripada (const Tacka& T) const=0; // Da li tačka pripada?
    friend bool operator< (const Tacka& T, const Figura& fig)
    { return fig.pripada (T); }
    virtual void pisi (ostream& it) const =0; // Pisanje figure.
    friend ostream& operator<< (ostream& it, const Figura& fig)
    { it << typeid (fig).name () [8] << fig.id << " boja=" << fig.b;
        fig.pisi (it); return it;
    }
}; // class Figura
} // namespace Figure

#endif

```

Program 8.4 – Definicija klase obojenih geometrijskih figura (figura2.h)

U klasi Figura predviđeno je automatsko obeležavanje svih figura u programu jedinstvenim uzastopnim celobrojnim identifikacionim brojevima. U tom cilju je definisan za-

8.4.1 Obrada obojenih geometrijskih figura u ravni △

jedničko polje uk_id čija se vrednost povećava pri svakom stvaranju novog objekta. Njegova trenutna vrednost dodeljuje se pojedinačnom polju id.

Prvi konstruktor novu figuru inicijalizuje zadatom bojom bb i sledećim po redu identifikacionim brojem (++uk_id). Podrazumevana vrednost za boju je rezultat podrazumevanog konstruktora klase Boja, a to je bela boja.

Drugi konstruktor je konstruktor kopije. Potreban je samo zato da obezbedi novi identifikacioni broj za kopiju svog parametra. Za razliku od uobičajenih konstruktora kopija, koji neizmenjeno kopiraju sve informacije iz izvorишnog objekta u novostvoreni objekat, ovde se kopira samo boja, a ne i identifikacioni broj.

Iz istog razloga je netipičan i operator za dodelu vrednosti. Prilikom dodele vrednosti određeni objekat već postoji i ima svoj identifikacioni broj. Taj broj ne treba menjati, jer se ne stvara novi objekat. Potrebno je iz izvorишnog objekta preneti samo boju figure.

Virtuelan destruktur praznog tela je uobičajeni element osnovnih klasa za ispravno polimorfno uništavanje objekata tipa izvedenih klasa. Isto tako, apstraktna metoda kopija () je čest element ovakvih klasa za polimorfno kopiranje objekata tipa izvedenih klasa.

Naredne metode u programu 8.4 daju neke informacije o figurama. Vrednost metode uzmi_id () je identifikacioni broj objekta. Vrednost prve metode boja () je vrednost polja b. To je boja jednostavnih figura, odnosno osnovna boja složenih figura (delova koji nisu pokriveni umetnutim komponentnim figurama).

Druga metoda boja () treba da određuje boju tačke unutar figure koja se navodi kao parametar. Greška je ako ta tačka ne pripada figuri, što se proverava pozivanjem privatne apstraktne metode pripada (). Ako tačka pripada figuri prikazana metoda boja () vraća vrednost polja b. Ovo rešenje zadovoljava za sve jednostavne figure čije su sve tačke iste boje. Metoda je virtuelna, da bi u klasama koje predstavljaju višebojne figure mogla da se zameni odgovarajućom specifičnom metodom.

Za javno ispitivanje da li tačka T pripada figuri fig izvršeno je preklapanje operatorka < čime je omogućeno da se to pitanje postavi izrazom T < fig. Pošto prvi operand operatorka < nije tipa Figura, ova operatorska funkcija ne može da bude metoda klase i kao takva, ne može da bude ni virtuelna ni apstraktna. To je pravi razlog uvođenja privatne apstraktne metode pripada ().

Slično rešenje je već korišćeno i u ranijim zadacima pri obezbeđivanju polimorfognog čitanja i pisanja podataka korišćenjem operatorka >> i <<. U klasi Figura predviđeno je samo polimorfno ispisivanje sadržaja figura kombinacijom privatne apstraktne metode pisi () i operatorske funkcije za operatorka <<. Polimorfno učitavanje figura nije predviđeno jer kod složenih figura, koje sadrže druge figure, ne bi moglo da se napravi rešenje koje ne zavisi od skupa postojećih vrsta figura. Uvođenje novih vrsta figura iziskivalo bi izmene u klasama složenih figura. To nije slučaj prilikom ispisivanja.

Metoda pisi () u programu 8.4 je apstraktna. Ispisivanje zajedničkih podataka za sve vrste figura je stavljeno u operatorsku funkciju za operatorka <<. Taj ispis sadrži jednoslovnu oznaku vrste figure, njen identifikacioni broj (na primer, K5 za krug čiji je broj 5) i na kraju boju figure. Tek posle toga se poziva metoda pisi () radi polimorfognog dopisivanja preostalih specifičnih podataka za konkretnu vrstu figure. Skreće se pažnja na to da, pošto se klase za figure nalaze u prostoru imena Figure, vrednost izraza typeid (fig).name () za klasu Krug je Figure::Krug, pa početno slovo imena same klase ima indeks 8.

Pošto u definiciji klase Figura sve metode koje mogu da se definisu (koje nisu apstraktne) definisane su u samoj klasi, izvan klase jedino treba definisati zajedničko polje uk_id (program 8.5).

```
// Definicija zajedničkog polja uz klasu Figura.
#include "figura2.h"

int Figura::Figura::uk_id = 0;
```

Program 8.5 – Definicija zajedničkog podatka uz klasu Figura (figura2.C)

Program 8.6 prikazuje definiciju klase Krug za obojene krugove. Jedino se skreće pažnja da se u metodi kopija() novi objekat inicijalizuje automatski generisanim konstruktorom kopije. Pošto taj konstruktor za inicijalizaciju polja osnovne klase poziva konstruktor kopije osnovne klase, kopija će imati drugačiji identifikacioni broj od originala.

```
// Definicija klase obojenih krugova (Krug).

#ifndef _krug3_h_
#define _krug3_h_

#include "figura2.h"

namespace Figure {
    class Krug: public Figura {
        Tacka C; // Centar.
        double r; // Poluprečnik.
    public:
        explicit Krug (double rr=1, Tacka cc=Tacka(), Boja bb=Boja())
            : Figura (bb), C(cc), r(rr) {}
        Krug* kopija () const { return new Krug (*this); } // Kopija.

        bool pripada (const Tacka& T) const // Da li tačka pripada?
            { return C - T <= r; }
        virtual void pisi (ostream& it) const // Pisanje.
            { it << " C=" << C << " r=" << r; }
    }; // class Krug
} // namespace Figure
#endif
```

Program 8.6 – Definicija klase obojenih krugova (krug3.h)

Sledećim malim programom može da se uveri u to:

```
#include "krug3.h"
#include <iostream>
using namespace Figure;
using namespace std;

int main ()
{
    Krug k1 (1, Tacka(1,1), Boja (1,1,1)); cout << k1 << endl;
    Krug k2 (2, Tacka(2,2), Boja (0,0,0)); cout << k2 << endl;
    Krug k3 (k1);
    k3 = k2;
}
```

8.4.1 Obrada obojenih geometrijskih figura u ravni ▾

Pri definisanju kruga k3 u trećem redu glavne funkcije koristi se konstruktor kopije. U četvrtom redu se koristi operator za dodelu vrednosti. Iz rezultata programa:

```
K1 boja=(1,1,1) C=(1,1) r=1
K2 boja=(0,0,0) C=(2,2) r=2
K3 boja=(1,1,1) C=(1,1) r=1
K3 boja=(0,0,0) C=(2,2) r=2
```

vidi se da je prilikom inicijalizacije novi objekat dobio novi identifikacioni broj, a prilikom dodelje vrednosti identifikacioni broj se nije promenio. To je zato, jer generisani operator za dodelu vrednosti za kopiranje polja osnovne klase poziva operator za dodelu vrednosti osnovne klase.

Program 8.7 prikazuje definiciju klase za obojene pravougaonike čije su ivice paralelne koordinatnim osama. U tom slučaju pravougaonik je jednoznačno određen koordinatama dva naspramna temena. Parametri konstruktora P i Q su tačke koje predstavljaju takva dva temena. Radi jednostavnijeg korišćenja objekta, polja A i C se inicijalizuju tako da predstavljaju donje levo i gornje desno teme, bez obzira koja temena i po kom redosledu predstavljaju parametri P i Q. Potencijalna greška je da ti parametri predstavljaju dva susedna, umesto naspramna temena. Taj problem se u prikazanoj klasi Pravoug ne razmatra.

```
// Definicija klase obojenih pravougaonika (Pravoug).

#ifndef _pravoug2_h_
#define _pravoug2_h_

#include "figura2.h"

namespace Figure {
    class Pravoug: public Figura {
        Tacka A, C; // Temena dole-levo i gore-desno.
    public:
        Pravoug (Tacka P, Tacka Q, Boja b=Boja())
            : Figura (b),
              A(P.uzmi_x()<Q.uzmi_x() ?P.uzmi_x():Q.uzmi_x(),
                  P.uzmi_y()<Q.uzmi_y() ?P.uzmi_y():Q.uzmi_y()),
              C(P.uzmi_x()<Q.uzmi_x() ?Q.uzmi_x():P.uzmi_x(),
                  P.uzmi_y()<Q.uzmi_y() ?Q.uzmi_y():P.uzmi_y()) {}

        Pravoug* kopija () const { return new Pravoug (*this); } // Kopija.

        bool pripada (const Tacka& T) const // Da li tačka pripada?
            { return A.uzmi_x()<=T.uzmi_x() && T.uzmi_x()<=C.uzmi_x() &&
                  A.uzmi_y()<=T.uzmi_y() && T.uzmi_y()<=C.uzmi_y(); }

        virtual void pisi (ostream& it) const // Pisanje.
            { it << " A=" << A << " C=" << C; }
    }; // class Pravoug
} // namespace Figure
#endif
```

Program 8.7 – Definicija klase obojenih pravougaonika (pravoug2.h)

Program 8.8 prikazuje definiciju klase Trougao za obojene trouglove. Kao polja postoje tri tačke koje predstavljaju temena trougla.

```
// Definicija klase obojenih trouglova (Trougao).

#ifndef _trougao2_h_
#define _trougao2_h_

#include "figura2.h"
#include <cmath>
using namespace std;

namespace Figure {
    class Trougao: public Figura {
        Tacka A, B, C; // Temena.
        public: // Konstruktor.
            Trougao (Tacka P, Tacka Q, Tacka R, Boja b=Boja())
                : Figura (b), A(P), B(Q), C(R) {}
            Trougao* kopija () const { return new Trougao (*this); } // Kopija.
        private:
            double P () const {
                double a = B - C, b = C - A, c = A - B, s = (a + b + c) / 2;
                return sqrt (s * (s-a) * (s-b) * (s-c));
            }
            bool pripada (const Tacka& T) const { // Da li tačka pripada?
                return fabs (Trougao(T,A,B).P() + Trougao(T,B,C).P() +
                    Trougao(T,C,A).P() - P()) < 1E-8;
            }
            virtual void pisi (ostream& it) const // Pisanje.
                { it << " A=" << A << " B=" << B << " C=" << C; }
    }; // class Trougao
} // namespace Figure

#endif
```

Program 8.8 – Definicija klase obojenih trouglova (trougao2.h)

Da li tačka T pripada trouglu ispituje se na sledeći način: Ako se tačka T spoji s temenima A, B i C trougla dobijaju se tri trougla ΔTAB , ΔTBC i ΔTCA . Ako tačka T pripada trouglu ΔABC zbir površine ta tri trougla jednak je površini ispitanoj trougla. Pošto pri radu s realnim brojevima ne mogu da se očekuju potpuno tačni rezultati, u metodi `pripada()` je uzeto da tačka pripada trouglu ako je absolutna vrednost razlike površine trougla i zbiru površina pomenuta tri trougla manja od 10^{-8} . Za računanje površine trouglova uvedena je privatna metoda `P()`. Metoda `P()` je privatna zato što ni kod ostalih vrsta figura nije predviđeno računanje površine. Ovako za javnost ne postoji ni računanje površine trouglova.

Mnogouga je određen nizom tačaka koje predstavljaju temena mnogouglia. Za usklađivanje niza tačka u klasi `Mnogouga` u programu 8.9 koristi se generička klasa `vector<Tacka>` s tipom `Tacka` kao argumentom šablonu.

Jedini konstruktor u klasi `Mnogouga` polje `temena` inicijalizuje kopijom vektora tačaka koji je njegov parametar. Naravno, pri tome koristi konstruktor kopije klase `vector<Tacka>`.

Bez obzira što objekat `temena` sadrži dinamičku strukturu podataka nije potrebno praviti konstruktor kopije, destruktorni operator za dodelu vrednosti. Te metode se automatski generišu tako da, uz pozivanje odgovarajućih metoda klase `vector<Tacka>`, obezbede ispravno kopiranje i uništavanje objekata tipa `Mnogouga`. Skreće se pažnja na to da je za

```
// Definicija klase obojenih mnogouglova (Mnogouga).

#ifndef _mnogouga_h_
#define _mnogouga_h_

#include "figura2.h"
#include <vector>
using namespace std;

namespace Figure {
    class Mnogouga: public Figura {
        vector<Tacka> temena; // Temena mnogouglia.
        public: // Konstruktor.
            explicit Mnogouga (const vector<Tacka>& tem, Boja b=Boja())
                : Figura (b), temena (tem) {}
            Mnogouga* kopija () const { return new Mnogouga (*this); } // Kopija.
        private:
            bool pripada (const Tacka& T) const; // Da li tačka pripada?
            virtual void pisi (ostream& it) const; // Pisanje.
    }; // class Mnogouga
} // namespace Figure

#endif
```

Program 8.9 – Definicija klase obojenih mnogouglova (mnogouga.h)

polimorfno uništavanje objekata ove klase važno da destruktorni operatori u osnovnoj klasi bude viruelan.

Program 8.10 prikazuje metode klase `Mnogouga` koje zbog njihovih složenosti nisu definisane unutar same klase.

```
// Definicija metoda uz klasu Mnogouga.

#include "mnogouga.h"
#include "trougao2.h"

bool Figure::Mnogouga::pripada (const Tacka& T) const { // Da li pripada?
    for (unsigned i=1; i<temena.size()-1; i++)
        if (T < Trougao(temena[0],temena[i],temena[i+1])) return true;
    return false;
}

void Figure::Mnogouga::pisi (ostream& it) const { // Pisanje.
    it << " temena=[";
    for (unsigned i=0; i<temena.size(); i++)
        it << temena[i]
            << (i==temena.size()-1 ? ")" : i%6==2 ? "\n\t" : " ");
}
```

Program 8.10 – Definicija metoda uz klasu `Mnogouga` (mnogouga.C)

U metodi `pripada()` mnogouga se podeli u trouglove povlačenjem dijagonale od temena s indeksom 0 do svih ostalih temena. Tačka pripada mnogouglu ako pripada jednom

od tih trouglova. Treba napomenuti da se na ovaj način tačni rezultati dobijaju samo za konveksne mnogouglove.

Metoda `pisi()` temena mnogougla ispisuje unutar para uglastih zagrada, međusobno razdvojenih razmacima. U prvom redu, u nastavku zajedničkih podataka za sve vrste figura, ispisac će se tri temena. Preostala temena, do šest u svakom redu, ispisuju se u narednim redovima. Prepušta se čitaocu da razjasni kako se to postiže uslovnim izrazom koji je naveden kao drugi podatak za ispisivanje.

Posebno se skreće pažnja, u obe prethodne metode, na korišćenje operatora za indeksiranje za objekat temena koji je tipa `vector<Tacka>` i na dohvatanje broja temena metodom `size()`.

Poslednju vrstu figura obradenih u ovom zadatku čine složene figure koje u sebi mogu da sadrže i druge figure. Ta vrsta figura je nazvana *crtež*.

Crtež je pravougaona figura s ivicama paralelnim koordinatnim osama. Zadaje se početko koordinata donjeg levog ugla crteža i širinom i visinom crteža. Crtež se stvara prazan i posle može da se doda proizvoljan broj figura, uključujući i druge crteže. Koordinate figura unutar crteža računaju se relativno u odnosu donji levi ugao crteža. Delovi umetnutih figura koji se nalaze i izvan granica crteža su nevidljivi. Ako se komponentne figure preklapaju, smatra se da kasnije dodata figura pokriva ranije dodatu figuru.

Program 8.11 prikazuje definiciju klase `Crtez` koja ostvaruje opisane obojene crteže.

Za smeštanje figura u crtež predviđeno je polje `figure` koje je vektor pokazivača na figure (tip `vector<Figura*>`, dakle, koristi se generička klasa `vector<>`). Skreće se pažnja na to da ne može da se pravi vektor figura, jer klasa `Figura` je apstraktna klasa.

Pošto će same figure biti u dinamičkoj zoni memorije pod kontrolom klase `Figura`, neophodno je da u klasi postoje konstruktor kopije, destruktur i operator za dodelu vrednosti. Privatne metode `kopiraj()` i `brisi()` su podrška za njihovo ostvarivanje.

Prvi konstruktor stvara prazan crtež na osnovu nepromenljivih osobina: donjeg levog temena, širine, visine i osnovne boje. Polje figure se inicijalizuje automatskim pozivanjem podrazumevanog konstruktora klase `vector<Figura*>` tako da sadrži nula elemenata.

Konstruktor kopije prvo poziva konstruktoru kopije osnovne klase (`Figura(crt)`) za inicijalizaciju nasledenih polja (na osnovu odgovarajućih polja iz `crt`) i posle toga prekopira specifični sadržaj izvorišnog crteža (`kopiraj(crt)`).

Operator za dodelu vrednosti prvo uništava stari specifični sadržaj tekućeg objekta (`brisi()`) i posle poziva operatorka za dodelu vrednosti osnovne klase (`Figura::operator=(crt)`) za dodelu vrednosti nasledenim poljima (koji će prvo uništiti stari sadržaj nasledenih polja i posle prekopirati sadržaj nasledenih polja iz izvorišnog objekta), da bi na kraju prekopirao specifični sadržaj izvorišnog objekta (`kopiraj(crt)`). Ovim je obezbeđeno opšte načelo po kome redosled uništavanja treba da je obrnut u odnosu na redosled stvaranja: pošto se specifični deo stvara posle nasledenog dela, specifični deo treba da se uništava pre nasledenog dela.

Destruktor, na uobičajeni način, samo poziva metodu `brisi()` koja će da uništi specifični sadržaj objekta. Posle toga se automatski poziva destruktur osnovne klase za uništavanje nasledenih delova objekta.

Metoda `kopija()`, kao i u ranijim slučajevima, kopiju svog parametra pravi pomoću konstruktora kopije svoje klase.

8.4.1 Obrada obojenih geometrijskih figura u ravni

```
// Definicija klase obojenih crteža (Crtez).

#ifndef _CRTEZ_H_
#define _CRTEZ_H_

#include "figura2.h"
#include "pravoug2.h"
#include <vector>
using namespace std;

namespace Figure {
    class Crtez: public Figura {
        Tacka A;
        double s, v;
        vector<Figura*> figure;
        void kopiraj (const Crtez& crt);
        void brisi ();
    public:
        explicit Crtez (const Tacka& T=Tacka(),
                        double sir=1, double vis=1, Boja bb=Boja());
        Figura (bb), s(sir), v(vis), A(T) {}
        Crtez (const Crtez& crt): Figura(crt) | kopiraj (crt);
        Crtez& operator= (const Crtez& crt) | // Dodela vrednosti.
            if (this != &crt) {
                brisi(); Figura::operator=(crt); kopiraj (crt);
            }
        return *this;
    }
    ~Crtez () { brisi (); } // Destruktor.
    Crtez* kopija () const | return new Crtez (*this); // Kopija.
    Crtez& operator+= (Figura* fig) // Dodavanje figure.
        { figure.push_back (fig); return *this; }
    Boja boja (const Tacka& T) const; // Boja tačke.
private:
    bool pripada (const Tacka& T) const // Da li tačka pripada?
        { return T < Pravoug(A.Tacka(A.uzmi_x() + s, A.uzmi_y() + v)); }
    void pisi (ostream& it) const; // Pisanje.
}; // class Crtez
} // namespace Figure

#endif
```

Program 8.11 – Definicija klase obojenih crteža (`crtez.h`)

Dodavanje figure crtežu ostvareno je preklapanjem operatorka `+=`. Desni operand je pokazivač na figuru koji se prosti dodaje na kraj vektora figure (`figure.push_back (fig)`). Ne pravi se kopija izvorišne figure, tj. smatra se da se time figura predaje u nadležnost crteža. Prvobitni vlasnik ne sme da uništava predatu figuru, a klasa `Crtez` treba da je uništi pri uništavanju samog crteža.

Pripadnost tačke crtežu u metodi `pripada()` proverava se ispitivanjem da li pripada pravougaoniku koji bi se poklopio sa samim crtežom.

Program 8.12 prikazuje definicije metoda klase `Crtez` koje su u definiciji klase samo deklarisane.

```
// Definicije metoda u klasu Crtez.

#include "crtez.h"
#include <string>
using namespace std;

void Figure::Crtez::kopiraj (const Crtez& crt) { // Kopiranje crteža.
    A = crt.A; s = crt.s; v = crt.v;
    for (vector<Figura*>::const_iterator i=figure.begin();
         i!=figure.end(); figure.push_back((*i++)->kopija()));
}

void Figure::Crtez::brisi () { // Uništavanje crteža.
    for (vector<Figura*>::const_iterator i=figure.begin();
         i!=figure.end(); delete *i++);
    figure.clear ();
}

Figure::Boja Figure::Crtez::boja (const Tacka& T) const { // Boja tačke.
    if (! pripada (T)) throw G_tacka_ne_pripada ();
    Tacka T1 (T.uzmi_x()-A.uzmi_x(), T.uzmi_y()-A.uzmi_y());
    for (vector<Figura*>::const_reverse_iterator i=figure.rbegin();
         i!=figure.rend(); i++)
        try { return (*i)->boja (T1); } catch (G_tacka_ne_pripada) {}
    return Figura::boja ();
}

void Figure::Crtez::pisi (ostream& it) const { // Pisanje.
    static string marg = "";
    it << "A=" << A << " s=" << s << " v=" << v;
    marg += " ";
    for (vector<Figura*>::const_iterator i=figure.begin();
         i!=figure.end(); it << endl << marg << **i++);
    marg.erase (0, 3);
}

```

Program 8.12 – Definicije metoda u klasu Crtez (crtez.C)

Metoda `kopiraj()` posle dodele vrednosti „jednostavnim” poljima, polimorfno obrazuje kopije pojedinih figura u izvorишnom crtežu (`(*i++)->kopija()`) i dobijene pokazivače stavlja u vektor u tekućem objektu (`figure.push_back(...)`). Skreće se pažnja na korišćenje iteradora za obilazak vektora `figure`. Iterator `i` pokazuje na tekući element vektora. Vrednost izraza `*i` je sam element (u ovom slučaju tipa `Figura*`). Zbog višeg prioriteta postfiksнog oblika operatora `++` od unarnog operatora `*`, u izrazu `*i++` povećava se vrednost iteradora, tj. pomera se na sledeći element vektora.

Metoda `brisi()` prvo uništi sve figure u crtežu (`delete *i++`) i na kraju uništi dinamički niz koji se nalazi u objektu `figure`, pozivajući metodu `clear()` klase `vector<Figura*>`.

Ako metoda `boja()` za određivanje boje tačke `T` u tekućem crtežu ustanovi da ta tačka ne pripada crtežu, prijavljuje izuzetak tipa `G_tacka_ne_pripada`. Ako je tačka `T` prihvatljiva, a pošto se koordinate u komponentnim figurama računaju u odnosu na donji levi ugao crteža, neophodno je odrediti relativne koordinate te tačke u odnosu na donji levi ugao crteža. Te koordinate se stavljuju u tačku `T1`. U ciklusu koji sledi traži se prva komponentna figura od kraja vektora (pošto kasnije dodate figure pokrivaju ranije figure) kojoj ta

8.4.1 Obrada obojenih geometrijskih figura u ravni ▲

tačka pripada korišćenjem obrnutog iteradora. Od svake figure `(*i)` se traži koje je boje tačka `T1`, polimornim pozivanjem metode `boja()` iz klase kojoj pripada ispitana figura. Ako tačka pripada figuri, dobijeni rezultat je istovremeno i vrednost metode `Crtez::boja()`. Ako tačka ne pripada figuri `*i`, metoda `boja()` iz klase ispitane figure će prijaviti izuzetak tipa `G_tacka_ne_pripada`. Rukovalac tim izuzecima u metodi `Crtez::boja()` ne radi ništa (ima prazno telo), ali omogućava da se ciklus ispitivanja komponentnih figura nastavi dalje. Prirodnji završetak ciklusa označava da tačka `T` (odносно `T1`) ne pripada nijednoj od figura u crtežu, pa kao rezultat daje se osnovna boja crteža.

Metoda `pisi()` prvo iza zajedničkih podataka za sve vrste figura dopisuje proste podatke o crtežu (donje levo teme, širinu i visinu). Posle toga u zasebnim redovima polimorfno ispisuje podatke o komponentnim figurama. Da bi ispis bio pregledniji na početku svakog reda ostavlja se marga u vidu određenog broja praznih mesta. Imajući u vidu da između komponentnih figura mogu da budu i crteži do proizvoljne dubine, širina marge treba da raste pri svakom koraku u dubinu. Zbog toga je uvedena trajna (`static`) promenljiva `marg` tipa `string` čija vrednost se inicijalizuje praznom niskom ("") samo pri prvom pozivanju metode `pisi()`. Kasnije u njoj se sačuvaju zatećene vrednosti od ranije. Na sadržaj promenljive `marg` se pre početka ciklusa za ispisivanje komponentnih figura dodaje niska od tri znaka razmaka (`marg+=...`). Po završetku ciklusa iz nje se izbacuju tri znaka razmaka (`marg.erase (0, 3)`). Kad god se u toku ispisivanja nailazi na unutrašnji crtež, sadržaj promenljive `marg` se produžuje za tri znaka razmaka pa sadržaj unutrašnjeg crteža biće odmaknut od početka reda više nego sadržaj spoljašnjeg crteža. Kad god se završi ispisivanje unutrašnjeg crteža marga se skraćuje za tri znaka, pa nastavak spoljašnjeg crteža se ispisuje početnom marginom. Po završetku ispisivanja crteža na osnovnom nivou, promenljiva `marg` će uvek sadržati praznu nisku, pa kasniji zahtevi za ispisivanje crteža će početi od početka redova, bez obzira što se promenljiva `marg` više ne inicijalizuje. Skreće se još pažnja na to da se iza poslednjeg ispisanih reda ne prelazi u novi red, pošto se na takav način postupa i kod svih ostalih vrsta figura. Eventualni prelazak u novi red je prepušten korisniku klase.

Programi 8.13 i 8.14 služe za ispitivanje klasa obojenih figura.

Program 8.13 prikazuje pomoćnu funkciju `citaj()` za čitanje podataka o jednoj figuri. Vrednost funkcije je pokazivač na pročitanu figuru ili nula ako korisnik označi da ne želi uneti ništa. Očigledno, ova funkcija zavisi od postojećih vrsta figura i uvođenje nove vrste figura zahteva njene izmene.

Da bi dijalog unosa podataka za slučaj crteža pratilo strukturu uklapanja crteža, slično metodi `pisi()` u programu 8.12, i ovde je uvedena trajna promenljiva `marg`. Njen početni sadržaj je prazna niska. Sadržaj te promenljive ispisuje se ispred svakog pitanja kojima se traži unošenje potrebnih podataka za figuru koja se učitava.

Na početku funkcije `citaj()` pročita se oznaka vrste figure koja želi da se pročita. Očekuje se početno slovo vrste figura (K – krug, P – pravougaonik, T – trougao, M – mnogougao, C – crtež) ili tačka ako se ne želi uneti ništa (u stvari, bilo šta osim navedenih slova označava tu želju).

Za čitanje podataka za svaku vrstu podataka postoji po jedna grana u naredbi `switch` koja sledi čitanje vrste figure. Svaka grana je uobičena kao blok, da bi se ograničio doseg promenljivih uvedenih u tim granama. Posle čitanja potrebnih podataka stvara se dinamički objekat odgovarajućeg tipa i pokazivač na tu figuru je vrednost funkcije `citaj()`. Pošto se svaka grana završava naredbom `return`, u njima nema uobičajenih naredbi `break`.

```

// Program za ispitivanje klasa obojenih geometrijskih figura.

#include "krug3.h"
#include "pravoug2.n"
#include "trouga2.h"
#include "mnogoug.h"
#include "crtez.h"
using namespace Figure;
#include <iomanip>
using namespace std;

Figura* citaj () {
    static string marg = "";           // ČITANJE JEDNE FIGURE:
    cout << endl << marg              // Leva margina dijaloga.

    switch (vrs) {
        case 'k': case 'K': {         // Čitanje kruga.
            cout << marg << "Boja figure (c,z,p)? "; Boja b; cin >> b;
            cout << marg << "Centar (x,y)? "; Tacka C; cin >> C;
            cout << marg << "Poluprecnik? "; double r; cin >> r;
            return new Krug (r, C, b);
        }
        case 'p': case 'P': {         // Čitanje pravougaonika.
            cout << marg << "Boja figure (c,z,p)? "; Boja b; cin >> b;
            cout << marg << "Jedno teme (x,y)? "; Tacka A; cin >> A;
            cout << marg << "Suprotno teme (x,y)? "; Tacka C; cin >> C;
            return new Pravoug (A, C, b);
        }
        case 't': case 'T': {         // Čitanje trougla.
            cout << marg << "Boja figure (c,z,p)? "; Boja b; cin >> b;
            cout << marg << "Prvo teme (x,y)? "; Tacka A; cin >> A;
            cout << marg << "Drugo teme (x,y)? "; Tacka B; cin >> B;
            cout << marg << "Treće teme (x,y)? "; Tacka C; cin >> C;
            return new Trougao (A, B, C, b);
        }
        case 'm': case 'M': {         // Čitanje mnogouglja.
            cout << marg << "Boja figure (c,z,p)? "; Boja b; cin >> b;
            cout << marg << "Broj temena? "; int n; cin >> n;
            vector<Tacka> temena (n);
            for (int i=0; i<n; i++) {
                cout << marg << setw(2) << (i+1) << ". teme (x,y)? ";
                cin >> temena[i];
            }
            return new Mnogoug (temena, b);
        }
        case 'c': case 'C': {         // Čitanje crteza.
            cout << marg << "Boja figure (c,z,p)? "; Boja b; cin >> b;
            cout << marg << "Dole-levo (x,y)? "; Tacka A; cin >> A;
            cout << marg << "Sirina, visina? ";
            double sir, vis; cin >> sir >> vis;
            Crtez* crt = new Crtez (A, sir, vis, b);
            marg += " ";
            while (Figura* fig = citaj ()) *crt += fig; // crteža povećanom
            marg.erase(0,2);                           // marginom.
            return crt;
        }
        default: return 0;             // "Prazna" figura.
    }
}

```

Program 8.13 – Ispitivanje klasa obojenih figura (figura2t.C, prvi deo)

8.4.1 Obrada obojenih geometrijskih figura u ravni ▾

Od svih vrsta figura samo čitanje crteža iziskuje objašnjenja. Posle čitanja prostih parametara (boje, koordinata donjeg levog temena, širine i visine) naprvi se prazan crtež s tim parametrima. Sadržaj promenljive `marg` se produži za dva znaka razmaka da bi nastavak dijaloga bio odmaknut od početka reda, istakavši time činjenicu da se unose podaci za figure koje su deo crteža. Posle se ulazi u ciklus čitanja figura koji se završava kada vrednost rekurzivno pozivane funkcije `citaj()` bude nula. Promenljiva za prihvatanje dobijenog pokazivača definiše se u uslovu naredbe `while`. Njen doseg je samo telo ciklusa, što je u ovom slučaju dovoljno. Po izlasku iz ciklusa sadržaj promenljive `marg` se skrati za dva razmaka i time je nastavak dijaloga odmaknut od početka reda, kao i na početku unošenja crteža. Skreće se pažnja na to da ako se u crtež stavlja crtež, margina će biti dva puta produžavana, pa dijalog za sadržaj unutrašnjeg crteža biće više odmaknut od početka reda od dijaloga za sadržaj spoljašnjeg crteža. Po završetku čitanja sadržaja crteža na osnovnom nivou završna vrednost promenljive `marg` je uvek prazna niska.

Program 8.14 prikazuje glavnu funkciju koja pročita jednu figuru, ispiše pročitanu figuru i posle čita tačke za koje određuje boju, sve dok za koordinatu `x` ne pročita vrednost 10^{38} . Rad programa je interesantniji ako se unese crtež koji sadrži nekoliko figura koje se i delimično preklapaju. Među tim figurama, naravno, mogu da budu i crteži.

```

int main () {                                // GLAVNA FUNKCIJA:
    Figura* fig = citaj ();
    cout << endl << *fig << endl << endl;
    while (true) {
        cout << "Tacka (x,y)? "; Tacka T; cin >> T;
        if (T.uzmi_x() == 1E38) break;
        try {
            cout << "Boja tacke: " << fig->boja (T) << endl;
        } catch (G_tacka_ne_pripada g) { cout << g; }
    }
}

```

Program 8.14 – Ispitivanje klasa obojenih figura (figura2t.C, drugi deo)

Rezultat 8.1 prikazuje primer rada programa 8.14.

```

Vrsta (K,P,T,M,C,..)? c
Boja figure (c,z,p)? .9 .9 .9
Dole-levo (x,y)? 3 1
Sirina, visina? 15 10
Vrsta (K,P,T,M,C,..)? p
Boja figure (c,z,p)? 1 0 0
Jedno teme (x,y)? 9 2
Suprotno teme (x,y)? -2 5
Vrsta (K,P,T,M,C,..)? m
Boja figure (c,z,p)? 0 1 0
Broj temena? 5
1. teme (x,y)? 8 2
2. teme (x,y)? 11 4
3. teme (x,y)? 12 8
4. teme (x,y)? 7 7
5. teme (x,y)? 6 4
Vrsta (K,P,T,M,C,..)? c
Boja figure (c,z,p)? .5 .5 .5
Dole-levo (x,y)? 9 6
Sirina, visina? 9 6
Vrsta (K,P,T,M,C,..)? t
Boja figure (c,z,p)? 0 1 1
Prvo teme (x,y)? 1 1
Drugo teme (x,y)? 5 1
Treće teme (x,y)? 1 3
Vrsta (K,P,T,M,C,..)? p
Boja figure (c,z,p)? 1 0 1
Jedno teme (x,y)? 4 4
Suprotno teme (x,y)? 8 6
Vrsta (K,P,T,M,C,..)? .
Vrsta (K,P,T,M,C,..)? k
Boja figure (c,z,p)? 0 0 1
Centar (x,y)? 5 6
Poluprecnik? 3
Vrsta (K,P,T,M,C,..)?
C1 boja=(0.9,0.9,0.9) A=(3,1) s=15 v=10
P2 boja=(1,0,0) A=(-2,2) C=(9,5)
M3 boja=(0,1,0) temena={(8,2) (11,4) (12,8)
(7,7) (6,4)}
C4 boja=(0.5,0.5,0.5) A=(9,6) s=9 v=6
T5 boja=(0,1,1) A=(1,1) B=(5,1) C=(1,3)
P6 boja=(1,0,1) A=(4,4) C=(8,6)
K7 boja=(0,0,1) C=(5,6) r=3
Tacka (x,y)? 6 2
Boja tacke: (0.9,0.9,0.9)
Tacka (x,y)? 11 4
Boja tacke: (0,1,0)
Tacka (x,y)? 14 8.5
Boja tacke: (0,1,1)
Tacka (x,y)? 17 9
Boja tacke: (0.5,0.5,0.5)
Tacka (x,y)? 2 4
*** Tacka ne pripada figuri ***
Tacka (x,y)? 1e38 0

```

Rezultat 8.1 – Obrada obojenih geometrijskih figura programom 8.14

9 Ulaz i izlaz Δ

Ulaz i izlaz podataka treba da omogućava komunikaciju programa sa spoljnjim svetom radi unošenja podataka za obradu i prikazivanje ili odlaganje rezultata. Zbog toga ne može da se zamisli program bez ulaza i izlaza podataka. Uprkos tome, ulaz i izlaz podataka nije deo jezika C++ u tom smislu što ne postoje zasebne naredbe za izvođenje tih radnji. Umesto toga postoje odgovarajuće bibliotečke klase za njihovo ostvarivanje.

Pošto se, bez obzira na fizičku prirodu izvorišta i odredišta, prenos podataka svodi na prenos nizova bajtova, oni se zajedničkim imenom nazivaju **tokovi** podataka. Tokovi koji su izvorišta podataka nazivaju se **ulazni tokovi**, a odredišta podataka **izlazni tokovi**.

Pošto nema sústinske razlike između nizova bajtova na ulazno-izlaznim uređajima (tastaturi, ekranu, u datoteci itd.) i u memoriji, tokovi za prenos podataka mogu biti i u memoriji, većina radnji s tokovima su istovetne, bez obzira gde su oni smešteni.

Rad s tokovima u jeziku C++ realizuje se odgovarajućim klasama. Konkretni tokovi predstavljaju se pomoću primeraka tih klasa. Klase za tokove su deo standardne biblioteke.

Za rad s datotekama (tokovima na magnetnim diskovima) postoje tri klase opisane u zaglavju **<fstream>**. Klasa **ofstream** predviđena je samo za izlazne datoteke, klasa **ifstream** samo za ulazne datoteke, dok klasa **fstream** za ulazno-izlazne datoteke.

Za rad s tokovima u operativnoj memoriji takođe postoje tri klase. Opisane su u zaglavju **<sstream>**. Klasa **ostringstream** služi samo za smeštanje podataka u tokove, klasa **istringstream** samo za uzimanje podatka iz tokova, dok klasa **stringstream** omogućava kako uzimanje tako i smeštanje podatka u tokove. Kao što se iz imena klasa da naslutiti, u ove tokove se uskladištavaju tekstovi.

Sve gore pomenute klase izvedene su iz odgovarajućih klasa **ostream** i **istream** koje ostvaruju efektivni prenos podatka sa ili bez ulazno-izlazne konverzije. Ove klase su zajedničke za sve vrste tokova bez obzira na vrstu nosioca podataka i definisane su u zaglavju **<iostream>**.

Postoje četiri standardna toka (primeraka klasa **ostream** ili **istream**) koji se automatski stvaraju na početku izvršavanja svakog programa:

cin – glavni (standardni) ulaz tipa **istream**. Predstavlja tastaturu dok se ne izvrši skretanje glavnog ulaza unutar samog programa ili u komandi operativnog sistema za izvršavanje programa.

cout – glavni (standardni) izlaz tipa **ostream**. Predstavlja ekran dok se ne izvrši skretanje glavnog izlaza unutar samog programa ili u komandi operativnog sistema

za izvršavanje programa. Koristi se za ispisivanje podataka koji čine rezultate izvršavanog programa.

`cerr` – (standardni) izlaz za poruke tipa `ostream`. Predstavlja ekran dok se ne izvrši skretanje izlaza za poruke unutar samog programa. Koristi se obično za ispisivanje poruka o greškama.

`clog` – (standardni) izlaz za zabeleške tipa `ostream`. Predstavlja ekran dok se ne izvrši skretanje izlaza za zabeleške unutar samog programa. Koristi se obično za „vođenje dnevnika” o događajima za vreme izvršavanja programa.

U narednim odeljcima prikazane su važnije metode za rad s tokovima iz spomenutih klasa. Pregled nije potpun, ali je sasvim dovoljan za rešavanje čak i relativno složenih problema pri radu s tokovima.

Treba još napomenuti da funkcije za ulaz i izlaz korišćene u jeziku C, koje su opisane u zaglavljima `<cstdio>`, odnosno `<stdio.h>`, i dalje stoje na raspolaganju i mogu i dalje da se koriste. Klase za ulaz i izlaz koje se nude u jeziku C++, međutim, omogućavaju efikasnije izvođenje ulazno-izlaznih operacija, pa se preporučuje da se koriste one. U svakom slučaju, nikako se ne preporučuje da se u nekom programu istovremeno koriste obe vrste ulazno-izlaznih funkcija.

9.1 Tokovi za datoteke

Datoteke se koriste za trajno uskladištanje podataka na magnetnim diskovima. U širem smislu u datoteke mogu da se ubrajaju i razni ulazno-izlazni uređaji, kao što su tastatura, ekran, štampač itd.

Datoteke na diskovima mogu da se podele u **binarne datoteke** i u **tekstualne datoteke**. Prva vrsta datoteka služi za kompaktno uskladištanje podataka. Druga vrsta datoteka za uskladištanje podataka u obliku čitljivom za čoveka.

9.1.1 Stvaranje tokova za datoteke

```
fstream ();
ofstream ();
ifstream ();
```

Ove metode su podrazumevani konstruktori svojih klasa. Pozivaju se automatski kad god se definiše objekat odgovarajuće klase bez inicijalizatora. Rezultat je stvaranje objekta bez otvaranja datoteke. To znači da ulazno-izlazne operacije, do daljnog, nisu dozvoljene za taj objekat.

```
fstream (const char* imedat, int režim=ios::in|ios::out);
ofstream (const char* imedat, int režim=ios::out);
ifstream (const char* imedat, int režim=ios::in);
```

Ove metode su konstruktori svojih klasa koji se pozivaju automatski kad god se definije objekat za tokove odgovarajućim inicijalizatorom.

Imedat predstavlja naziv datoteke koja treba da se otvari u toku inicijalizacije objekta u navedenom režimu rada. *Režim* rada s datotekom može da se označi jednom od sledećih simboličkih konstanti ili njihovom kombinacijom (pomoću operatora *uključivo ili* po bitovima (!)):

9.1.1 Stvaranje tokova za datoteke

`ios::in`

- Otvaranje datoteke za ulaz. Ako se ne navodi i `ios::out`, datoteka mora da postoji na disku.

`ios::out`

- Otvaranje datoteke za izlaz. Ako se ne navodi i `ios::in`, posle otvaranja datoteke će biti prazna, čak i ako je pre otvaranja već postojala na disku. Tada će stari sadržaj datoteke na disku biti uništen.

`ios::ate`

- Pozicioniranje na kraj datoteke posle otvaranja.

`ios::app`

- Pozicioniranje na kraj datoteke pre svakog upisivanja.

`ios::binary`

- Binarna datoteka. Ako se ništa ne kaže, podrazumeva se rad s tekstualnom datotekom u kojoj prelazak u novi red ('\'n') može da bude predstavljen u nekoj izmenjenoj formi (na primer, kao dva uzastopna bajta CR+LF – carriage return, line feed).

`ios::trunc`

- Uništavanje sadržaja datoteke, ukoliko ista već postoji na disku. Ovo se podrazumeva pri otvaranju samo za izlaz (`ios::out`), izuzev kad se navede i jedan od režima `ios::ate` ili `ios::app`.

Evo primera za stvaranje objekata za tokove na disku:

```
fstream podaci;
ifstream uldat ("podaci.ul"); // Bez otvaranja.
ofstream izldat ("podaci.izl", ios::app); // Za čitanje.
// Za dopisivanje.
```

9.1.2 Otvaranje i zatvaranje datoteke

Otvaramje i zatvaranje datoteke omogućavaju da isti objekat tipa nekog od tokova za datoteke u raznim trenucima predstavlja različite datoteke.

`void open (const char* imedat, int režim);`

Ova metoda otvara datoteku *imedat* u navedenom režimu pridružujući je tekućem objektu (**this*). Tip tekućeg objekta može da bude `fstream`, `ofstream` ili `ifstream`. Mogući režimi opisani su u odeljku 9.1.1.

`void close ();`

Ova metoda zatvara datoteku koja je otvorena u tekućem objektu. Posle zatvaranja datoteke, u istom objektu može da se otvari druga datoteka. Samo zatvaranje datoteke ne uništava objekat, samo ga čini „praznim”.

`bool is_open ();`

Ova metoda ispituje da li u tekućem objektu postoji otvorena datoteka.

Evo primera otvaranja i zatvaranja datoteke u ranije definisanom objektu `podaci` tipa `fstream`:

```
podaci.open ("podat.dat", ios::in | ios::out);
podaci.close ();
```

9.2 Tokovi u memoriji

Glavna upotrebljena vrednost tokova u memoriji su interne konverzije podataka iz binarnog u tekstualni oblik i obrnuto. To omogućava efikasno pisanje programa za obradu teksta kada u tekstu treba ugraditi neku brojčanu vrednost, ili iz teksta izdvajati neku brojčanu vrednost. To je razlog zbog čega se za spoljašnje prikazivanje sadržaja tokova u memoriji koriste objekti tipa `string`. Uprkos tome, sadržaj toka u memoriji ne mora biti tekst. Može da bude i niz binarnih podataka, kao što je to slučaj kod binarnih datoteka.

9.2.1 Stvaranje tokova u memoriji

```
explicit stringstream (int režim=ios::in|ios::out);
explicit ostringstream (int režim=ios::out);
explicit istringstream (int režim=ios::in);
```

Pomoću ovih podrazumevanih konstruktora stvaraju se prazni tokovi u memoriji. Mogući režimi opisani su u odeljku 9.1.

```
explicit stringstream (const string& s,
                      int režim=ios::in|ios::out);
explicit ostringstream(const string& s, int režim=ios::out);
explicit istringstream(const string& s, int režim=ios::in);
```

Pomoću ovih konstruktora stvaraju se tokovi u memoriji čiji je početni sadržaj kopija parametra `s`. Mogući režimi su opisani u odeljku 9.1.

Evo primera za stvaranje objekata za tokove u memoriji:

```
stringstream tekst; // Prazan objekat.
istringstream ultekst ("Dobar dan!"); // Za čitanje.
ostringstream izltekst (ios::out | ios::app); // Za dopisivanje.
```

9.2.2 Pristup sadržaju tokova u memoriji

```
string& str();
```

Vrednost ove metode je trenutni sadržaj toka koji je tekući objekat metode.

```
void str (const string& s);
```

Ova metoda postavlja vrednost kopije parametra `s` kao novi sadržaj toka koji je tekući objekat metode.

Evo primera za postavljanje i dohvatanje sadržaja toka u memoriji:

```
tekst.str ("Danas je divan dan.");
string rez = izltekst.str();
```

9.3 Rad s tekstualnim tokovima

9.3.1 Prenos bez konverzije

```
ostream& put (char znak);
```

Ova metoda smešta `znak` u izlazni tok koji je tekući objekat metode. Vrednost metode je upućivač na tekući izlazni tok. To omogućava uklapanje poziva metode u složenje izraze.

```
ostream& flush ();
```

Kada se jedan ili više znakova pošalju u izlazni tok oni se ne smeštaju uvek odmah u pridruženu datoteku. Znakovi se prvo smeštaju u neki interni bafer, a u samu datoteku se smeštaju povremeno. Na primer, kada se bafer napuni ili kada se datoteka pridružena izlaznom toku zatvara.

U slučaju izlaznog toka cout bafer se prazni uvek i kada se pristupa ulaznom toku cin. Time se obezbeđuje da u momentu čitanja podataka s tastature sví podaci poslati na ekran budu stvarno i prikazani na ekranu. To može da bude važno za korisnika koji sedi pred ekranom i treba da unese podatke.

Pozivom metode `flush()` zahteva se da se bafer izlaznog toka isprazni, smeštanjem svih znakova u njemu u izlaznu datoteku. Ovo je korisno da se uradi kad god se duže vremena ne očekuje novi pristup posmatranom toku. Ako slučajno dođe do nesilnog prekida programa (na primer zbog nestanka napajanja), podaci iz bafera neće biti upisani na disk, čime može da se naruši integritet sadržaja cele datoteke.

Vrednost metode je upućivač na tekući izlazni tok. To omogućava uklapanje poziva metode u složenje izraze.

```
int get();
```

Ova metoda uzima sledeći znak iz tekućeg ulaznog toka.

Vrednost metode je kód dohvaćenog znaka ili EOF ako se stiglo do kraja toka.

```
istream& get (char& znak);
```

Ova metoda uzima sledeći znak iz ulaznog toka i smešta ga u parametar `znak`.

Vrednost metode je upućivač na tekući ulazni tok. To omogućava uklapanje poziva metode u složenje izraze.

```
istream& get (char* niz, int max, char kraj='\n');
```

```
istream& getline (char* niz, int max, char kraj='\n');
```

Ove metode uzimaju najviše `max-1` znakova iz ulaznog toka i smeštaju ih u `niz`. Prenos se završava kada se nađe na završni znak `kraj` ili kada je preneto `max-1` znakova. U `niz`, iza poslednjeg prenetog znaka, uvek se dodaje znak '\0', a znak `kraj` nikada. U slučaju metode `get()` završni znak `kraj` ostaje u ulaznom toku. Biće isporučen kao prvi znak prilikom prvog sledećeg uzimanja znakova iz toka. U slučaju metode `getline()` završni znak `kraj` izbacuje se iz ulaznog toka, ali se ni tada ne stavlja u `niz`.

Vrednost obe metode je upućivač na tekući ulazni tok. To omogućava uklapanje poziva ovih metoda u složenje izraze.

```
int peek();
```

Ova metoda dohvata sledeći znak iz ulaznog toka, s tim da sâm znak ostaje u ulaznom toku. Biće isporučen kao prvi znak prilikom prvog sledećeg uzimanja znakova iz toka. Vrednost metode je kôd sledećeg znaka u ulaznom toku.

```
istream& putback (char znak);
```

Ova metoda vraća *znak* u ulazni tok. Vraćeni znak biće isporučen kao prvi znak prilikom prvog sledećeg uzimanja znakova iz toka.

Vrednost metode je upućivač na tekući ulazni tok. To omogućava uklapanje poziva metode u složenije izraze.

```
istream& ignore (int max=1, int kraj=EOF);
```

Ova metoda preskače najviše *max* znakova u ulaznom toku. Ti znakovi neće nikada da se isporuče korisniku toka. Preskakanje se završava pre vremena ako nađe završni znak *kraj*.

Vrednost metode je upućivač na tekući ulazni tok. To omogućava uklapanje poziva metode u složenije izraze.

Evo primera kojim se, korišćenjem nekih od prethodnih metoda, sadržaj jedne tekstualne datoteke prepisuje u drugu:

```
int zn; while ((zn = uldat.get ()) != EOF) izldat.put (zn);
```

9.3.2 Prenos s konverzijom

Za potrebe ulazne i izlazne konverzije operatori `>>` i `<<` preklopljeni su za sve standardne tipove podataka odgovarajućim operatorskim funkcijama (videti odeljak 2.3.3.1). Za klasne tipove podataka korisnik može dalje da preklapa ove operatore (videti odeljak 4.3).

Na podatke tipova `short`, `int` i `long` primenjuju se ulazno-izlazne konverzije za cele brojeve (kao što je `%d` u jeziku C). Na podatke tipova `float`, `double` i `long double` primenjuju se konverzije za realne brojeve (kao što je `%g` u jeziku C). Podaci tipa `char` prenose se konverzijom koja odgovara konverziji `%c` u jeziku C, s tom razlikom da se prijezika C i prenose se konverzijom koja odgovara konverziji `%s` u jeziku C. Na kraju pokazivači na objekte proizvoljnih tipova (`void*`) prenose se konverzijom koja odgovara konverziji `%p` u jeziku C.

Kod izlaznih konverzija primenjuju se podrazumevane konverzije funkcije `printf()` jezika C. Odstupanje od toga može da se postigne korišćenjem manipulatora koji su objašnjeni u odeljku 2.3.3.2 ili pozivanjem sledećih metoda:

```
long setf (long maska);
long setf (long maska, long vrsta);
```

Ove metode uključuju (postavljaju na 1) sve indikatore načina konverzije za koje odgovarajući bitovi u *maski* imaju vrednost 1. Preostali indikatori zadržaće svoje vrednosti. Kod metode sa dva parametra, *vrsta* označava jednu ili više grupa indikatora koje treba isključiti (postaviti na 0) pre uključivanja indikatora predviđenih *maskom*.

Vrednost obe metode je maska vrednosti indikatora načina konverzije koja je važila pre pozivanja metoda.

9.3.2 Prenos s konverzijom

Maska indikatora načina konverzije može da se sastavlja kombinovanjem (pomoću operatora `|`) sledećih simboličkih konstanti:

- | | |
|------------------------------|---|
| <code>ios::skipws</code> | - preskakanje belih znakova u toku ulaza. |
| <code>ios::left</code> | - ravnjanje podataka uz levu ivicu izlaznog polja. |
| <code>ios::right</code> | - ravnjanje podataka uz desnu ivicu izlaznog polja. |
| <code>ios::internal</code> | - ravnjanje predznaka i/ili oznake brojevnog sistema uz levu ivicu, a cifara broja uz desnu ivicu izlaznog polja. |
| <code>ios::boolalpha</code> | - predstavljanje logičkih podataka u tekstualnom obliku. |
| <code>ios::dec</code> | - korišćenje decimalne izlazne konverzije celih brojeva. |
| <code>ios::oct</code> | - korišćenje oktalne izlazne konverzije celih brojeva. |
| <code>ios::hex</code> | - korišćenje heksadecimalne izlazne konverzije celih brojeva. |
| <code>ios::showbase</code> | - prikazivanje oznake baze brojevnog sistema (0 ili 0x) pri izlazu celih brojeva. |
| <code>ios::showpoint</code> | - obavezno prikazivanje decimalne tačke pri izlazu realnih brojeva. |
| <code>ios::showpos</code> | - obavezno prikazivanje predznaka (čak i +) pri izlazu. |
| <code>ios::uppercase</code> | - korišćenje velikih slova za prikazivanje heksadecimalnih vrednosti i slova E u eksponencijalnom obliku realnih brojeva. |
| <code>ios::scientific</code> | - prikazivanje realnih brojeva u eksponencijalnom obliku. |
| <code>ios::fixed</code> | - prikazivanje realnih brojeva bez eksponenta. |

Za označavanje grupe indikatora za isključivanje pomoću parametra *vrsta* može da se koristi kombinacija (pomoću operatora `|`) sledećih simboličkih konstanti:

- | | |
|-------------------------------|---|
| <code>ios::adjustfield</code> | - isključivanje indikatora <code>left</code> , <code>right</code> i <code>internal</code> . |
| <code>ios::basefield</code> | - isključivanje indikatora <code>dec</code> , <code>oct</code> i <code>hex</code> . |
| <code>ios::floatfield</code> | - isključivanje indikatora <code>scientific</code> i <code>fixed</code> . |

```
long unsetf (long maska);
```

Ova metoda isključuje (postavlja na 0) sve indikatore načina konverzije za koje odgovarajući bitovi u *maski* imaju vrednost 1. Preostali indikatori zadržaće svoje vrednosti. *Maska* se obrazuje na isti način kao i kod metode `setf()`.

Vrednost metode je maska vrednosti indikatora načina konverzije koja je važila pre pozivanja metode.

```
long flags (long maska);
long flags ();
```

Metoda s jednim parametrom uključuje (postavlja na 1) sve indikatore načina konverzije za koje odgovarajući bitovi u *maski* imaju vrednost 1 i isključuje (postavlja na 0) preostale indikatore. *Maska* se obrazuje na isti način kao i kod metode `setf()`.

Vrednost obe metode je maska vrednosti indikatora načina konverzije koja je važila pre pozivanja metode.

```
int width (int širina);
int width ();
```

Metoda s jednim parametrom podešava širinu polja (broja znakova za predstavljanje podataka) na *širina* znakova.

Vrednost obe metode je širina polja koja je važila pre pozivanja metode.

```
char fill (char znak);
char fill ();
```

Metoda s jednim parametrom zadaje *znak* kojim treba popunjavati izlazno polje kad god tekući podatak nije dovoljno dugačak.

Vrednost obe metode je znak za popunjavanje koji se koristio pre pozivanja metode.

```
int precision (int tačnost);
int precision ();
```

Metoda s jednim parametrom podešava broj prikazanih cifara realnih brojeva na *tačnost* značajnih cifara.

Vrednost obe metode je tačnost koja je važila pre pozivanja metode.

Evo primera za podešavanje parametara izlazne konverzije koji će ispisati ***123:

```
int i=123; cout.fill('*'); cout.width(6); cout << i;
```

9.4 Rad s binarnim tokovima △

```
ostream& write (const char* niz, int broj);
```

Ova metoda smešta *broj* bajtova iz *niza* u izlazni tok. Prenose se svi znakovi (bajtovi) i nijedan znak nema posebno značenje (na primer: '\n' ili '\0').

Vrednost metode je upućivač na tekući izlazni tok. To omogućava uklapanje poziva metode u složenije izraze.

```
ostream& flush ();
```

Ova metoda je detaljno objašnjena u odeljku 9.3.1.

```
istream& read (char* niz, int broj);
```

Ova metoda uzima *broj* bajtova iz ulaznog toka u *niz*. U slučaju greške broj prenetih bajtova može da bude manji od traženog broja.

Vrednost metode je upućivač na tekući ulazni tok. To omogućava uklapanje poziva metode u složenije izraze.

```
int gcount ();
```

Vrednost ove metode je broj bajtova koji su uzeti iz ulaznog toka za koji je pozvana u toku poslednjeg pristupa.

Evo primera za uzimanje niza bajtova iz ulaznog toka *podaci* uz proveru uspeha na osnovu stvarno pročitanog broja bajtova:

```
greska = podaci.read (vekt, n).gcount () < n
```

9.5 Pozicioniranje unutar toka (direktan pristup) △

```
istream& seekg (long pozicija);
ostream& seekp (long pozicija);
```

Ove metode vrše pozicioniranje na bajt rednog broja *pozicija* unutar ulaznog (*seekg*) ili izlaznog (*seekp*) toka. Sledеći prenos podataka počće od tog bajta.

Vrednost obe metode je upućivač na tekući ulazni, odnosno izlazni tok. To omogućava uklapanje poziva metoda u složenije izraze.

```
istream& seekg (long pomeraj, seek_dir reper);
ostream& seekp (long pomeraj, seek_dir reper);
```

Ove metode vrše pomeranje pozicije unutar ulaznog (*seekg*) ili izlaznog (*seekp*) toka za *pomeraj* bajtova relativno u odnosu na navedenu *repernu* tačku. Sledеći prenos podataka počće od tako dobijene nove pozicije unutar toka. Reperna tačka može da se naznači jednom od sledećih simboličkih konstanti:

- | | |
|----------|-----------------------------|
| ios::beg | - početak toka. |
| ios::cur | - trenutna pozicija u toku. |
| ios::end | - kraj toka. |

Vrednost obe metode je upućivač na tekući ulazni, odnosno izlazni tok. To omogućava uklapanje poziva metoda u složenije izraze.

```
long tellg ();
long tellp ();
```

Vrednost ovih metoda je trenutna pozicija unutar ulaznog (*tellg*) ili izlaznog (*tellp*) toka kao redni broj bajta u odnosu na početak toka.

Evo primera kojim se u relativnu datoteku sa zapisima dužine *duz* bajtova upiše novi sadržaj u zapis rednog broja *k* (*k*=1,2,...):

```
podaci.seekp ((k-1)*duz).write ((char*)&zapis, duz);
```

9.6 Signalizacija grešaka △

Kada se objekti tipa tokova koriste kao logičke vrednosti logička istina (**true**) označava da je tok u ispravnom stanju, logička neistina (**false**) označava da se desila neka greška u toku pristupanja datom toku.

U nastavku su navedene metode za rad s greškama u toku čitanja i pisanja tokova.

```
bool good ();
bool eof ();
bool fail ();
bool bad ();
```

Vrednost **true** prve metode označava da je tok u ispravnom stanju. Prethodni pristup je bio uspešan. Sledеće čitanje može da bude uspešno. Ako tok nije u ispravnom stanju ulazno-izlazne operacije nemaju efekta.

Vrednost **true** druge metode označava da se stiglo do kraja toka. Prethodni pristup je bio uspešan. Sledеće čitanje neće uspeti (osim ako se trenutna pozicija prethodno ne odmakne od kraja toka).

Vrednost `true` treće metode označava da je prethodni pristup bio uspešan i nijedan bajt nije izgubljen, ali da naredni pristup neće da bude uspešan.

Vrednost `true` poslednje metode označava da je tok u neispravnom stanju. Ne može ništa da se predviđi o uspehu narednih pristupa toku.

```
void exceptions (int maska);
int exceptions ();
```

Da ne bi moralo svaki čas da se ispituje da li je došlo do promene stanja toka, moguće je podesiti da se odredene promene stanja toka prijavljuju izuzetkom tipa `ios::failure`.

Parametar `maska` prve metode određuje promene stanja toka za koje se traži prijavljivanje izuzetka. Vrednost druge metode je maska koja označava koje promene stanja toka se trenutno prijavljuju izuzecima.

Parametar `maska` može da bude kombinacija (pomoću operatora `|`) sledećih simboličkih konstanti:

- | | |
|---------------------------|---|
| <code>ios::eofbit</code> | - izuzetak se prijavljuje kad vrednost metode <code>eof()</code> postane <code>true</code> . |
| <code>ios::failbit</code> | - izuzetak se prijavljuje kad vrednost metode <code>fail()</code> postane <code>true</code> . |
| <code>ios::badbit</code> | - izuzetak se prijavljuje kad vrednost metode <code>bad()</code> postane <code>true</code> . |
| <code>ios::goodbit</code> | - ovom konstantom se isključuje prijavljivanje izuzetaka. Ova konstanta ima vrednost nula i zato ne može da se kombinuje sa gore navedenim konstantama. |

```
void clear (int maska=ios::goodbit);
```

Ova metoda postavlja stanje toka koji je tekući objekat metode na osnovu vrednosti parametra `maska`. Vrednost parametra može da bude `ios::goodbit` ili kombinacija preostala tri gore prikazana bita. Metoda `clear()` će da prijavi izuzetak tipa `ios::failure` ako je to metodom `exceptions()` ranije zatraženo za novopostavljeno stanje toka.

Evo primera kojim se sadržaj jedne tekstualne datoteke prepiše u drugu, uz prekid ako se desi bilo kakva neispravnost (`!good()`) kod bilo kog toka:

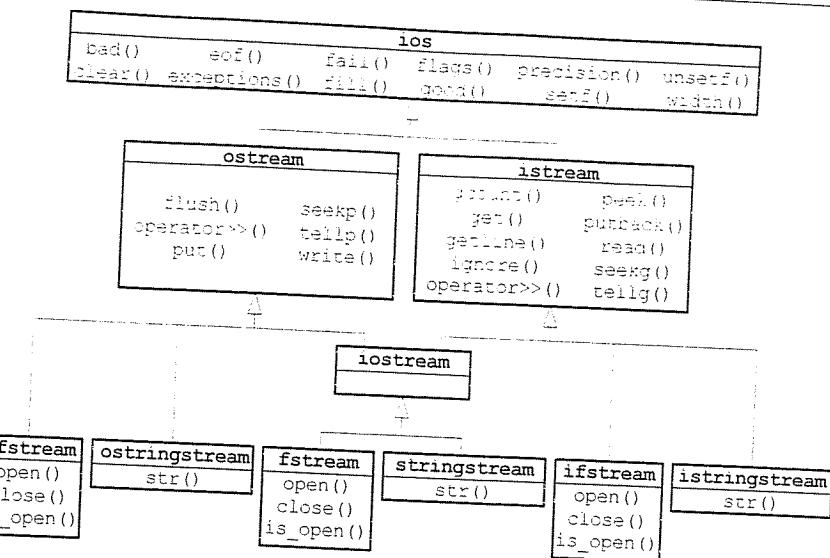
```
char zn; while (izlazat && ulazat.get (zn)) izlazat.put (zn);
```

9.7 Klase za ulaz i izlaz

U prethodnim odeljcima prikazane su najvažnije metode koje bibliotečke klase za ulaz i izlaz u jeziku C++ pružaju programerima. Postoji još izvestan broj metoda koje nisu pomenute. One su potrebne samo za krajnje profesionalne primene.

Sve metode za rad s tokovima razvrstane su u nekoliko klasa. Slika 9.1 prikazuje njihov dijagram klasa. Za svaku klasu prikazane su samo javne metode koje su pomenute u prethodnim odeljcima. Pored toga, izostavljeni su konstruktori i destruktori klasa.

Osnovna klasa celog sistema je klasa `ios` koja sadrži najelementarnije radnje, zajedničke za sve vrste tokova. Ona sadrži i definicije raznih simboličkih konstanti koje su, takođe,



Slika 9.1 – Dijagram klasa za rad s tokovima

pomenute u prethodnim odeljcima. Za njihovo korišćenje ispred identifikatora treba dodati oznaku klase u kojoj su definisane (`ios::`).

Iz klase `ios` neposredno su izvedene dve klase: `ostream` i `istream`. Klasa `ostream` sadrži sve radnje vezane za izlazne tokove, nezavisno od toga da li se tok nalazi na disku ili u memoriji. Slično tome, klasa `istream` objedinjuje radnje vezane za ulazne tokove.

Izvedena klasa `iostream` služi samo za objedinjavanje ulaznih i izlaznih radnji u jednu klasu.

Na poslednjem nivou nalaze se klase koje predstavljaju pojedine vrste tokova: ulazne, izlazne, ulazno-izlazne i to na disku ili u memoriji (`ifstream`, `ofstream`, `fstream`, `istringstream`, `ostringstream` i `stringstream`).

Rad svih tih klasa podržava jedan sistem klasa koji ostvaruje elementarne radnje vezane za rukovanje fizičkim uređajima. Sastoji se od osnovne klase `streambuf` i iz nje izvedenih klasa `filebuf` i `stringbuf`. Njihov zadatak je rukovanje ulazno-izlaznim baferima i neposredan pristup do samih ulazno-izlaznih uređaja.

Ovaj sistem klasa nije ni u kakvom nasledničkom odnosu s gore prikazanim sistemom klasa. Usluge tih klasa dobijaju se uobičajenim pozivanjima njihovih javnih metoda.

9.8 Zadaci △

9.8.1 Ostvarenje relativnih datoteka △

Zadatak:

Napisati na jeziku C++ klasu za obradu relativnih binarnih datoteka sa zapisima fiksne dužine.

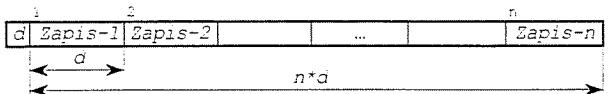
Napisati na jeziku C++ programe za stvaranje uređene linearne liste u relativnoj datoteci i za prikazivanje sadržaja takve liste. Elementi liste treba da se sastoje od celobrojne šifre kao kriterijuma za uređivanje i niza znakova od 50 elemenata.

Rešenje:

Osnovna karakteristika relativnih datoteka je da se zapisima pristupa po proizvoljnom redosledu na osnovu rednog broja zapisa. Dok neki viši programski jezici kao deo jezika nude mogućnost rada s relativnim datotekama (na primer: *FORTRAN*), u jeziku C++ to može da se ostvara definisanjem odgovarajuće klase.

Pored samih zapisova s podacima datoteke mogu da imaju i neko zaglavljivo koje sadrži potrebne parametre datoteke. U slučaju relativnih datoteka sa zapisima fiksne dužine neophodno je da se u zaglavljivu nalazi podatak o dužini pojedinih zapisova. Pored toga, može da se nalazi i podatak o broju zapisova u datoteci. To, međutim, nije neophodno jer može da se izračuna na osnovu veličine datoteke u bajtovima i dužine zapisova.

Na slici 9.2 prikazana je struktura relativnih datoteka za ovo rešenje zadatka. Dužina pojedinih zapisova je a bajtova.



Slika 9.2 – Struktura relativne datoteke

Uobičajene radnje s datotekama su otvaranje, zatvaranje, upisivanje zapisa u datoteku i čitanje zapisa iz datoteke. Te radnje u posmatranom rešenju ostvarene su metodama `open()`, `close()`, `write()` i `read()`. Korišćeni su engleski nazivi da bi izgledalo kao da su te radnje deo samog jezika.

Program 9.1 prikazuje definiciju klase `RFile` za rad s relativnim datotekama. Izvedena je iz klase `fstream` pošto se kod relativnih datoteka u toku jedne obrade obično obavlja kako upisivanje, tako i čitanje zapisova.

Na samom početku klase `RFile` nalazi se definicija unutrašnje klase `Greska` za predstavljanje grešaka koje mogu da se dese u toku rada s relativnim datotekama. Metode klase `RFile` za prijavljivanje grešaka u naredbama `throw` koriste objekte tipa `Greska`.

U početnom, javnom delu klase `Greska` nalazi se definicija nabrajanja `Gre` kojom se uvođe simboličke konstante za šifriranje pojedinih grešaka. Pošto su javne, te konstante mogu da se koriste i izvan klase `RFile`, na primer: `RFile::Greska::DUZ`.

Privatno pojedinačno polje `gre` tipa `Gre` služi za smeštanje šifre greške koju predstavlja dati objekat tipa `Greska`. Privatno zajedničko polje `por` služi za definisanje tekstualnih opisa pojedinih grešaka za potrebe upisivanja u neki izlazni tok.

```
// Definicija klase za relativne datoteke (RFile).

#include <iostream>
using namespace std;

class RFile : fstream {
public:
    class Greska { // Unutrašnja klasa za predstavljanje grešaka.
        public:
            enum Gre { OK, DUZ, BRO, VELI };
            // Šifre grešaka:
            // neispravna dužina zapisa,
            // neispravan redni broj zapisa,
            // datoteka nije ceo broj zapisa,
            // datoteka nije otvorena
            ZAP, OTV, POZ, PIS, CIT, ZAP;
            // nije uspelo otvaranje,
            // nije uspelo pozicioniranje,
            // nije uspelo upisivanje,
            // nije uspelo čitanje,
            // zapis ne postoji.
    };
    private:
        Gre gre; // Šifra greške.
        static const char* por[]; // Tekstualni opisi grešaka.
    public:
        Greska (Gre g) { gre = g; }
        friend ostream& operator<< (ostream& it, const Greska& g);
    };
private:
    long duz_zap, bro_zap, tek_zap, tek_poz; // bro_zap == -1 - nije otvorena.
    void otvori (const char* ime, long duz=0); // Otvaranje datoteke (duz>0
                                                // -> stvaranje nove).
public:
    RFile () : fstream () { bro_zap = -1; } // Inic. bez otvaranja datoteke.
    RFile (const char* ime, long duz=0) // Inic. sa otvaranjem datoteke.
        : fstream () { otvori (ime, duz); } // Uništavanje objekta.
    ~RFile () { close (); } // Zatvaranje datoteke.
    RFile& open (const char* ime, long duz=0) // Otvaranje datoteke.
        { otvori (ime, duz); return *this; }
    int open () { return bro_zap >= 0; } // Da li je datoteka otvorena?
    RFile& write (const void* niz, long zap=0); // Pisanje zap (zap==0 - sekvoj).
    RFile& read (void* niz, long zap=0); // Čitanje zap (zap==0 - sekvoj).
    RFile& close (); // Zatvaranje datoteke.
    long recn () {return bro_zap >= 0 ? -1 : bro_zap;} // Broj zapisova u datoteci.
    long recl () {return bro_zap >= 0 ? -1 : duz_zap;} // Dužina zapisova.
};
```

Program 9.1 – Definicija klase relativnih datoteka (rfile.h)

Od funkcija postoji javan konstruktor kojim se objekat tipa `Greska` inicijalizuje odgovarajućom šifrom greške i prijateljska operatorska funkcija za upisivanje tekstualnog opisa greške u neki izlazni tok.

Klasa `Greska` je pravljena kao unutrašnja klasa ne bi li se omogućilo da se i uz druge klase prave slične klase za greške, bez problema s odabiranjem imena klase. Izvan klase `RFile`, posmatrana klasa `Greska` mora da se obeležava sa `RFile::Greska`. Ako se uz

Klase takođe napravi klasu Greska, ona će morati da se obeležava sa Klase::Greska, što obezbeđuje jednoznačnost označavanja istoimenih klase Greska.

U klasi Rfile postoje četiri privatna polja: duz_zap predstavlja dužinu zapisa u bajtovima, bro_zap broj zapisa u datoteci, tek_zap redni broj tekućeg zapisa (1, 2, ...) i tek_poz poziciju tekućeg zapisa u datoteci u bajtovima (0, 1, ...). Usvojeno je da se „prazni“ objekti tipa Rfile, u kojima nisu otvorene datoteke, obeležavaju sa bro_zap=-1.

Privatna pomoćna metoda otvori() predviđena je za otvaranje datoteke. Nju pozivaju konstruktor Rfile() pri inicijalizaciji primeraka klase s otvaranjem datoteke i metoda open() za otvaranje datoteke u datom primerku kasnije u toku izvršavanja programa. Obavezni parametar metode otvori() je naziv datoteke koja se otvara. Neobaveznji parametar duz treba da se navodi samo pri otvaranju datoteke koja još ne postoji na disku. Predstavlja željenu dužinu zapisa u novoj datoteci. Ako se ne navodi duz, podrazumevana vrednost 0 označava da treba otvoriti datoteku koja već postoji na disku. Dužina zapisa uzimaće se iz zaglavlja datoteke.

Od metoda na prvom mestu su konstruktori i destruktur. Prvi konstruktor stvara „prazan“ objekat bez otvaranja datoteke. Drugi konstruktor otvara datoteku određenog imena. Zadatak destruktora je da zatvori datoteku koja je u momentu uništavanja objekta otvorena u njemu.

Osnovne radnje s datotekama ostvaruju metode open(), close(), write() i read(). Vrednosti ovih metoda su upućivači na tekući objekat, čime je omogućeno lančanje poziva tih metoda u jednom izrazu.

Pored već pomenutih javnih metoda postoje još tri uslužne metode. Metoda open() bez parametara utvrđuje da li je u datom primerku klase Rfile otvorena neka datoteka. Metoda recn() daje broj zapisa, a metoda recl() dužinu zapisa u datoteci koja je otvorena u datom objektu. Obe metode daju vrednost -1 ukoliko u datom objektu trenutno nije otvorena datoteka.

Programi 9.2, 9.3 i 9.4 prikazuju definicije metoda i zajedničkih polja uz klasu Rfile. Prvi od njih, program 9.2, odnosi se na unutrašnju klasu Greska.

```
// Definicije metoda i zajedničkih polja uz klasu Rfile.

#include "rfile.h"

const char Rfile::Greska::por[] = { "", // PORUKE O GREŠKAMA:
    "Neispravna duzina zapisa", // DUZ
    "Neispravan redni broj zapisa", // BRO
    "Datoteka nije ceo broj zapisa", // VEL
    "Datoteka nije otvorena", // ZAT
    "Nije uspelo otvaranje", // OTV
    "Nije uspelo pozicioniranje", // POZ
    "Nije uspelo upisivanje", // PIS
    "Nije uspelo citanje", // CIT
    "Zapis ne postoji" // ZAP
};

ostream& operator<< (ostream& it, const Rfile::Greska& g)
{
    return it << "*** " << Rfile::Greska::por[g.gre] << "! ***\n\a"; }
}
```

Program 9.2 - Definicije metoda u klasu Rfile (rfile.C, prvi deo)

9.3.1 Ostvarenje relativnih datoteka

Zajedničko polje por sadrži tekstove poruka o greškama. Skreće se pažnja na oznaku klase kojoj on pripada. Pošto se radi o unutrašnjoj klasi, treba da se navede i identifikator splošnje klase, tj. Rfile::Greska.

Operatorska funkcija operator<<() upisuje tekst poruke, koja odgovara sadržaju vrednosti parametra g, u izlazni tok it, koji je prvi parametar. Parametar g je objekat tipa Greska. Posto je ta klasa unutrašnja u klasi Rfile, za tip tog operanda mora da se piše Rfile::Greska. Šifra greške koju predstavlja taj operand (g.gre) u telu funkcije koristi se kao indeks za izbor odgovarajućeg teksta iz zajedničkog niza (por[]). Pored navedenog načina Rfile::Greska::por[], koji jasno daje do znanja da se koristi zajedničko polje načina Rfile::Greska::por[], pošto je g objekat tipa te klase. Sama operatorska funkcija operator<<() je prijateljska funkcija, a ne metoda klase, pa se ispred njenog imena ne piše oznaka klase za koju se pravi.

Program 9.3 sadrži definicije metoda klase Rfile za otvaranje i zatvaranje datoteke.

```
void Rfile::otvori (const char* ime, long duz) { // OTVARANJE DATOTEKE.
    if (duz < 0) throw Greska (Greska::DUZ); // Dator. otvorena zatvori.
    if (bro_zap >= 0) close(); // Stvaranje nove datoteke.
    if (duz) {
        fstream::open (ime, ios::in);
        if (!is_open ()) throw Greska (Greska::OTV);
        fstream::open (ime, ios::in | ios::out | ios::binary);
        if (!is_open ()) throw Greska (Greska::OTV);
        if (!fstream::write ((char*)&duz, sizeof (long)) ||
            !flush()) throw Greska (Greska::PIS);
        duz_zap = duz;
        bro_zap = 0;
    } else {
        fstream::open (ime, ios::in);
        if (!is_open ()) throw Greska (Greska::OTV);
        fstream::close ();
        fstream::open (ime, ios::in | ios::out | ios::binary);
        if (!is_open ()) throw Greska (Greska::OTV);
        if (!fstream::read ((char*)&duz_zap, sizeof (long)))
            throw Greska (Greska::CIT);
        if (!seekg (0L, ios::end)) throw Greska (Greska::POZ);
        long k = tellg () - (long)sizeof (long);
        if (k < duz_zap) throw Greska (Greska::VEL);
        bro_zap = k / duz_zap;
    }
    tek_zap = 0;
    tek_poz = sizeof (long);
}

Rfile& Rfile::close () { // ZATVARANJE DATOTEKE.
    fstream::close ();
    bro_zap = -1;
    return *this;
}
```

Program 9.3 - Definicije metoda u klasu Rfile (rfile.C, drugi deo)

9.8 Zadaci △

9.8.1 Ostvarenje relativnih datoteka △

Zadatak:

Napisati na jeziku C++ klasu za obradu relativnih binarnih datoteka sa zapisima fiksne dužine.

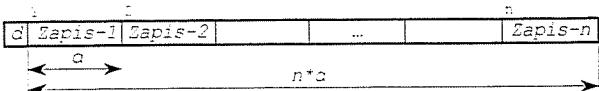
Napisati na jeziku C++ programe za stvaranje uredene linearne liste u relativnoj datoteci i za prikazivanje sadržaja takve liste. Elementi liste treba da se sastoje od celobrojne šifre kao kriterijuma za uređivanje i niza znakova od 50 elemenata.

Rešenje:

Osnovna karakteristika relativnih datoteka je da se zapisima pristupa po proizvoljnom redosledu na osnovu rednog broja zapisa. Dok neki viši programski jezici kao deo jezika nude mogućnost rada s relativnim datotekama (na primer: FORTRAN), u jeziku C++ to može da se ostvaruje definisanjem odgovarajuće klase.

Pored samih zapisu s podacima datoteke mogu da imaju i neko zaglavje koje sadrži potrebne parametre datoteke. U slučaju relativnih datoteka sa zapisima fiksne dužine neophodno je da se u zaglavju nalazi podatak o dužini pojedinih zapisu. Pored toga, može da se nalazi i podatak o broju zapisu u datoteci. To, međutim, nije neophodno jer može da se izračuna na osnovu veličine datoteke u bajtovima i dužine zapisu.

Na slici 9.2 prikazana je struktura relativnih datoteka za ovo rešenje zadatka. Dužina pojedinih zapisu je a bajtova.



Slika 9.2 – Struktura relativne datoteke

Uobičajene radnje s datotekama su otvaranje, zatvaranje, upisivanje zapisa u datoteku i čitanje zapisa iz datoteke. Te radnje u posmatranom rešenju ostvarene su metodama `open()`, `close()`, `write()` i `read()`. Korišćeni su engleski nazivi da bi izgledalo kao da su te radnje deo samog jezika.

Program 9.1 prikazuje definiciju klase `RFile` za rad s relativnim datotekama. Izvedena je iz klase `fstream` pošto se kod relativnih datoteka u toku jedne obrade obično obavlja kako upisivanje, tako i čitanje zapisu.

Na samom početku klase `RFile` nalazi se definicija unutrašnje klase `Greska` za predstavljanje grešaka koje mogu da se dese u toku rada s relativnim datotekama. Metode klase `RFile` za prijavljivanje grešaka u naredbama `throw` koriste objekte tipa `Greska`.

U početnom, javnom delu klase `Greska` nalazi se definicija nabranja `Gre` kojom se uvođe simboličke konstante za šifriranje pojedinih grešaka. Pošto su javne, te konstante mogu da se koriste i izvan klase `RFile`, na primer: `RFile::Greska::DUZ`.

Privatno pojedinačno polje `gre` tipa `Gre` služi za smeštanje šifre greške koju predstavlja dati objekat tipa `Greska`. Privatno zajedničko polje `por` služi za definisanje tekstualnih opisa pojedinih grešaka za potrebe upisivanja u neki izlazni tok.

9.8.1 Ostvarenje relativnih datoteka

```
// Definicija klase za relativne datoteke (RFile).

#include <fstream>
using namespace std;

class RFile : fstream {
public:
    class Greska { // Unutrašnja klasa za predstavljanje grešaka.
        public:
            enum Gre { OK,           // Šifre grešaka:
                      DUZ,          // neispravna dužina zapisa,
                      BRO,          // neispravan redni broj zapisa,
                      VEL,          // datoteka nije ceo broj zapisa,
                      VELI,         // 
                      ZAT,          // datoteka nije otvorena,
                      OTV,          // nije uspelo otvaranje,
                      POZ,          // nije uspelo pozicioniranje,
                      PIS,          // nije uspelo upisivanje,
                      CIT,          // nije uspelo čitanje,
                      ZAP };        // zapis ne postoji.

        private:
            Gre gre;           // Šifra greške.
            static const char* por[]; // Tekstualni opisi grešaka.
    };
    public:
        Greska (Gre g) { gre = g; }
        friend ostream& operator<< (ostream& it, const Greska& g);
    };

private:
    long duz_zap, bro_zap, tek_zap, tek_poz; // bro_zap== -1 → nije otvorena.
    void otvori (const char* ime, long duz=0); // Otvaranje datoteke (duz>0
                                                // → stvaranje nove).

public:
    RFile () : fstream () { bro_zap = -1; } // Inic. bez otvaranja datoteke.
    RFile (const char* ime, long duz=0) // Inic. sa otvaranjem datoteke.
        : fstream () { otvori (ime, duz); }
    ~RFile () { close (); } // Uništavanje objekta.
    RFile& open (const char* ime, long duz=0) // Otvaranje datoteke.
        { otvori (ime, duz); return *this; }
    int open () { return bro_zap >= 0; } // Da li je datoteka otvorena?
    RFile& write (const void* niz, long zap=0); // Pisanje zap(zap==0 - sekvi).
    RFile& read (void* niz, long zap=0); // Čitanje zap(zap==0 - sekvi).
    RFile& close ();
    long recn () {return bro_zap < 0 ? -1 : bro_zap;} // Broj zapisu u datoteci.
    long recl () {return bro_zap < 0 ? -1 : duz_zap;} // Dužina zapisu.
};


```

Program 9.1 – Definicija klase relativnih datoteka (rfile.h)

Od funkcija postoji javan konstruktor kojim se objekat tipa `Greska` inicijalizuje odgovarajućom šifrom greške i prijateljska operatorska funkcija za upisivanje tekstualnog opisa greške u neki izlazni tok.

Klase `Greska` je pravljena kao unutrašnja klasa ne bi li se omogućilo da se i uz druge klase prave slične klase za greške, bez problema s odabiranjem imena klase. Izvan klase `RFile`, posmatrana klasa `Greska` mora da se obeležava sa `RFile::Greska`. Ako se uz

klasu Klasa takođe napravi klasa Greska, ona će morati da se obeležava sa Klasa::Greska, što obezbeđuje jednoznačnost označavanja istoimenih klasa Greska.

U klasi Rfile postoje četiri privatna polja: duz_zap predstavlja dužinu zapisa u bajtovima, bro_zap broj zapisa u datoteci, tek_zap redni broj tekućeg zapisa (1, 2, ...), i zni poz poziciju tekućeg zapisa u datoteci u bajtovima (0, 1, ...). Usvojeno je da se „pravni“ objekti tipa Rfile, u kojima nisu otvorene datoteke, obeležavaju sa cro_zap=-1.

Privatna pomoćna metoda otvori() predviđena je za otvaranje datoteke. Nju pozivaju konstruktor Rfile() pri inicijalizaciji primeraka klase s otvaranjem datoteke i metoda open() za otvaranje datoteke u datom primerku kasnije u toku izvršavanja programa. Obavezni parametar metode otvori() je naziv datoteke koja se otvara. Neobavezni parametar duz treba da se navodi samo pri otvaranju datoteke koja još ne postoji na disku. Predstavlja željenu dužinu zapisa u novoj datoteci. Ako se ne navodi duz, podrazumevana vrednost 0 označava da treba otvoriti datoteku koja već postoji na disku. Dužina zapisa uzimaće se iz zaglavljiva datoteke.

Od metoda na prvom mestu su konstruktori i destruktur. Prvi konstruktor stvara „pravni“ objekat bez otvaranja datoteke. Drugi konstruktor otvara datoteku određenog imena. Zadatak destruktora je da zatvori datoteku koja je u momentu uništavanja objekta otvorena u njemu.

Osnovne radnje s datotekama ostvaruju metode open(), close(), write() i read(). Vrednosti ovih metoda su upućivači na tekući objekat, čime je omogućeno lančanje poziva tih metoda u jednom izrazu.

Pored već pomenutih javnih metoda postoje još tri uslužne metode. Metoda open() bez parametara utvrđuje da li je u datom primerku klase Rfile otvorena neka datoteka. Metoda recn() daje broj zapisa, a metoda recl() dužinu zapisa u datoteci koja je otvorena u datom objektu. Obe metode daju vrednost -1 ukoliko u datom objektu trenutno nije otvorena datoteka.

Programi 9.2, 9.3 i 9.4 prikazuju definicije metoda i zajedničkih polja uz klasu Rfile. Prvi od njih, program 9.2, odnosi se na unutrašnju klasu Greska.

```
// Definicije metoda i zajedničkih polja uz klasu Rfile.

#include "rfile.h"

const char* Rfile::Greska::por[] = { "", // PORUKE O GREŠKAMA:
    "Neispravna duzina zapisa", // DUZ
    "Neispravan redni broj zapisa", // BRO
    "Datoteka nije ceo broj zapisa", // VEL
    "Datoteka nije otvorena", // ZAT
    "Nije uspelo otvaranje", // OTV
    "Nije uspelo pozicioniranje", // POZ
    "Nije uspelo upisivanje", // PIS
    "Nije uspelo citanje", // CIT
    "Zapis ne postoji" // ZAP
};

ostream& operator<< (ostream& it, const Rfile::Greska& g)
{
    return it << "*** " << Rfile::Greska::por[g.gre] << "! ***\n\a"; }
}
```

Program 9.2 – Definicije metoda uz klasu Rfile (rfile.C, prvi deo)

9.8.1 Ostvarenje relativnih datoteka

Zajedničko polje por sadrži tekstove poruka o greškama. Skreće se pažnja na oznaku klase kojoj on pripada. Pošto se radi o unutrašnjoj klasi, treba da se navede i identifikator spoljašnje klase, tj. Rfile::Greska.

Operatorska funkcija operator<<() upisuje tekst poruke, koja odgovara sadržaju vrednosti parametra g, u izlazni tok it, koji je prvi parametar. Parametar g je objekat tipa Rfile::Greska. Posto je ta klasa unutrašnja u klasi Rfile, za tip tog operanda mora da se piše Rfile::Greska. Šifra greške koju predstavlja taj operand (g.gre) u telu funkcije koristi se kao indeks za izbor odgovarajućeg teksta iz zajedničkog niza (por[]). Pored navedenog načina Rfile::Greska::por[], koji jasno daje do znanja da se koristi zajedničko polje klase, u posmatranom slučaju moglo bi da se piše i g.por[], pošto je g objekat tipa te klase. Sama operatorska funkcija operator<<() je prijateljska funkcija, a ne metoda klase, pa se ispred njenog imena ne piše oznaka klase za koju se pravi.

Program 9.3 sadrži definicije metoda klase Rfile za otvaranje i zatvaranje datoteke.

```
void Rfile::otvori (const char* ime, long duz) { // OTVARANJE DATOTEKE.
    if (duz < 0) throw Greska (Greska::DUZ);
    if (bro_zap >= 0) close (); // Datot. otvorena zatvori.
    if (duz) {
        fstream::open (ime, ios::in);
        if (is_open ()) throw Greska (Greska::OTV);
        fstream::open (ime, ios::in | ios::out | ios::binary);
        if (! is_open ()) throw Greska (Greska::OTV);
        if (! fstream::write ((char*)&duz, sizeof (long)) ||
            ! flush()) throw Greska (Greska::PIS);
        duz_zap = duz;
        bro_zap = 0;
    } else {
        fstream::open (ime, ios::in);
        if (! is_open ()) throw Greska (Greska::OTV);
        fstream::close ();
        fstream::open (ime, ios::in | ios::out | ios::binary);
        if (! is_open ()) throw Greska (Greska::OTV);
        if (! fstream::read ((char*)&duz_zap, sizeof (long)))
            throw Greska (Greska::CIT);
        if (! seekg (0L, ios::end)) throw Greska (Greska::POZ);
        long k = tellg () - (long)sizeof (long);
        if (k < duz_zap) throw Greska (Greska::VEL);
        bro_zap = k / duz_zap;
    }
    tek_zap = 0;
    tek_poz = sizeof (long);
}

Rfile& Rfile::close () {
    fstream::close ();
    bro_zap = -1;
    return *this;
} // ZATVARANJE DATOTEKE.
```

Program 9.3 – Definicije metoda uz klasu Rfile (rfile.C, drugi deo)

U metodi `otvori()`, pre svega, proverava se da li je navedena dužina zapisa ispravna. Negativne vrednosti nisu dozvoljene. Ako se to desi, prekida se dalji rad prijavljivanjem greške operatom `throw`, čiji je operand objekat dobijen konstruktorom `Greska()`. Parametar konstruktora je šifra greške DUZ. Pošto je ta simbolička konstanta definisana unutar klase `Greska` (a ne klase `RFile`), mora da se navodi kao `Greska::DUZ`.

Posle toga, ako je tekućem objektu pridružena neka datoteka ona se zatvori pozivanjem metode `close()`.

Pri stvaranju nove datoteke greška je ako na disku već postoji datoteka zadatog imena. Jedini način da se to proveri je da se pokuša s otvaranjem datoteke navedenog imena za ulaz (`ios::in`). Ako otvaranje uspe datoteka već postoji. Pošto je u izvedenoj klasi `RFile` definisana metoda `open()`, da bi se koristila istoimena metoda iz osnovne klase `fstream`, mora da se navede i ime osnovne klase (`fstream::open()`). Uspeh ovog probnog otvaranja ispituje se pozivanjem metode `is_open()`. U ovom slučaju, uspeh otvaranja smatra se greškom i tada se metoda `otvori()` završava prijavljivanjem greške sa šifrom OTV.

Kada se utvrdi da datoteka navedenog imena još ne postoji potrebno je stvoriti novu praznu datoteku. Pošto se pri radu s relativnim datotekama obično vrše i pisanja i čitanja podataka naizmenično, datoteku treba otvoriti i za čitanje i za pisanje (`ios::in|ios::out`). Pri takvom otvaranju, ako datoteka još ne postoji, stvara se ova prazna datoteka. Ako datoteka već postoji, njen zatečeni sadržaj se očuva. Pošto se ovde radi o binarnoj datoteci, to se moralo naznačiti dodavanjem i konstante `ios::binary` drugom argumentu metode `open()`. Sada je greška ako otvaranje datoteke nije uspelo. To može da se desi zato što, na primer, navedeno ime datoteke nije ispravno.

Posle uspešnog otvaranja treba obrazovati zaglavje datoteke upisujući u nju dužinu zapisa. Uspeh te radnje ispituje se pomoću vrednosti metoda `fstream::write()` i `flush()`. Vrednosti tih metoda su upućivači na tokove čije vrednosti kao logički podaci ukazuju na uspeh. U slučaju neuspeha prijavljuje se greška sa šifrom PIS.

Ako do ovog momenta nije otkrivena nijedna greška, otvaranje nove datoteke je uspešno i mogu da se popune polja tekućeg objekta.

U slučaju otvaranja postojeće (stare) datoteke greška je ako datoteka navedenog imena ne postoji na disku. To može da se utvrdi samo tako da se datoteka pokuša otvoriti samo za ulaz (naime, ako se otvori i za ulaz i za izlaz datoteka će se stvoriti ako je nema). Ako otvaranje uspe datoteka postoji. Pošto na taj način otvorena datoteka ne odgovara za nastavak rada s datotekom ona se zatvori i ponovo otvori, ali sada i za ulaz i za izlaz. Takođe, sada je već potrebno naznačiti i da se radi o binarnoj datoteci (pri probnom otvaranju to nije bilo neophodno).

Posle uspešnog otvaranja dužina zapisa se čita iz zaglavja datoteke, a broj zapisa se izračunava na osnovu veličine cele datoteke. Greška je ako veličina datoteke umanjena za veličinu zaglavja nije deljiva s dužinom zapisa. I u ovom slučaju polju `bro_zap` vrednost se dodeljuje tek kada je sigurno da će metoda `otvori()` biti uspešno završena. Ovo je važno, jer vrednost člana `bro_zap` se koristi za utvrđivanje da li je u datom objektu otvorena neka datoteka ili ne.

Bez obzira da li je otvorena stara ili nova datoteka, posle otvaranja nalazi se na početku datoteke. Na to ukazuju vrednosti dodeljene preostalim poljima `tek_zap` i `tek_poz`.

Zatvaranje datoteke je mnogo jednostavnije od otvaranja. Pozivom metode `fstream::close()` uradi se sve što treba za bilo kakvu datoteku u jeziku C++. Jedina specifična

radnja za ovde posmatrane relativne datoteke je stavljanje vrednosti -1 u polje `bro_zap` kao oznaka da datom objektu više nije pridružena datoteka.

Program 9.4 sadrži definicije preostale dve metode klase `RFile` za pisanje i čitanje po jednog zapisa.

```
RFile& RFile::write (const void* niz, long zap) { // PISANJE ZAPISA.
    if (bro_zap == -1) throw Greska (Greska::ZAT);
    if (zap < 0)         throw Greska (Greska::BRO);
    if (zap) {
        tek_poz = (zap - 1) * duz_zap + sizeof (long);
        tek_zap = zap;
    } else {
        tek_poz += duz_zap;
        tek_zap++;
    }
    if (tek_zap > bro_zap) bro_zap = tek_zap;
    if (! seekp (tek_poz)) throw Greska (Greska::POZ);
    if (! fstream::write ((char*)niz, duz_zap) ||
        ! flush ())           throw Greska (Greska::PIS);
    return *this;
}

RFile& RFile::read (void* niz, long zap) { // ČITANJE ZAPISA.
    if (bro_zap == -1) throw Greska (Greska::ZAT);
    if (zap < 0)         throw Greska (Greska::BRO);
    if (zap) {
        tek_poz = (zap - 1) * duz_zap + sizeof (long);
        tek_zap = zap;
    } else {
        tek_poz += duz_zap;
        tek_zap++;
    }
    if (zap > bro_zap) throw Greska (Greska::ZAP);
    if (! seekg (tek_poz)) throw Greska (Greska::POZ);
    if (! fstream::read ((char*)niz, duz_zap)) throw Greska (Greska::CIT);
    return *this;
}
```

Program 9.4 – Definicije metoda uz klasu `RFile` (rfile.C, treći deo)

Metode za pisanje (`write()`) i čitanje (`read()`) međusobno su vrlo slične. Parametri su im pokazivač na niz bajtova u memoriji čiji sadržaj treba preneti i redni broj zapisa u datoteci u koji treba pisati ili iz kojeg treba čitati.

Za pokazivač je korišćen generički pokazivač (`void*`) da bi argument prilikom pozivanja mogao da bude bilo kog pokazivačkog tipa. Za razliku od toga, kod metoda `fstream::write()` i `fstream::read()` pokazivač na područje u memoriji je deklarisan da je tipa `char*`. Zbog toga ako je argument nekog drugog pokazivačkog tipa, mora da se koristi eksplicitna konverzija tipa pomoću operatora (`char*`).

Za čitanje i pisanje zapisa greška je ako u datom objektu nije otvorena datoteka ili ako je redni broj traženog zapisa negativan. Podrazumevana vrednost 0 za redni broj zapisa predviđena je za pristup do prvog narednog zapisa u odnosu na prethodno korišćeni zapis (sekvenčni pristup). Prilikom čitanja je greška ako je (zadati ili izračunati) redni broj

traženog zapisa veći od broja zapisa u datoteci. Prilikom pisanja u takvim slučajevima doći će do povećavanja broja zapisa u datoteci. Sam prenos se izvodi uz prethodno pozicioniranje na početak traženog zapisa u datoteci.

Obrazovanje linearnih listi, i drugih lančanih struktura, u datotekama starijeg je datuma od obrazovanja listi u operativnoj memoriji. Koristi se za predstavljanje uređenih nizova podataka kod kojih se logički redosled i fizički redosled razlikuju.

Kao i u memoriji, i u datotekama svaki element liste, pored korisnog sadržaja, treba da sadrži i pokazivač na naredni element u listi. Samo što se elementi smještaju kao pojedinačni zapisi u datotekama, a pokazivači su redni brojevi zapisa, dakle, celi brojevi. Pošto se zapisi numerišu brojevima 1, 2, ..., za označavanje kraja liste obično se koristi vrednost 0.

Na slici 9.3 prikazan je primer smještanja niza traženih podataka u relativnu datoteku u obliku linearne liste.

	sif	txt	nar
1	0	četvrtak	6
2	5	petač	5
3	2	utorak	7
4	7	nedelja	0
5	6	subota	4
6	1	ponedeljak	3
7	3	sreda	8
8	4	cetvrtak	2

Slika 9.3 – Predstavljanje linearne liste u relativnoj datoteci

Svaka vrsta na slici predstavlja jedan zapis. Novi zapisi se uvek dodaju na kraj datoteke. Na taj način fizički redosled tih zapisa ne mora da se poklapa sa željenim logičkim redosledom, po rastućim vrednostima šifara (sif). Taj redosled se uspostavlja odgovarajućim vrednostima pokazivača (nar).

Pošto prvi zapis (s najmanjom šifrom) može da bude bilo gde u datoteci, njegovo mesto mora da se zabeleži na nekom fiksnom mestu u datoteci. Na slici 9.3 za tu namenu rezervisan je zapis rednog broja 1. Taj zapis ne sadrži koristan podatak, već samo pokazivač na zapis koji sadrži prvi element liste. Naziva se i **glava liste**.

Program 9.5 obrazuje opisanu uređenu listu u novoj relativnoj binarnoj datoteci.

Pošto zaglavljje `rfile.h` u sebi sadrži direktivu preprocesora `#include <iostream>` ona nije stavljena u posmatrani program, mada koristi operatore `>>` i `<<` za ulaz i izlaz podataka standardnih tipova. Ovo nije stil koji se preporučuje, jer je posmatrani program na taj način zavisan od nekih tehničkih detalja u klasi koju koristi.

Za opis zapisa u datoteci korišćena je obična struktura (`struct`).

Početna prazna datoteka fiksnog imena (`Lista.dat`) stvara se i otvara kao posledica inicijalizacije objekta `dat` tipa `RFile`. Uspešnost inicijalizacije posebno se ne proverava pošto će se u slučaju neuspeha prijavljivanjem greške u metodi `otvor()` (koju poziva konstruktor `RFile()`) prekinuti niz naredbi i skočiće se na rukovaoca izuzecima, koji se nalazi na kraju naredbe `try`. Pošto ta naredba obuhvata celokupan koristan sadržaj posmatranog programa, nigde u programu posebno se ne pita da li je neka operacija bila uspešna ili ne. Bilo gde da se desi greška posle prekida pozivaće se isti rukovalac izuzecima.

U slučaju uspešne inicijalizacije toka `dat`, u objektu `zap` tipa `Zapis` sastavi se početni sadržaj liste koji obeležava praznu listu. Taj zapis upisuje se na prvo mesto u datoteci.

9.8.1 Ostvarenje relativnih datoteka

```
// Obrazovanje uređene liste u relativnoj datoteci.

#include "rfile.h"
#include <iostream>
#include <iomanip>
using namespace std;

struct Zapis { int sif; char txt[50]; unsigned nar; };

int main () {
    try {
        RFile dat ("Lista.dat", sizeof (Zapis));
        Zapis zap; zap.sif = zap.txt[0] = zap.nar = 0; // Glava liste.
        dat.write (&zap, 1);
        ifstream pod ("Lista.pod"); // Tekstualna datoteka s podacima.
        while (pod >> zap.sif >> zap.txt) {
            cout << setw(5) << zap.sif << " " << zap.txt << endl;
            unsigned nov = dat.recn () + 1; dat.write (&zap, nov);
            int sif = zap.sif;
            dat.read (&zap, 1); unsigned pre=1, tek=zap.nar;
            while (tek && (dat.read (&zap, tek) , zap.sif<sif))
                if pre == tek, tek = zap.nar; else
                    dat.read (&zap, pre); zap.nar = nov; dat.write (&zap, pre);
                    dat.read (&zap, nov); zap.nar = tek; dat.write (&zap, nov);
        }
    } catch (RFile::Greska g) { cout << g; }
    return 0;
}
```

Program 9.5 – Obrazovanje uređene liste u relativnoj datoteci (`lispravi.C`)

Posle toga, inicijalizuje se ulazni tok pod tipa `ifstream`. U njemu se otvori tekstualna datoteka `Lista.pod` iz koje će da se čitaju podaci za umetanje u listu. Mogla bi da se proveri uspešnost otvaranja za ovu datoteku pošto se u klasi `ifstream` (kao ni u ostalim bibliotečkim klasama za rad s tokovima) izuzeci za prijavljivanje konfliktnih situacija ne koriste automatski. Međutim, ako otvaranje nije uspelo, neće uspeti ni čitanje podatka koji se traži izrazom u uslovu ciklusa s izlazom na vrhu. U tom slučaju sadržaj ciklusa neće ni jednom da se izvrši.

Sve dok se ne dode do kraja ulazne datoteke pod pročitani podaci za sledeći element liste prikazuju se na glavnem izlazu i upisuju se iza poslednjeg postojećeg zapisa u datoteci. Redni broj tog zapisa je dat izrazom `dat.recn() + 1`.

Preostaje još da se novi element (zapis) uključi na odgovarajuće mesto u uređenoj listi. U tom cilju prvo se pročita glava liste (zapis rednog broja 1), pa se zatim kreće duž liste, tako što se uzimaju sukcesivni elementi liste. Za razliku od listi u memoriji sada uzimanje sledećeg elementa liste podrazumeva čitanje jednog zapisa iz datoteke.

U toku pretraživanja liste celobrojna promenljiva `tek` sadrži redni broj tekućeg zapisa, a pre prethodnog. Mesto novog elementa je pronađeno ako se stiglo do kraja liste (`tek==0`) ili ako šifra u tekućem zapisu postane veća ili jednaka šifri novog elementa liste. Tada je potrebno u polje `nar` zapisa rednog broja pre upisati redni broj novog zapisa, a u novi zapis redni broj tekućeg zapisa `tek`. Menjanje sadržaja dela zapisa podrazumeva učitavanje celog zapisa u memoriju, promenu sadržaja potrebnih polja i upisivanje izmenjenog zapisa na svoje staro mesto.

Rukovalac izuzecima tipa `RFile::Greska` samo ispisuje poruku koja odgovara sadržaju objekta `g` tog tipa. Za to se koristi operatorska funkcija kojom je u programu 9.2 operator `<<` preklopljen za slučaj kada je drugi operand tipa `RFile::Greska`.

Program 9.6 ostvaruje drugu traženu radnju, prikazivanje sadržaja liste iz datoteke. Ovo je mnogo jednostavnije od obrazovanja liste. Potrebno je samo proći duž liste (sledeći sadržaje polja nar u zapisima), pročitati sadržaj zapisa iz datoteke i prikazati na glavnom izlazu računara. Redni broj početnog zapisa se, naravno, uzima iz glave liste, tj. iz zapisa rednog broja 1.

```
// Prikazivanje sadržaja liste iz relativne datoteke.

#include "rfile.h"
#include <iostream>
#include <iomanip>
using namespace std;

struct Zapis { int sif; char txt[50]; unsigned nar; }

int main () {
    try {
        RFile dat ("Lista.dat");
        Zapis zap;
        dat.read (&zap, 1);
        while (zap.nar) {
            dat.read (&zap, zap.nar);
            cout << setw(5) << zap.sif << " " << zap.txt
        }
    } catch (RFile::Greska g) { cout << g; }
    return 0;
}
```

Program 9.6 – Prikazivanje sadržaja liste iz relativne datoteke (lispisi.c)

Rezultat 9.1 prikazuje primer rada programa 9.5 i 9.6

```
8 lispravi
 5 petak
 2 utorak
 7 nedelja
 6 subota
 1 ponedeljak
 3 sreda
 4 cetvrtak

8 lispisi
 1 ponedeljak
 2 utorak
 3 sreda
 4 cetvrtak
 5 petak
 6 subota
 7 nedelja
```

Rezultat 9.1 – Obrazovanje liste u relativnoj datoteci programima 9.5 i 9.6

Literatura

- [1] Bjarne Stroustrup: **The C++ Programming Language**, *Third Edition*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1997.
 - [2] **Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++**, Accredited Standard Committee X3, 1996.
 - [3] Margaret A. Ellis, Bjarne Stroustrup: **The Annotated C++ Reference Manual**, *ANSI Base Document*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1992.
 - [4] **Borland C++ Builder**, Version 5.0, Inprise Corporation, 2000.
 - [5] **Microsoft Visual C++ .NET**, Version 7.1, Microsoft Corporation, 2003.
 - [6] Brian W. Kernighan, Dennis M. Ritchie: **The C Programming Language**, Second Edition, *Based on Draft-Proposed ANSI C*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.
 - [7] Laslo Kraus: **Rešeni zadaci iz programskog jezika C++**, drugo izdanie, Akademska misao, Beograd, 2006.
 - [8] Laslo Kraus: **Programski jezik C sa rešenim zadacima**, šesto izdanie, Akademska misao, Beograd, 2006.
 - [9] Laslo Kraus: **Rešeni zadaci iz programskog jezika C**, drugo izdanie, Akademska misao, Beograd, 2005.
 - [10] Mark G. Sobell: **A Practical Guide to the UNIX System**, The Benjamin / Cummings Publishing Company, Inc., Menlo Park, California, 1984.

Indeks

- !, operator, 10
- !=, operator, 9
 - u klasi `type_info`, 158
- #, direktyve pretprocесora, 17
- %, operator, 9
- %=, operator, 10, 28
- &
 - modifikator, 24, 55
 - operator
 - binarni, 10
 - unarni, 10, 55, 74
- &&, operator, 10
- &=, operator, 10, 28
- ()
 - operator, 11
 - preklapanje, 114
 - redosled izvršavanja operatora, 11
 - u definiciji funkcije, 14
 - u deklaraciji funkcije, 15
- u naredbi `do`, 13
- u naredbi `for`, 13
- u naredbi `if`, 12
- u naredbi `switch`, 13
- u naredbi `while`, 13
- (tip), operator, 11
 - preklapanje, 106
 - *
 - modifikator, 8, 55
 - operator
 - binarni, 9
 - unarni, 10, 55
 - *=, operator, 10, 28
 - const, modifikator, 8
 - volatile, modifikator, 8
- ,, operator, 10
- ,, operator, 11, 55
- ,, operator, 74
- /. operator, 9
- /*...*/, komentar, 6, 19
- //, komentar, 19
- /=, operator, 10, 28
- ::, operator
 - binarni, 42, 56
 - definisanje zajedničkog polja, 70
 - unarni, 44
- ::*, modifikator, 74
- :
 - kraj proste naredbe, 12
 - prazna naredba, 12
- ?:, operator, 10, 28
- []
 - modifikator, 8, 55
 - operator, 11, 55
 - preklapanje, 112
 - u izrazu `delete`, 35
- ^, operator, 10
- ^=, operator, 10, 28
 - _ kao početak identifikatora, 20
 - _ kao početak identifikatora, 20
- ?)
 - nabranjanje, 7
 - početna vrednost niza, 8
 - sekvenca, složena naredba, 12
 - u opisu `enum`, 7
- !, operator, 10
- !!., operator, 10
- !=, operator, 10, 28
- ~, operator, 10
- +, operator
 - binarni, 9
 - unarni, 9
- ++., operator, 9, 28
 - preklapanje, 104
- +=, operator, 10, 28
- ., operator
- ., operator
 - binarni, 9
 - unarni, 9
- ., operator, 9, 28
 - preklapanje, 104
- =, operator, 10, 28
- >, operator, 11, 55
 - preklapanje, 116
- >*, operator, 74
- <, operator, 9
- <<, operator, 10
 - izlazna konverzija, 31, 105, 290
 - preklapanje, 105
- <=, operator, 10, 28
- <=, operator, 9
 - u šablonu, 218
- =, operator, 10, 28, 55
 - preklapanje, 109
- ==, operator, 9
 - u klasi `type_info`, 158
- >, operator, 9
- >=, operator, 9
- >>, operator, 10
 - preklapanje, 105
 - ulazna konverzija, 31, 105, 290
- >>=, operator, 10, 28
- '\0', znak, 8

A

- abort, funkcija, 198
- abs, funkcija, 241
- adjustfield, konstanta, 291
- adresa
 - &, operator, 10
 - člana, 74
 - objekta, 55
 - podatka, 10
 - polja, 74

diagram klasa, 78
 dinamička dodela memorije, 34
 dinamička konverzija tipa, 157
~~cast~~, klasa za izuzetke, 157
dynamic_cast, operator, 157
~~typeid~~, zaglavje, 157
 dinamički podatak, 17
 početna vrednost, 17
 dinamički tip pokazivača i upućivača, 153
 dinamičko određivanje tipa podataka, 158
~~cast~~, klasa za izuzetke, 158
typeid, operator, 158
~~typeidinfo~~, zaglavje, 158
type_info, klasa, 158
 direktni iterotor, 244
 direktni pristup toku podataka, 293
 direktiva pretpresosora
~~#define~~, 17
 u jeziku C++, 36
~~#elif~~, 17
~~#else~~, 17
~~#endif~~, 17
~~#if~~, 17
~~#ifdef~~, 17
~~#ifndef~~, 17
~~#include~~, 17
~~#makro~~, 17
~~#undef~~, 17
divides<>, klasa, 239
do, naredba, 13
~~while~~, službena reč, 13
 dodela memorije, 34
~~bad_alloc~~, klasa za izuzetke, 35
~~ceil~~, funkcija, 34
delete, operator, 34
free, funkcija, 34
 inicijalizacija, 60
malloc, funkcija, 34
new, operator, 34, 117
new, zaglavje, 35
operator delete, metoda, 117, 118
operator delete[], metoda, 117, 118
operator new, metoda, 117
operator new[], metoda, 117
 oslobođanje memorije, 34
realloc, funkcija, 34
 u jeziku C, 34
 u jeziku C++, 34
 dodela vrednosti, 10, 111
~~=~~, operator, 10, 28
 preklapanje, 109

dodela vrednosti (*nastavak*) objektu, 55
 odnos prema inicijalizaciji, 110
operator=, metoda, 110
 automatsko generisanje, 110
 podrazumevano tumačenje za klasne tipove, 101
 struktura, 10
 dohvatanje delova složenih podataka, 39
doseg, 16
~~::~~, operator
 binarni, 42, 56, 57
 unarni, 44
 blokovski, 16, 20
 člana
 bezimene unije, 23
 klase, 55
 strukture, 16
 unije, 16
 datotečki, 16
 funkciski, 16
explicit, modifikator, 63
extern, modifikator
 funkcije na različitim jezicima, 40
 lokalni identifikator, 16
 podatak, 16

F

fail, metoda, 293
failbit, konstanta, 294
failure, klasa za izuzetke, 294
false, konstanta, 20
false, logička vrednost u jeziku C, 6
filebuf, klasa, 295
fill, metoda, 292
fill<>, funkcija, 261
fill_n<>, funkcija, 261
find, metoda, 253, 257
find<>, funkcija, 263
find_end<>, funkcija, 264
find_first_not_of, metoda, 257
find_first_of, metoda, 257
find_first_of<>, funkcija, 264
find_if<>, funkcija, 263
find_last_not_of, metoda, 257
find_last_of, metoda, 257
fixed, konstanta, 291
fixed, manipulator, 33
flags, metoda, 291
flip, metoda, 259
float, tip, 6
floatfield, konstanta, 291
flush, metoda, 289, 292
for, naredba, 13, 21
for_each<>, funkcija, 260
 formalan argument funkcije, 14

EDIT
 komanda, 4
 urednik teksta, 4
 efekat, bočni, 12
 funkcije, 14
 eksponent realnog broja, 7
 element niza, 8
 indeks, 8
~~#elif~~, direktiva, 17
~~#else~~, direktiva, 17

free, funkcija, 34
friend, modifikator, 72
front, metoda, 247
fstream, klasa, 285, 295
 konstruktor, 286
~~ifstream~~, zaglavje, 285
~~functional~~, zaglavje, 239
funkcija, 14
~~()~~, operator, 11
 argument, 14
 bez parametara, 14
 blok, telo funkcije, 14
 bočni efekat, 14
 član klase, 53
 pristupanje, 70
const, modifikator parametra, 14, 26
 definisanje, 14
 deklarisanje, 15
 destruktur, 66
 dohvatanje delova
 složenih podataka, 39
doseg identifikatora parametra, 16
 formalan argument, 14
 generička, 217
 glavna, 15
 u jeziku C++, 41
 inicijalizacija parametara, 14
 inline, modifikator, 36
 izostavljanje argumenta, 37
 konstruktor, 59
 lokalni podatak, 14
main, 15
 u jeziku C++, 41
 mana ugradivanja u kôd, 36
 metoda, 56
 naziv funkcije, 14
 operatorska, 101
 za binarni operator, 102
 za unarni operator, 102
 parametar funkcije, 14
 parametar glavne funkcije, 15
 podrazumevana vrednost parametra, 37
 pokazivač kao vrednost funkcije, 14
 pokazivač na funkciju, 15
 povezivanje funkcija pisanih na različitim jezicima, 40
 povratak iz funkcije, 14
 pozivanje, 11, 15
 pravilo za izbor medu funkcijama s preklapljenim imenima, 38
 preklapanje imena, 38
 prijateljska, 72
 prototip, 15

funkcija, bibliotečka (*nastavak*) redosled izračunavanja argumenta, 12
return, naredba, 14
 rukovalac izuzecima, 195
 struktura kao vrednost funkcije, 14
 stvaran argument, 14
 telo, 14
 tip vrednosti funkcije, 14
 ugradena, 36
 ugradivanje u kôd, 36
 unija kao vrednost funkcije, 14
 upućivač
 kao parametar, 26
 kao vrednost funkcije, 26
 virtuelna, 152
void, prazan niz parametara, 14
void, vrednost funkcije, 14
 volatile, modifikator parametra, 14, 26
 vrednost funkcije, 14
funkcija, bibliotečka
~~abort~~, 198
~~abs~~, 241
~~arg~~, 241
binary_search<>, 268
~~calloc~~, 34
~~conj~~, 241
~~copy~~<>, 265
~~copy_backward~~<>, 265
~~cos~~, 241
~~cosh~~, 241
~~count~~<>, 264
~~count_if~~<>, 264
~~equal~~<>, 267
~~equal_range~~<>, 268
~~exp~~, 241
~~fill~~<>, 261
~~fill_n~~<>, 261
~~find~~<>, 263
~~find_end~~<>, 264
~~find_first_of~~<>, 264
~~find_if~~<>, 263
~~for_each~~<>, 260
~~free~~, 34
~~generate~~<>, 261
~~generate_n~~<>, 261
~~imag~~, 241
 jeziku C, 18
lexicographical_compare<>, 267
~~log~~, 241
~~log10~~, 241
~~lower_bound~~<>, 268
~~malloc~~, 34
~~max~~<>, 260
~~max_element~~<>, 264

funkcija, bibliotečka (*nastavak*)
~~merge~~<>, 268
~~min~~<>, 260
~~min_element~~<>, 264
~~next_permutation~~<>, 267
~~norm~~, 241
~~polar~~, 241
~~pow~~, 241
~~prev_permutation~~<>, 267
~~printf~~, 31
~~rand~~, 50
~~real~~, 241
~~realloc~~, 34
~~remove~~<>, 266
~~remove_copy~~<>, 266
~~remove_copy_if~~<>, 266
~~remove_if~~<>, 266
~~replace~~<>, 265
~~replace_copy~~<>, 265
~~replace_copy_if~~<>, 265
~~replace_if~~<>, 265
~~reverse~~<>, 266
~~reverse_copy~~<>, 266
~~scanf~~, 31
~~search~~<>, 264
~~set_terminate~~, 198
~~set_unexpected~~, 198
~~sin~~, 241
~~sinh~~, 241
~~sort~~<>, 268
~~sqrt~~, 241
~~stable_sort~~<>, 268
~~swap~~<>, 260
~~tan~~, 241
~~tanh~~, 241
~~terminate~~, 198
~~transform~~<>, 261
~~unexpected~~, 198
~~unique~~<>, 266
~~unique_copy~~<>, 266
~~upper_bound~~<>, 268
funkcija, prijateljska, 72
friend, modifikator, 72
 operatorska funkcija, 102
funkcijska klasa, 239
~~divides~~<>, 239
~~equal_to~~<>, 240
~~functional~~, zaglavje, 239
~~greater~~<>, 240
~~greater_equal~~<>, 240
~~less~~<>, 240
~~less_equal~~<>, 240
~~logical_and~~<>, 240
~~logical_not~~<>, 240
~~logical_or~~<>, 240
~~minus~~<>, 239
~~modulus~~<>, 239
~~multiplies~~<>, 239

dijagram klasa, 78
 dinamička dodata memorije, 34
 dinamička konverzija tipa, 157
~~bad_cast~~, klasa za izuzetke, 157
~~dynamic_cast~~, operator, 157
~~<typeinfo>~~, zaglavje, 157
 dinamički podatak, 17
 početna vrednost, 17
 dinamički tip pokazivača i upućivača, 153
 dinamičko određivanje tipa podataka, 158
~~bad_typeid~~, klasa za izuzetke, 158
~~typeid~~, operator, 158
~~<typeinfo>~~, zaglavje, 158
~~type_info~~, klasa, 158
 direktan iterator, 244
 direktan pristup toku podataka, 293
 direktna pretprocesora
~~#define~~, 17
 u jeziku C++, 36
~~#elif~~, 17
~~#else~~, 17
~~#endif~~, 17
~~#if~~, 17
~~#ifdef~~, 17
~~#ifndef~~, 17
~~#include~~, 17
~~#makro~~, 17
~~#undef~~, 17
~~divides<>~~, klasa, 239
~~do~~, naredba, 13
~~while~~, službena reč, 13
 dodata memorije, 34
~~bad_alloc~~, klasa za izuzetke, 35
~~callloc~~, funkcija, 34
~~delete~~, operator, 34
~~free~~, funkcija, 34
 inicijalizacija, 60
~~malloc~~, funkcija, 34
~~new~~, operator, 34, 117
~~new~~, zaglavje, 35
~~operator delete~~, metoda, 117, 118
~~operator delete[]~~, metoda, 117, 118
~~operator new~~, metoda, 117
~~operator new[]~~, metoda, 117
 oslobadanje memorije, 34
~~realloc~~, funkcija, 34
 u jeziku C, 34
 u jeziku C++, 34
 dodata vrednosti, 10, 111
~~=~~, operator, 10, 28
 preklapanje, 109

dodata vrednosti (*nastavak*) objektu, 55
 odnos prema inicijalizaciji, 110
~~operator=~~, metoda, 110
 automatsko generisanje, 110
~~#endif~~, direktiva, 17
 podrazumevano tumačenje za klasne tipove, 101
 struktura, 10
 dohvatanje delova složenih podataka, 39
 doseg, 16
~~::~~, operator
 binarni, 42, 56, 57
 unarni, 44
 blokovski, 16, 20
 člana
 bezimene unije, 23
 klase, 55
 strukture, 16
 unije, 16
 datotečki, 16
 funkciski, 16
 identifikator
 funkcije na različitim jezicima, 40
 lokalni identifikator, 16
 podatak, 16

F

~~fail~~, metoda, 293
~~failbit~~, konstanta, 294
~~failure~~, klasa za izuzetke, 294
~~false~~, konstanta, 20
~~false~~, logička vrednost u jeziku C, 6
~~filebuf~~, klasa, 295
~~fill~~, metoda, 292
~~fill<>~~, funkcija, 261
~~fill_n<>~~, funkcija, 261
~~find~~, metoda, 253, 257
~~find<>~~, funkcija, 263
~~find_end<>~~, funkcija, 264
~~find_first_not_of~~, metoda, 257
~~find_first_of~~, metoda, 257
~~find_first_of<>~~, funkcija, 264
~~find_if<>~~, funkcija, 263
~~find_last_not_of~~, metoda, 257
~~find_last_of~~, metoda, 257
~~fixed~~, konstanta, 291
~~fixed~~, manipulator, 33
~~flags~~, metoda, 291
~~flip~~, metoda, 259
~~float~~, tip, 6
~~floatfield~~, konstanta, 291
~~flush~~, metoda, 289, 292
~~for~~, naredba, 13, 21
~~for_each<>~~, funkcija, 260
 formalan argument funkcije, 14

E

EDIT
 komanda, 4
 urednik teksta, 4
 efekat, bočni, 12
 funkcije, 14
 eksponent realnog broja, 7
 element niza, 8
 indeks, 8
~~#else~~, direktiva, 17
~~#else~~, direktiva, 17

~~free~~, funkcija, 34
~~friend~~, modifikator, 72
~~front~~, metoda, 247
~~fstream~~, klasa, 285, 295
 konstruktor, 286
~~<fstream>~~, zaglavje, 285
~~<functional>~~, zaglavje, 239
 funkcija, 14
~~()~~, operator, 11
 argument, 14
 bez parametara, 14
 blok, telo funkcije, 14
 bočni efekat, 14
 član klase, 53
 pristupanje, 70
~~const~~, modifikator parametra, 14, 26
 definisanje, 14
 deklarisanje, 15
 destruktur, 66
 dohvatanje delova složenih podataka, 39
 doseg identifikatora parametra, 16
 formalan argument, 14
 generička, 217
 glavna, 15
 u jeziku C++, 41
 inicijalizacija parametara, 14
~~inline~~, modifikator, 36
 izostavljanje argumenta, 37
 konstruktor, 59
 lokalni podatak, 14
~~main~~, 15
 u jeziku C++, 41
 mana ugradivanja u kôd, 36
 metoda, 56
 naziv funkcije, 14
 operatorska, 101
 za binarni operator, 102
 za unarni operator, 102
 parametar funkcije, 14
 parametar glavne funkcije, 15
 podrazumevana vrednost parametra, 37
 pokazivač kao vrednost funkcije, 14
 pokazivač na funkciju, 15
 povezivanje funkcija pisanih na različitim jezicima, 40
 povratak iz funkcije, 14
 pozivanje, 11, 15
 pravilo za izbor među funkcijama s preklapljenim imenima, 38
 preklapanje imena, 38
 prijateljska, 72
 prototip, 15
 funkcija (*nastavak*) redosled izračunavanja argumenta, 12
~~return~~, naredba, 14
 rukovalac izuzecima, 195
 struktura kao vrednost funkcije, 14
 stvaran argument, 14
 telo, 14
 tip vrednosti funkcije, 14
 ugradena, 36
 ugradivanje u kôd, 36
 unija kao vrednost funkcije, 14
 upućivač
 kao parametar, 16
 kao vrednost funkcije, 26
 virtuelna, 152
~~void~~, prazan niz parametara, 14
~~void~~, vrednost funkcije, 14
 volatile, modifikator parametra, 14, 26
 vrednost funkcije, 14
 funkcija, bibliotečka
~~abort~~, 198
~~abs~~, 241
~~arg~~, 241
~~binary_search<>~~, 268
~~calloc~~, 34
~~conj~~, 241
~~copy<>~~, 265
~~copy_backward<>~~, 265
~~cos~~, 241
~~cosh~~, 241
~~count<>~~, 264
~~count_if<>~~, 264
~~equal<>~~, 267
~~equal_range<>~~, 268
~~exp~~, 241
~~operatorska~~, 101
~~fill<>~~, 261
~~fill_n<>~~, 261
~~find<>~~, 263
~~find_end<>~~, 264
~~find_first_of<>~~, 264
~~find_if<>~~, 263
~~for_each<>~~, 260
~~free~~, 34
~~generate<>~~, 261
~~generate_n<>~~, 261
~~imag~~, 241
 jezika C, 18
~~lexicographical_compare<>~~, 267
~~log~~, 241
~~log10~~, 241
~~lower_bound<>~~, 268
~~malloc~~, 34
~~max<>~~, 260
~~max_element<>~~, 264
 funkcija, bibliotečka (*nastavak*)
~~merge<>~~, 268
~~min<>~~, 260
~~min_element<>~~, 264
~~next_permutation<>~~, 267
~~norm~~, 241
~~polar~~, 241
~~pow~~, 241
~~prev_permutation<>~~, 267
~~printf~~, 31
~~rand~~, 50
~~real~~, 241
~~realloc~~, 34
~~remove<>~~, 266
~~remove_ccpy<>~~, 266
~~remove_copy_if<>~~, 266
~~remove_if<>~~, 266
~~replace<>~~, 265
~~replace_copy<>~~, 265
~~replace_copy_if<>~~, 265
~~replace_if<>~~, 265
~~reverse<>~~, 266
~~reverse_copy<>~~, 266
~~scanf~~, 31
~~search<>~~, 264
~~set_terminate~~, 198
~~set_unexpected~~, 198
~~sin~~, 241
~~sinh~~, 241
~~sort<>~~, 268
~~sqrt~~, 241
~~stable_sort<>~~, 268
~~swap<>~~, 260
~~tan~~, 241
~~tanh~~, 241
~~terminate~~, 198
~~transform<>~~, 261
~~unexpected~~, 198
~~unique<>~~, 266
~~unique_copy<>~~, 266
~~upper_bound<>~~, 268
 funkcija, prijateljska, 72
~~friend~~, modifikator, 72
 operatorska funkcija, 102
 funkcionska klasa, 239
~~divides<>~~, 239
~~equal_to<>~~, 240
~~<functional>~~, zaglavje, 239
~~greater<>~~, 240
~~greater_equal<>~~, 240
~~less<>~~, 240
~~less_equal<>~~, 240
~~logical_and<>~~, 240
~~logical_not<>~~, 240
~~logical_or<>~~, 240
~~minus<>~~, 239
~~magnitude<>~~, 239
~~multimap<>~~, 239

funkcijska klasa (*mastavak*)
operator<<, 239
operator>>, 240
plus, 239
 funkcijski doseg, 16
 tuzija nizova, primer generičke funkcije, 226

G

greater, metoda, 292
generate, funkcija, 261
generate_n, funkcija, 261
 generička
 metoda klase, 224
 unutrašnja klasa, 225
 generička funkcija, 217
<>, 218
class, u šablonu, 218
 definisanje šablonu, 217
 generisanje, 219
 automatsko, 219
 na zahtev, 220
 mana, 218
 parametar šablonu, 218
 specijalizacija, potpuna, 223
template, službena reč, 217
typename, u šablonu, 218

generička klasa, 217
<>, 218
class, u šablonu, 218
 definisanje šablonu, 217
 generička
 metoda klase, 225
 unutrašnja klasa, 225
 generisanje, 219
 automatsko, 219
 na zahtev, 220
 mana, 218
 parametar šablonu, 218
 podrazumevana vrednost
 parametra, 221
 specijalizacija, 222
 delimična, 222
 potpuna, 222
template, službena reč, 217
typename, u šablonu, 218

generički pokazivač, 8
 generisanje
 automatsko
 destruktora, 67
 konstruktor
 kopije, 62
 podrazumevanog, 61
 operatorske funkcije
 operator=, 110
 klase na osnovu šablonu, 219

geometrijske figure, primer klase, 160, 269
 get, metoda, 33, 34, 289
getline, metoda, 34, 256, 289
 glavna funkcija
 u jeziku C++, 41
 glavni izlaz, 3, 5
cout, 30, 105, 285
 skretanje na datoteku, 3, 5
 glavni program, 15
 glavni ulaz, 3, 5
cin, 30, 105, 285
 skretanje na datoteku, 3, 5
 globalni identifikator, 16
<<, operator, unarni, 44
 povezivanje, 16
static, modifikator, 16
 globalni podatak, trajnost, 16
good, metoda, 293
goto, naredba, 14
greater, klasa, 240
greater_equal, klasa, 240
 greška, signalizacija, 293
 greške, primer klase, 205
 grupisanje objekata, 141
 grupisanje operatora, 11, 27

H

heksadecimalna celobrojna konstanta, 6
hex
 konstanta, 291
 manipulator, 32

I

i, logičko, *&&*, 10
 po bitovima, *&*, 10
 identifikator, 6
<<, operator
 binarni, 42, 56
 unarni, 44
_, kao početak, 20
__, kao početak, 20
 destruktora, 66
 doseg, 16
 blokovski, 16, 20
 datotečki, 16
 funkcijski, 16
 klasni, 55
 prostorski, 41
 prototipski, 16
 strurni, 16
 doseg identifikatora
 člana unutrašnje klase, 75
 lokalne klase, 76
 unutrašnje klase, 75
 dužina, 6

geometrijske figure, primer klase, 160, 269
 globalni, 16
 klase, 54
 konstruktora, 59
 lokalna za datoteku, 16
 lokalni, 16
 nabranja
 u jeziku C, 7
 u jeziku C++, 22
 povezivanje, 16
 spoljašnje, 16
 unutrašnje, 16, 44
 redeklarisanje unutar izvedene klase, 145
 uklopjenog doseg, 16
static, modifikator
 unutrašnje povezivanje, 16
 zajednički član klase, 69
 strukture
 u jeziku C, 9
 u jeziku C++, 22
 tipa, 8
 unije
 u jeziku C, 9
 u jeziku C++, 22
 vidljivost, 16
#if, direktiva, 17
#if, naredba, 12, 21
 pravilo uklapanja, 12
#ifdef, direktiva, 17
#ifndef, direktiva, 17
ifstream, klasa, 285, 295
 konstruktur, 286
ignore, metoda, 290
lli, logičko, 11, 10
 po bitovima, isključivo, *^*, 10
 po bitovima, uključivo, *|*, 10
imag, funkcija, 241
imag, metoda, 241
in, konstanta, 287
#include, direktiva, 17
 indeks elementa niza, 8
 indeksiranje, 11
[], operator, 11
 odnos prema adresnoj aritmetici, 11
operator[], metoda, 112
 u klasi *basic_string*, 255
 u klasi *bitset*, 259
 u klasi *map*, 253
 u klasi *vector*, 247
 indirektno adresiranje, 10
 inicijalizacija, 110
()
 niza, 8
 strukture, 9
 unije, 9

inicijalizacija (*mastavak*)
 dinamičkog niza, 65
 dinamičkog objekta, 60
 elementara nizova, 65
 inicijalizator, 59
 klase, 59
 klasnim objektom, 111
 konstruktur, 58
 konverzije, 63
 kopije, 62, 111
 podrazumevani, 61
 niza, 8
 objekata, 65
 odnos prema dodeli vrednosti, 110
 osnovnih tipova, 7
 parametra funkcije, 63
 podataka, 21
 u dinamičkoj zoni memorije, 35
 strukture, 9
 unije, 9
 upućivača, 24
 zajedničkog polja, 70
 inicijalizator, 7, 59
 u definiciji konstruktora, 60
inline, modifikator, 36
 podrazumevani za metodu, 54
insert, metoda, 247, 253, 257
int, tip, 6
unsigned, 6
 integral, određeni
 metoda pravougaonika, 208
 primer sistema klase, 204
 integrisano okruženje za razvoj programa, 5
internal, konstanta, 291
invalid_argument, klasa za izuzetke, 294
iomanip, zaglavlj. 32
ios, klasa, 294
iostream, klasa, 295
<iostream>, zaglavlj. 31, 47, 105, 285
<iostream.h>, zaglavlj. 47
is_open, metoda, 287
istream, klasa, 105, 285, 295
istringstream, klasa, 285, 295
 konstruktur, 288
 iterator s proizvoljnim pristupom, 243
 iterator, klasa, 242
const_iterator, klasa, 244
const_reverse_iterator, klasa, 244
 direktan iterator, 244
 dvostran iterator, 243

iterator, klasa, 242
 iterator s proizvoljnim pristupom, 243
iterator, klasa, 244
 izlazni iterator, 243
 jednosmeran iterator, 243
 obrnut iterator, 244
 odnos prema pokazivačima, 243
 operator, 243
reverse_iterator, klasa, 244
 ulazni iterator, 243

iteratori za zbirke geometrijskih figura, primer klase, 170
 iteratori za zbirke podataka, 171
 izlaz podataka, 285
<<, operator, 105
<cstdio>, zaglavlj. 286
<iostream>, zaglavlj. 47
ostream, klasa, 105
<stdio.h>, zaglavlj. 286
 izlaz za poruke, *cerr*, 286
 izlaz za zabeleške, *clog*, 286
 izlazna konverzija, 31
<<, operator, 31, 105, 290
<iomanip.h>, zaglavlj. 32
 manipulatori, 32
operator<<, funkcija, 105
 izlazni iterator, 243
 izlazni tok podataka, 30, 285
<<, operator, 31, 105, 290
cerr, izlaz za poruke, 286
clog, izlaz za zabeleške, 286
cout, glavni izlaz, 30, 105, 285
 datoteka
 binarna, 286
 stvaranje, 286
failure, klasa za izuzetke, 294
fstream, klasa, 285
 konstruktur, 286
<fstream>, zaglavlj. 285
<iomanip>, zaglavlj. 32
<iostream>, zaglavlj. 31, 285
<iostream.h>, zaglavlj. 47
 izlazna konverzija, 31
 manipulatori, 32
ofstream, klasa, 285
 konstruktur, 286
operator<<, funkcija, 105
ostream, klasa, 285
 konstruktur, 286
 pozicioniranje unutar toka, 293
 signalizacija grešaka, 293
<stdio.h>, zaglavlj. 31
 u memoriji, 288
 za datoteke, 286

izuzetak, 9
 adresni, 10
 kao naredba, 12
 lanac izraza, 10
tvrdnost, 11
 redosled izračunavanja, 11
 uslojni, 10
 izuzetak, 193
abort, funkcija, 198
bad_exception, klasa za izuzetke, 198
catch, službena reč, 195
 definisanje rukovalaca, 195
exception, klasa, 199
<exception>, zaglavlj. 199
 neочекivani, 198
 neprihvaci, 198
 pravilo za redosled navedenja rukovalaca, 197
 prihvatanje, 197
 prijavljivanje, 195
 rukovalac izuzecima, 193
 rukovanje izuzecima, 194
set_terminate, funkcija, 198
set_unexpected, funkcija, 198
 spisak tipova mogućih izuzetaka, 196
 standardni, 199
terminate, funkcija, 198
throw, operat. 195
 tip izuzetka, 194
 tok prihvatanja, 197
try, naredba, 194
 uklapanje naredbi *try*, 195
unexpected, funkcija, 198
 univerzalni rukovalac, 195
 izuzetak, bibliotečki
bad_alloc, 35
bad_cast, 157
bad_exception, 198
bad_typeid, 158
exception, 199
failure, 294
invalid_argument, 259
length_error, 246
out_of_range, 247, 255, 258, 259
overflow_error, 259
 izvedena klasa, 142
 apstraktne
 klase, 156
 metoda, 156
class, opis tipa, 142
 definisanje, 142
 destruktorn, tok izvršavanja, 149
 dijagram klasa, 144
dynamic_cast, operat. 157

Indeks**Indeks**

314

izvedena klasa (*nastavak*) inicijalizacija polja, 149
javan član, 142
javno izvođenje, 143
konstruktor, tok izvršavanja, 149
konverzija tipa
 nadole, 152
 nagore, 152
 pokazivača, 151
 u osnovnu klasu, 150
 upućivača, 152
nasleđivanje članova, 142
podrazumevani način izvođenja, 143
prijateljska funkcija, 143
privatan član, 142
private, oznaka, 142
privatno izvođenje, 143
protected, oznaka, 142
public, oznaka, 142
redeklarisanje identifikatora, 145
stvaranje primerka, 149
uništavanje
 polja, 149
 primerka, 149
virtual, modifikator
 za metode, 153
virtuelna metoda, 152
 pozivanje, 153
virtuelna osnovna klasa, 148
višestruko nasleđivanje, 142
zaštićen član, 142
zaštićeno izvođenje, 143
izvedena struktura, 143
 podrazumevani način izvođenja, 143
izvedeni tip, 8
 niz, 8
 pokazivač, 8
 struktura, 9
 unija, 9
izvođenje klase, 143
 javno, 143
 podrazumevani način, 143
 privatno, 143
 zaštićeno, 143
izvršavanje programa, 3. 5

J

javan član klase, 53, 142
public, oznaka, 54, 142
javna osnovna klasa, 143
public, modifikator, 143
javno izvođenje klase, 143
public, modifikator, 143
jednako, ==, 9
jednosmeran iterotor, 243
jednostruka tačnost realnih brojeva, 6

K

klasa
 diagram klasa, 78
 odnos među klasama, 79
 agregacija, 79
 asocijacija, 79
 kompozicija, 79
 nasleđivanje, 144
 zavisnost, 79
klasa, bibliotečka (*nastavak*)
 stack<>, 250
 streambuf, 295
 string<>, 254
 stringbuf, 295
 stringstream, 285, 295
 type_info, 158
 vector<>, 245
 vector<bool>, 258
klasa, primer
 geometrijske figure, 160, 269
 greške, 205
 iteratori za zbirke geometrijskih figura, 170
 krugovi u ravni, 93
 određeni integral, 204
 polinomi, 125
 redovi, 133
 relativne datoteka, 296
 stekovi, 90, 229
 tačke u ravni, 81
 upoređivanje nizova, 232
 uređeni skupovi, 84
 uređivanje nizova, 232
 vektori, 199
 vremenski intervali, 120
 zbirke geometrijskih figura, 170
klasa, tip (*nastavak*)
 &, modifikator, 55
 &, operator, 55, 74
 *, modifikator, 55
 *. operator, 55
 .. operator, 55
 .*, operator, 74
 ::*, modifikator, 74
 [], modifikator, 55
 [], operator, 55
 ->, operator, 55
 ->*, operator, 74
 adresa objekta, 55
apstraktna
 klasa, 156
 metoda, 156
atribut klase, 53
član klase, 53
class, opis tipa, 54, 142
definicije, 54, 142
 s konstruktorima, 59
definicije tumačenja operatora, 101
deklaracija, 54
destruktur, 66
 tok izvršavanja, 67, 149
 virtuelni, 154
dodata vrednosti objektu, 55
doseg identifikatora
 člana klase, 55
 člana unutrašnje klase, 75
 identifikatora unutrašnje klase, 75

Indeks

klasa, tip (*nastavak*)
friend, modifikator, 72
funkcija kao član, 53
generička, 217
 unutrašnja, 225
identifikator klase, 54
inicijalizacija
 niza objekata, 65
 objekta, 58
 odnos prema dodeli vrednosti, 110
 polja, 64, 149
 nepromenljivog, 65
 upućivačkog, 65
 zajedničkog, 70
izvedena, 142
 nasleđivanje članova, 142
 višestruko nasleđivanje članova, 142
javan član, 53, 142
javna osnovna klasa, 143
javno izvođenje, 143
Klassa:::*, modifikator, 74
konstanta klasnog tipa, 69
konstruktor, 58, 59
explicit, modifikator, 63
 konverzije, 63
 kopije, 62, 111
 podrazumevani, 61
 tok izvršavanja, 64, 149
konverzija tipa, 63, 107
 izvedene klase u osnovnu klasu, 150
 nadole, 152
 nagore, 152
 pokazivača, 151
 upućivača, 152
lokalna, 76
metoda klase, 53, 56
definicije
 izvan definicije klase, 57
 unutar definicije klase, 54, 57
deklarisanje, 54
 odnos prema prijateljskoj funkciji, 72
 skriveni parametar metode, 56
this, pokazivač, 56
 ugradena, 54
niz objekata, 55
objekat klase, 53
 definisanje, 55
 tekući, 56
odnos prema strukturm dosegu, 76
operatorska funkcija, 101
osnovna, 142
 podatak kao član, 53
 pojedinačan član, 69
 kao beli znak, 6

kompleksan broj, tip, 241
čitanje i pisanje, 241
<complex>, zaglavlje, 241
complex<>, klasa, 241
operatori, 241
komplementiranje
 logičko ne, !, 10
 po bitovima, ~, 10
kompozicija, odnos među klasama, 79
 diagram klasa, 78
konstanta, 6
celobrojna, 6
const, modifikator
u jeziku C,
 u jeziku C++, 20
decimalna, 6
heksadecimalna, 6
klasnog tipa, 69
logička
 u jeziku C, 6
 u jeziku C++
 false, 20
 true, 20
nabrojana
 u jeziku C, 7
 u jeziku C++, 22
neoznačena, 7
niska, 8
oktalna, 6
realna, 7
 dvostruka tačnost, 7
 jednostruka tačnost, 7
 višestruka tačnost, 7
simbolička
#define, direktiva, 17
nabranjanje
 u jeziku C, 7
 u jeziku C++, 22
NULL, 8
 u jeziku C++, 20
 u nabranjanju, 22
znakovna
 u jeziku C, 7
 u jeziku C++, 20
konstanta, simbolička
 adjustfield, 291
 app, 287
 ate, 287
 badbit, 294
 basefield, 291
 beg, 293
 binary, 287
 boolalpha, 291
 cur, 293
 dec, 291
 end, 293
 exp, 293, 289

konstanta, simbolička (*mastavak*)
eofbit, 294
failbit, 294
fixed, 291
floatfield, 291
hex, 291
in, 287
internal, 291
left, 291
oct, 291
out, 287
right, 291
scientific, 291
showbase, 291
snowpoint, 291
snowpos, 291
skipws, 291
trunc, 287
uppercase, 291
konstruktor, 58, 59
 definisanje, 60
explicit, modifikator, 63
inicijalizacija parametra funkcije, 63
konstante kao argumenti konstruktora, 69
konverzije, 63
kopije, 62, 111
 automatsko generisanje, 62
podrazumevani, 61
 automatsko generisanje, 61
pozivanje, 59
 automatsko, 59
 eksplicitno, 60
preklapanje imena, 59
tok izvršavanja, 64, 149
unije, 77
za proste tipove, 65
konstruktor konverzije, 63
 za proste tipove, 66
konstruktor kopije, 62, 111
 automatsko generisanje, 62
 za proste tipove, 65
konverzija tipa, 11, 107
 (*tip*), operator, 11
automatska, 11
 ograničenja za klasne tipove, 107
bad_cast, klasa za izuzetke, 157
const_cast, operator, 30
dinamička, 157
dynamic_cast, operator, 157
klasnih tipova
 konstruktorom, 63, 107
 preklapanjem operatora
 (*tip*), 106
nabranjanja, 22

konverzija tipa (*nastavak*)
 nadole, 152
 nagore, 152
 operator tip, metoda, 106
 pokazivača, 24, 151
 preklapanje operatora za konverziju tipa, 106
reinterpret_cast, operator, 29
static_cast, operator, 28
 upućivača, 152
 upućivačkog parametra funkcije, 107
konverzija, ulazno-izlazna, 31, 290
 <<, operater, 31, 105, 290
 >>, operater, 31, 105, 290
fill, metoda, 292
flags, metoda, 291
 <<omanip>>, zaglavlj. 32
manipulatori, 32
operator<<, funkcija, 105
operator>>, funkcija, 105
precision, metoda, 292
printf, funkcija, 31
scanf, funkcija, 31
setf, metoda, 290
unsetf, metoda, 291
width, metoda, 292
krugovi u ravni, primer klase, 93

L

lanac izraza, 10
left, konstanta, 291
left, manipulator, 32
leksički simbol, 6
#define, direktiva, 17
identifikator, 6
konstanta, 6
službena reč, 6
#undef, direktiva, 17
zamena leksičkih simbola, 17
length_error, klasa za izuzetke, 246
less<>, klasa, 240
less_equal<>, klasa, 240
lexicographical_compare<>, funkcija, 267
 <iostream>, zaglavlj. 249
 list<>, klasa, 249
lista, sekvenčialna zbirka podataka, 249
log, funkcija, 241
log10, funkcija, 241
logical, tip
 u jeziku C, 6
 u jeziku C++, 20
 bool, 20

M

main, funkcija, 15
 parametar glavne funkcije, 15
 u jeziku C++, 41

makro, direktiva, 17
 upotrebljiva vrednost u jeziku C++, 36
malloc, funkcija, 34
mana
 generičkih funkcija i klasa, 218
 ugradenih funkcija, 36
manipulator, 32
 coolalpha, 32
 dec, 32
 endl, 32
 fixed, 33
 hex, 32
 left, 32
 noboolalpha, 32
 noshowbase, 33
 noshowpoint, 33
 noshowpos, 32
 noskipws, 32
 oct, 32
 right, 32
 scientific, 33
 setbase, 32
 setfill, 32
 setprecision, 33
 setw, 32
 showbase, 32
 showpoint, 33
 showpos, 32
 skipws, 32
 ws, 32
manje, <, 9
manje ili jednako, <=, 9
mantisa realnog broja, 7
<map>, zaglavlj. 252
map<>, klasa, 252
 konstruktor, 252
matematičke funkcije, 18
 <cmath>, zaglavlj. 47
 <math.h>, zaglavlj. 18
 <math.h>, zaglavlj. 18
matrica, 8
max<>, funkcija, 260
max_element<>, funkcija, 264
max_size, metoda, 246
memorijska, operativna – dinamička dodata, 34
merge, metoda, 250
merge<>, funkcija, 268
metoda klase, 53, 56
 .., operator, 11
 ->, operator, 11
const, modifikator, 57
definisanje
 izvan definicije klase, 57
 unutar definicije klase, 54,
 57
deklarisanje, 54

metoda klase (*mastavak*)
 destruktur, 66
 doseg, 16
 generička, 224
javna, 53
konstruktor, 59
 konverzije, 63
 kopije, 62
 podrazumevani, 61
lokalne klase, 76
odnos prema prijateljskoj funkciji, 72
operatorska funkcija, 102
pojedinačna, 69
polimorfna, 154
pozivanje, 55
preklapanje operatora
 binarnog, 102
 unarnog, 102
pristupanje, 70
private, oznaka, 53, 142
privatna, 53
protected, oznaka, 142
public, oznaka, 53, 142
iskriveni parametar, 56
static, modifikator, 69
tekući objekat, 56
 nepromenljiv i nepostojan, 57
this, pokazivač, 56
ugradena, 54
upotreba zajedničke metode, 70
virtual, modifikator, 153
virtuelna, 152, 153
 destruktur, 154
volatile, modifikator, 57
vrednost funkcije, 54
zajednička, 69, 70
metoda podele za uređivanje, 48
metoda pravougaonika za izračunavanje određenog integrala, 208
metoda za uređivanje
 metoda podele, 48
 quick sort, 48
metoda, bibliotečka
 any, 259
 append, 255
 assign, 246, 255
 at, 247
 back, 247
 bad, 293
 before, 159
 begin, 246
 bitset, konstruktor, 258
 c_str, 257
 capacity, 246
 clear, 248, 294
 put, 34, 289
 putback, 290

metoda, bibliotečka (*mastavak*)
 compare, 256
 compare, konstruktor, 241
 copy, 256
 count, 252, 259
 data, 257
 empty, 246
 end, 246
 end, 293
 equal_range, 253
 erase, 248, 253, 257
 exceptions, 294
 fail, 293
 fill, 292
 find, 253, 257
 find_first_not_of, 257
 find_first_of, 257
 find_last_not_of, 257
 find_last_of, 257
 flags, 291
 flip, 259
 flush, 289, 292
 front, 247
 fstream, konstruktor, 286
 gcount, 292
 get, 33, 34, 289
 getline, 34, 256, 289
 good, 293
 ifstream, konstruktor, 286
 ignore, 290
 imag, 241
 insert, 247, 253, 257
 is_open, 287
 istringstream, konstruktor, 288
 lower_bound, 253
 map, konstruktor, 252
 max_size, 246
 merge, 250
 multimap, konstruktor, 252
 name, 159
 none, 259
 ofstream, konstruktor, 286
 open, 287
 ostringstream, konstruktor, 288
 peek, 290
 pop, 251
 pop_back, 247
 pop_front, 249
 precision, 292
priority_queue, konstruktor, 251
push, 251
push_back, 247
push_front, 249
put, 34, 289
putback, 290

metoda, biblioteka (*nastavak*)
 queue, konstruktor, 251
 rbegin, 246
 read, 292
 real, 241
 remove, 250
 remove_if, 250
 rend, 246
 replace, 257
 reserve, 246
 reset, 259
 resize, 246
 reverse, 250
 rfind, 257
 seekg, 293
 seekp, 293
 set, 259
 setf, 290
 size, 246
 sort, 250
 splice, 249
 stack, konstruktor, 251
 str, 288
 string, konstruktor, 255
 stringstream, konstruktor, 288
 substr, 256
 swap, 248
 tellg, 293
 tellp, 293
 test, 259
 to_string, 259
 to_ulong, 259
 top, 251
 unique, 250
 unsetf, 291
 upper_bound, 253
 vector, konstruktor, 245
 width, 292
 write, 292
 min<>, funkcija, 260
 min_element<>, funkcija, 264
 minus<>, klasa, 239
 množenje, *
 modifikator, 7
 &, 24, 55
 *, 8, 55
 *const, 8
 *volatile, 8
 ::*, 74
 [], 8
 auto, 16
 const, 14
 u jeziku C, 7
 u jeziku C++, 20
 za tekući objekat metode, 57
 za upućivački parametar
 funkcije, 107
 modifikator (*nastavak*)
 explicit, 63
 extern, 16
 funkcije na različitim jezicima
 ma, 40
 friend, 72
 inline, 36
 Klasa::*, 74
 mutable, 23
 register, 16
 static
 trajati podatak, 16
 unutrašnje povezivanje globalnih identifikatora, 16
 zajednički član klase, 69
 virtual
 za klase, 148
 za metode, 153
 volatile, 7, 14
 za tekući objekat metode, 57
 modulus<>, klasa, 239
 MS-DOS, operativni sistem, 4
 BC, komanda, 5
 BCE, komanda, 5
 BCC, komanda, 4
 BCC32, komanda, 4
 EDIT
 komanda, 4
 urednik teksta, 4
 integrисано окружење за развој
 programa, 5
 C++ Builder, 5
 izvršавање програма, 5
 obrada programa, 4
 пovezivanje prevedenog oblika
 programa, 4
 prevedenje izvornog teksta programa, 4
 sastavljanje izvornog teksta programa, 4
 skretanje glavnog ulaza i izlaza, 5
 multimap<>, klasa, 252
 konstruktor, 252
 nepromenljiv
 podatak, 7
 const, modifikator
 u jeziku C, 7
 u jeziku C++, 20
 pokazivač, 8
 nestrukturirani tip, 6
 new, operator, 34, 117
 bad_alloc, klasa za izuzetke,
 35
 <new>, zaglavlj. 35
 next_permutation<>, funkcija, 267
 niska, tip, 8
 '\0', 8
 <cstring>, zaglavlj. 47
 konstantna, 8
 namespaces, naredba, 41
 naredba, 12
 break, 13
 class, 142
 continue, 13
 definisanje
 klase, 54
 podataka
 inicijalizator, 59
 u jeziku C, 7
 u jeziku C++, 20
 tipa, 8
 do, 13
 enum, 7
 for, 13, 21
 goto, 14
 if, 12, 21
 namespace, 41
 prazna, 12
 prosta, 12
 return, 14
 složena, 12
 switch, 12, 21
 try, 194
 typedef, 8, 22
 upravljačka, 12
 using, 42
 while, 13, 21
 uređnik teksta, 4
 integrисано окружење за развој
 programa, 5
 C++ Builder, 5
 izvršавање programa, 5
 obrada programa, 4
 povezivanje prevedenog oblika
 programa, 4
 prevedenje izvornog teksta programa, 4
 sastavljanje izvornog teksta programa, 4
 skretanje glavnog ulaza i izlaza, 5
 multimap<>, klasa, 252
 konstruktor, 252
 nepromenljiv
 podatak, 7
 const, modifikator
 u jeziku C, 7
 u jeziku C++, 20
 pokazivač, 8
 nestrukturirani tip, 6
 new, operator, 34, 117
 bad_alloc, klasa za izuzetke,
 35
 <new>, zaglavlj. 35
 next_permutation<>, funkcija, 267
 niska, tip, 8
 '\0', 8
 <cstring>, zaglavlj. 47
 konstantna, 8

određeni integral
 metoda pravougaonika, 208
 primer sistema klasa, 204
 oduzimanje, -, 9
 ofstream, klasa, 285, 295
 konstruktor, 286
 ograničenja za
 automatsku konverziju za ne-standardne tipove, 107
 bezimene unije, 77
 definisanje tumačenja operatora, 101
 lokalne klase, 76
 metode lokalne klase, 76
 nasledivanje metoda, 149
 operatorske funkcije, 102
 polja unija, 77
 preklapanje operatora, 101
 unije, 77, 143
 oktalna celobrojna konstanta, 6
 open, metoda, 287
 operativni sistem
 MS-DOS, 4
 UNIX, 2
 Windows, 4
 operator, 9, 27
 !, 10
 !=, 9
 %, 9
 %, 10, 28
 &, binarni, 10
 &, unarni, 10, 55, 74
 &&, 10
 &, 10, 28
 (), 11
 preklapanje, 114
 (), redosled izvršavanja, 11
 (tip), 11
 *, binarni, 9
 *, unarni, 10, 55
 *=, 10, 28
 ,, 10
 ,, 11, 55
 ,, 74
 /, 9
 /=, 10, 28
 ::, binarni, 42, 56, 70
 ::, unarni, 44
 ?:, 10, 28
 [], 55
 preklapanje, 112
 ^, 10
 ^=, 10, 28
 ||, 10
 ||, 10
 |=, 10, 28
 ~, 10
 +, binarni, 9

operator (nastavak)
 - unarni, 9
 - -, 9, 28
 preklapanje, 104
 - -, 10, 28
 - binarni, 9
 - unarni, 9
 - -, 9, 28
 preklapanje, 104
 - -, 10, 28
 - -, 11, 55
 preklapanje, 116
 - -, 74
 - , 9
 - , 10
 izlazna konverzija, 31, 105,
 290
 preklapanje, 105
 - -, 10, 28
 - -, 9
 - , 10, 28, 55
 preklapanje, 109
 ==, 9
 >, 0
 >=, 9
 >-, 10
 preklapanje, 105
 ulazna konverzija, 31, 105,
 290
 >>=, 10, 28
 adresni, 10
 aritmetički, 9
cast, 106
const_cast, 30
defined, 17
 definisanje tumačenja, 101
delete, 34, 118
 preklapanje, 117
 dodele vrednosti, 10
dynamic_cast, 157
 grupisanje, 11, 27
 konverzija tipa, 11
 logički, 10
new, 34, 117
 preklapanje, 117
 operatorska funkcija, 101
 po bitovima, 10
 pozivanje funkcije, 11
 prioritet, 11, 27
 prvenstvo, 11, 27
 razrešenje dosegova, 42, 44, 56
 redosled izvršavanja, 11
reinterpret_cast, 29
 relacijski, 9
sizeof, 11
static_cast, 28
throw, 195
typeid, 158
 uslovni, 10

operator delete. metoda, 117, 118
operator delete[]. metoda, 117, 118
operator new. metoda, 117
operator new[]. metoda, 117
operator tip. metoda, 106
operator (). metoda, 114
operator []. metoda, 112
operator++, funkcija, 104
operator--, funkcija, 104
operator->. metoda, 116
operator<<, funkcija, 105
operator=. metoda, 110
 automatsko generisanje, 110
operator>>, funkcija, 105
 operatorska funkcija
 ograničenja, 102
 operator delete. 117, 118
 operator delete[]. 117, 118
 operator new. 117
 operator new[]. 117
 operator tip. 106
 operator (). 114
 operator []. 112
 operator++, 104
 operator--, 104
 operator->, 116
 operator<<, 105
 operator=, 110
 automatsko generisanje, 110
operator>>, 105
 za binarni operator, 102
 za nabrojane tipove, 118
 za unarni operator, 102
 opis tipa
 class, 54
 enum, 7
 struct, 9
 union, 9
 opseg vrednosti nabranja, 22
 oslobadanje memorije
 delete, operator, 34
 free, funkcija, 34
operator delete. metoda, 117, 118
operator delete[]. metoda, 117, 118
 u jeziku C, 34
 osnovna klasa, 142
 apstraktna
 klasa, 156
 metoda, 156
 dijagram klasa, 144
dynamic_cast. operator, 157
 javna, 143

P

osnovna klasa (*nastavak*)
 konverzija tipa
 iz izvedene klase, 150
 nadole, 152
 nagore, 152
 pokazivača, 151
 upućivača, 152
private. modifikator, 143
 privatna, 143
protected. modifikator, 143
public. modifikator, 143
virtual. modifikator
 za klase, 148
 za metode, 153
 virtuelna
 metoda, 152
 osnovna klasa, 148
 zaštićena, 143
 osnovni tip
 ceo broj, 6
 realan broj, 6
 ostatak deljenja, 1, 9
ostream klasa, 105, 285, 295
ostringstream klasa, 285, 295
 konstruktor, 288
 otvaranje datoteke
 open, metoda, 287
 out, konstanta, 287
 out_of_range, klasa za izuzetke,
 247, 255, 258, 259
overflow_error, klasa za izu-
 zetke, 259
 oznaka
 case, 12
 default, 12
 odredišta skoka, 14
 doseg, 16
private, 54, 142
protected, 142
public, 54, 142

parametar funkcije (*nastavak*)
 void, prazan niz parametara, 14
 volatile. modifikator, 14, 26
 parametar šablona, 218
 podrazumevana vrednost, 221
 parametri funkcije, 14
peek. metoda, 290
plus<, klasa, 239
 početna vrednost
 dinamičkog podatka, 17
 niza, 8
 podatak, 7, 21
 prolaznog podatka, 16
 strukture, 9
 trajnog podatka, 16
 unije, 9
 upućivača, 24
 podatak
 auto. modifikator, 16
 član klase, 53
const. modifikator
 u jeziku C, 7
 u jeziku C++, 20
 definisanje
 u jeziku C, 7
 u jeziku C++, 20
 dinamička dodela memorije, 34
 dinamički, 17
extern. modifikator, 16
 inicijalizacija, 7, 21
 u dinamičkoj zoni memorije,
 35
 inicijalizator, 7
 logički
 u jeziku C, 6
 u jeziku C++, 20
 lokalni za funkciju, 14
lvrednost, 11
 nepostojan, 7
 nepromenljiv, 7
 nestrukturiran, 6
 niz, 8
 početna vrednost, 8
 numerički, 6
 početna vrednost, 7, 21
 pokazivač, 8
 privremen, 16
 prolazan, 16
 promenljiv, 7
 prost, 6
register. modifikator, 16
 skalaran, 6
 složen, 6
static. modifikator, 16
 struktura, 9
 strukturiran, 6
 tip podataka, 6
 trajan, 16
 parametar funkcije (*nastavak*)
 void, prazan niz parametara, 14
 volatile. modifikator, 14, 26
 parametar šablona, 218
 podrazumevana vrednost, 221
 parametri funkcije, 14
peek. metoda, 290
plus<, klasa, 239
 početna vrednost
 dinamičkog podatka, 17
 generičke klase, 221
 podrazumevani konstruktor, 61
 automatsko generisanje, 61
 za proste tipove, 65
 podrazumevani način izvođenja
 klase, 143
 strukture, 143
 pojedinačan član klase, 69
 pokazivač
 konverzija tipa, 151
 na objekat, 55
 na polja klase, 74
 pokazivač, tip, 8
 &, operator, 10
 *, modifikator, 8
 ~, operator, 10
 *const, modifikator, 8
 *volatile, modifikator, 8
 dinamički tip pokazivača, 153
 generički, 8
 konverzija tipa
 nadole, 152
 nagore, 152
 konverzija tipa, 24
 na funkciju, 15
 nepostojan, 8
 nepromenljiv, 8
 NULL, konstanta, 8
 odnos prema upućivaču, 24
 pristup podatku, 10
 dinamičkom, 17
 promenljiv, 8
 statički tip pokazivača, 153
void*. tip, 8
 vrednost funkcije, 14
polar. funkcija, 241
 polimorfizam, 152
 dinamički tip pokazivača i
 upućivača, 153
dynamic_cast. operator, 157
 statički tip pokazivača i
 upućivača, 153
 virtuelna metoda, 154
 polimorfna metoda, 154
 polinomi, primer klase, 125
 polje
 klase, 53
 strukture, 9
 unije, 9
 polje klase, 53
 .. operator, 11
 .*., operator, 74
 ::., operator, 70
 ::.*., modifikator, 74
 ->, operator, 11
 ->*, operator, 74
 definisanje, 54
 zajedničkog polja, 70
 doseg, 16
 inicijalizacija, 60, 64, 149
 nepromenljivog polja, 65
 upućivačkog polja, 65
 zajedničkog polja, 70
 javno, 53
Klass::*, modifikator, 74
mutable. modifikator, 23
 ograničenja za unije, 77
 pojedinačno, 69
 pokazivač na polja, 74
 pristupanje, 70
 pomoću pokazivača na polja,
 74
 privatno, 53
private, oznaka, 54, 142
protected, oznaka, 142
public, oznaka, 54, 142
 spoljašnje povezivanje, 70
static. modifikator, 69
 uništavanje, 67, 149
 zajedničko, 69
 definisanje, 70
 trajnost, 70
 pomeranje binarne vrednosti, <<,
 >>, 10
pop. metoda, 251
pop_back. metoda, 247
pop_front. metoda, 249
 posredan pristup, 10
 *, operator, 10
 članu klase, 55
 potprogram, 14
 potpuna specijalizacija generičke
 funkcije, 223
 klase, 222
 povećavanje, ++, 9
 povezivanje
 funkcija pisanih na različitim
 jezicima, 40
 identifikator, 16
 spoljašnje, 16
 polja klase, 70
 unutrašnje, 16, 44
 prevedenog oblika programa, 3,
 4
pow. funkcija, 241
 pozicioniranje unutar toku
 podataka, 293

pozivanje
 destruktora
 automatsko, 66
 eksplicitno, 67
 funkcije, 11, 15
 `()`, operator, 11
 `operator()`, metoda, 114
 konstruktora, 59
 automatsko, 59
 eksplicitno, 60
 virtuelne metode, 153
 pravilo za
 izbor medu funkcijama s pre-
 klopljenim imenima, 38
 navodenje rukovalaca
 izuzecima, 197
 obrazovanje zaglavja, 161
 prazna naredba, 12
 precision, metoda, 292
 preklapanje imena funkcije, 38
 konstruktora, 59
 odnos prema virtuelnoj metodi,
 153
 operatorske funkcije, 102
 pravilo za izbor, 38
 preklapanje operatora, 102
 `()`, 114
 `(tip)`, 106
 `[]`, 112
 `++`, 104
 `--`, 104
 `->`, 116
 `<<`, 105
 `=`, 109
 `>>`, 105
 binarnog, 102
 delete, 117
 genericka klase za relacijske
 operatorere, 237
 new, 117
 ograničenja, 101
 unarnog, 102
 za nabrojani tip, 118
 prelazak-na-novi-list, beli znak, 6
 prelazak-u-novi-red, beli znak, 6
 preslikavanje, asocijativna zbirka
 podataka, 252
 pretprocessor, 17
 `#defined` operator, 17
 umetanje sadržaja datoteke, 17
 uslovno prevođenje, 17
 zamena leksičkih simbola, 17
 prev_permutation`<>`, funkcija,
 267
 prevođenje izvornog teksta
 programa, 4
 • prevodilac Borland C++, 4

prihvatanje izuzetka, 197
 tok, 197
 prijateljska funkcija, 72
 `friend`, modifikator, 72
 izvedene klase, 143
 odnos prema metodi, 72
 operatorska funkcija, 102
 prijateljska klasa, 72
 prijavljivanje izuzetka, 195
 primerak klase, 53
 printf, funkcija, 31
 prioritet operatora, 11, 27
 prioritet red, zbirka podataka, 250
 priority_queue`<>`, klasa, 250
 konstruktur, 251
 pristupanje
 članu izvedene klase, 145
 članu klase, 55
 članu lokalne klase, 76
 metodi, 55
 objektu, 55
 polju klase pomoću pokazivača
 na polja, 74
 zajedničkom članu klase, 70
 pristupanje članu, 11
 `.`, operator, 11
 `->`, operator, 11
 bezimene unije, 23
 privatni član klase, 53, 142
 private, oznaka, 54, 142
 private, modifikator, 143
 privatno izvođenje klase, 143
 private, modifikator, 143
 privremen podatak, 16
 argument funkcije, 107
 programski jezik
 C, 1
 C sa klasama, 1
 C++, 1
 prolazan podatak, 16
 auto, modifikator, 16
 početna vrednost, 16
 register, modifikator, 16
 promena predznaka, `-`, 9
 promenljiv
 podatak, 7
 pokazivač, 8
 prosta naredba, 12
 prost podatak, 6
 pokazivač, 8
 prost tip, 6
 definisanje podatka, 7
 konstruktur, 65

prostor imena, 41
 bezimeni, 44
 definisane, 41
 definisane podataka i funkcija
 izvan prostora imena, 44
 unutar prostora imena, 42
 deklarisanje podataka i funkcija
 unutar prostora imena, 42
 dohvatjanje identifikatora u
 prostoru imena, 42
 namespace, naredba, 41
 prostorski doseg, 41
 uklapanje, 45
 using, naredba, 42
 prostor imena, standardni, std, 46
 protected, modifikator, 143
 protected, oznaka, 142
 prototip funkcije, 15
 doseg identifikatora parametra, 16
 prototipski doseg, 16
 prvenstvo operatora, 11, 27
 public, modifikator, 143
 public, oznaka, 54, 142
 push, metoda, 251
 push_back, metoda, 247
 push_front, metoda, 249
 put, metoda, 34, 289
 putback, metoda, 290

Q

<queue>, zaglavje, 251
 queue`<>`, klasa, 250
 konstruktur, 251
 quick sort, metoda za uređivanje, 48

R

rand, funkcija, 50
 različito, `!=`, 9
 razmak, beli znak, 6
 razrešenje dosega
 `::`, operator
 binarni, 42, 56
 unarni, 44
 rbegin, metoda, 246
 read, metoda, 292
 real, funkcija, 241
 real, metoda, 241
 realan broj, tip, 6
 double, 6
 dvostruka tačnost, 6
 ekspONENT, 7
 float, 6
 jednostruka tačnost, 6
 konstanta, 7
 long double, 6
 mantisa, 7
 višestruka tačnost, 6

realan podatak, 6
 realloc, funkcija, 34
 red s dva kraja, zbirka podataka, 249
 red, zbirka podataka, 250
 redeclarisanje identifikatora unutar
 izvedene klase, 145
 uklopjenog dosega, 16
 redosled izračunavanja
 argumenata funkcija, 12
 operanada operatora, 12
 redovi, primer klase, 133
 register, modifikator, 16
 reinterpret_cast, operatror, 29
 relacijski operatör, 9
 `!=`, 9
 `<`, 9
 `<=`, 9
 `==`, 9
 `>`, 9
 `>=`, 9
 za iteratore, 243
 za klasu basic_string`<>`,
 256
 za klasu bitset`<>`, 258
 za klasu vector`<>`, 248
 za kompleksne brojeve, 241
 relativne datoteke, primer klase,
 296
 remove, metoda, 250
 remove`<>`, funkcija, 266
 remove_copy`<>`, funkcija, 266
 remove_copy_if`<>`, funkcija, 266
 remove_if, metoda, 250
 remove_if`<>`, funkcija, 266
 rend, metoda, 246
 replace, metoda, 257
 replace`<>`, funkcija, 265
 replace_copy_if`<>`, funkcija,
 265
 replace_copy`<>`, funkcija, 265
 replace_if`<>`, funkcija, 265
 reserve, metoda, 246
 reset, metoda, 259
 resize, metoda, 246
 return, naredba, 14
 reverse, metoda, 250
 reverse`<>`, funkcija, 266
 reverse_copy`<>`, funkcija, 266
 reverse_iterator, klasa, 244
 rezervisana reč, 6, 19
 rfind, metoda, 257
 right, konstanta, 291
 right, manipulator, 32
 rukovalac izuzecima, 193
 catch, službena reč, 195
 definisanje, 195
 izbor na osnovu tipa izuzetka,
 197

rukovalac izuzecima (nastavak)
 pravilo za redosled navodenja,
 197
 prihvatanje izuzetka, 197
 throw, operatror, 196
 univerzalni, 195
 rukovanje izuzecima, 194

S

sabiranje, `+`, 9
 sadržavanje, odnos medu klasama
 dijagram klasa, 78
 sastavljanje izvornog teksta pro-
 grama, 4
 scanf, funkcija, 31
 scientific, konstanta, 291
 scientific, manipulator, 33
 search`<>`, funkcija, 264
 seekg, metoda, 293
 seekp, metoda, 293
 sekvenca, upravljačka struktura, 12
 (), 12
 kao blok, 12
 prazna, 12
 sekvencialna zbirka podataka, 242
 lista, 249
 niz bitova, 258
 tekst, 254
 vektor, 245
 selekcija, upravljačka struktura, 12
 break, naredba, 13
 case, oznaka, 12
 default, oznaka, 12
 else, službena reč, 12
 if, naredba, 12, 21
 osnovna, 12, 21
 pomoću skretnice, 12, 21
 pravilo uklapanja naredbi if, 12
 switch, naredba, 12, 21
 uslovni preskok, 12
 set, metoda, 259
 <set>, zaglavje, 254
 set<>, klasa, 254
 set_terminate, funkcija, 198
 set_unexpected, funkcija, 198
 setbase, manipulator, 32
 setf, metoda, 290
 setfill, manipulator, 32
 setprecision, manipulator, 33
 setw, manipulator, 32
 short int, tip, 6
 unsigned, 6
 short, tip, 6
 unsigned, 6
 showbase, konstanta, 291
 showbase, manipulator, 32
 showpoint, konstanta, 291

showpoint, manipulator, 33
 showpos, konstanta, 291
 showpos, manipulator, 32
 signalizacija grešaka, 293
 signed char, tip, 6
 simbol, leksički, 6
 #define, direktiva, 17
 identifikator, 6
 konstanta, 6
 službena reč, 6
 #undef, direktiva, 17
 zamena, 17
 simbolička konstanta, 17, 20
 const, modifikator, 20
 #define, direktiva, 17
 nabranje
 u jeziku C, 7
 u jeziku C++, 22
 sin, funkcija, 241
 sinh, funkcija, 241
 size, metoda, 246
 size_t, tip, 117
 sizeof, operatror, 11
 skalaran podatak, 6
 skalaran tip, 6
 definisanje podatka, 7
 skipws, konstanta, 291
 skipws, manipulator, 32
 skok, upravljačka naredba, 13
 break, naredba, 13
 continue, naredba, 13
 goto, naredba, 14
 iz upravljačke strukture, 13
 na kraj ciklusa, 13
 oznaka oderđista skoka, 14
 povratak iz funkcije, 14
 return, naredba, 14
 s proizvoljnim odredištem, 14
 skretanje glavnog ulaza i izlaza, 3, 5
 skup znakova u jeziku C/C++, 6
 skup, zbirka podataka, 254
 skupovi, uredeni, primer klase, 84
 složena naredba, 12
 (), 12
 do, 13
 for, 13, 21
 if, 12, 21
 switch, 12, 21
 while, 13, 21
 složen tip, 6
 [], modifikator, 8
 class, opis tipa, 54
 klasa, 53
 niz, 8
 struct, opis tipa, 9
 struktura, 9
 unija, 9
 union, opis tipa, 9

službena reč, 6, 19
catch, 195
else, 12
template, 217
while, 13
smanjivanje, --, 9
sort, metoda, 250
sort<>, funkcija, 268
specifikacija, 222
 generičke funkcije
 potpuna, 223
 generičke klase, 222
 delimična, 222
 potpuna, 222
splice, metoda, 249
spoljašnje povezivanje
 extern, modifikator, 16
 identifikatora, 16
 polja klase, 70
sqr, funkcija, 241
<sstream>, zaglavlje, 285
stable_sort<>, funkcija, 268
<stack>, zaglavlje, 251
stack<>, klasa, 250
 konstruktor, 251
standard
 ANSI, za jezik C, 1
 za jezik C++, 2
standardna biblioteka funkcija i
 klasa, 237
 STL, 237
stanje objekta, 53
static, modifikator
 trajan podatak, 16
 unutrašnje povezivanje globalnih identifikatora, 16
 zajenički član klase, 69
static_cast, operador, 28
statički tip pokazivača i upućivača, 153
std, prostor imena, 46
<stddef.h>, zaglavlje, 117
<stdio.h>, zaglavlje, 18, 31, 286
<stdlib.h>, zaglavlje, 18
stek, zbirka podataka, 250
stekovi, primer klase, 90, 229
STL, biblioteka, 237
str, metoda, 288
streambuf, klasa, 295
string, tip u jeziku C, 8
<string>, zaglavlje, 254
<string.h>, zaglavlje, 18
string<>, klasa, 254
 konstruktor, 255
stringbuf, klasa, 295
stringstream, klasa, 285, 295
 konstruktor, 288

Š

šablon, 217
 <>, 218
 class, u šablonu, 218
 definisanje, 217
 mana, 218

struct, opis tipa, 9
struktura od bitova, tip, 9
struktura programa, 15
struktura, tip, 9, 76
 .. operator, 11
 (), početna vrednost, 9
 ->, operator, 11
 definisanje, 9
 dodela vrednosti, 10
doseg član, 16
identifikator
 u jeziku C, 9
 u jeziku C++, 22
inicijalizacija u dinamičkoj zoni
 memorije, 35
izvedena, 143
javno izvođenje, 143
odnos prema klasi, 76
početna vrednost, 9
podrazumevani način
 izvođenja, 143
polje strukture, 9
 mutable, modifikator, 23
pristup članu, 11
privatno izvođenje, 143
struct, opis tipa, 9
struktura od bitova, tip, 9
vrednost funkcije, 14
zaštićeno izvođenje, 143
strukturiran tip, 6
 klasa, 53
 niz, 8
 struktura, 9
 uniya, 9
struktorni doseg, 16
 odnos prema klasnom dosegu, 76
stvaran argument funkcije, 14
stvaranje
 objekta, 58
 konstruktor, 58, 59
toka podataka
 u memoriji, 288
 za datoteku, 286
substr, metoda, 256
swap, metoda, 248
swap<>, funkcija, 260
switch, naredba, 12, 21
 case, oznaka, 12
 default, oznaka, 12

T

tabulacija, beli znak, 6
tačke u ravnim, primer klase, 81
tan, funkcija, 241
tann, funkcija, 241
tekst, zbirka podataka, 254
tekući objekat, 56
 nepromenljiv i nepostojan, 57
telig, metoda, 293
teilp, metoda, 293
telo funkcije, 14
template, službena reč, 217
terminante, funkcija, 198
test, metoda, 259
this, pokazivač, 56
throw, operator, 195
 unutar rukovaoca izuzecima, 196
tip, 6
 &, modifikator, 24
 *, modifikator, 8
 *const, modifikator, 8
 *volatile, modifikator, 8
 ::*, modifikator, 74
 [], modifikator, 8
bool, 20
ceo broj, 6
 konstanta, 6
char, 6
class, opis tipa, 54
 definisanje, 8, 22
double, 6
enum, opis tipa, 7
float, 6
int, 6
izvedeni, 8
klasa, 53
Klase::*, modifikator, 74
konverzija tipa, 11, 28
 dinamička, 157
logical
 u jeziku C, 6
 u jeziku C++, 20
 bool, 20
logički
 false, konstanta, 20
 true, konstanta, 20
long, 6
long double, 6
long int, 6

šablon (nastavak)
 parametar šablonu, 218
 podrazumevana vrednost, 221
template, službena reč, 217
typename, u šablonu, 218

T

tabulacija, beli znak, 6
tačke u ravnim, primer klase, 81
tan, funkcija, 241
tann, funkcija, 241
tekst, zbirka podataka, 254
tekući objekat, 56
 nepromenljiv i nepostojan, 57
telig, metoda, 293
teilp, metoda, 293
telo funkcije, 14
template, službena reč, 217
terminante, funkcija, 198
test, metoda, 259
this, pokazivač, 56
throw, operator, 195
 unutar rukovaoca izuzecima, 196
tip (nastavak)
 nabranjane
 u jeziku C, 7
 u jeziku C++, 22
 nestrukturiran, 6
 niska, 8
 konstantni, 8
 niz, 8
 numerički, 6
 pokazivač, 8
 prost, 6
 realan broj, 6
 konstanta, 7
 short, 6
 short int, 6
 signed char, 6
 size_t, 117
 skalaran, 6
 složen, 6
 string, u jeziku C, 8
struct, opis tipa, 9
struktura, 9
 strukturiran, 6
typedef, naredba, 8
uniya, 9
union, opis tipa, 9
unsigned char, 6
unsigned int, 6
unsigned long, 6
unsigned long int, 6
unsigned short, 6
unsigned short int, 6
upućivač, 24
void*, 8
vrednosti funkcije, 14
to_string, metoda, 259
to_ulong, metoda, 259
tok izvršavanja
 destruktora, 67, 149
 konstruktora, 64, 149
tok podataka, 285
 <<, operator, 31, 105, 290
 >>, operator, 31, 105, 290
cerr, izlaz za poruke, 286
cin, glavni ulaz, 30, 105, 285
clog, glavni izlaz za zabeleške, 286
cout, glavni izlaz, 30, 105, 285
<cstdio>, zaglavlje, 47
datoteka
 binarna, 286
 stvaranje, 286
 tekstualna, 286
failure, klasa za izuzetke, 294
fstream, klasa, 285
 konstruktor, 286
<fstream>, zaglavlje, 285
ifstream, klasa, 285
tok podataka (nastavak)
 nabranjane
 u jeziku C, 7
 u jeziku C++, 22
 nestrukturiran, 6
 niska, 8
 konstantni, 8
 niz, 8
 numerički, 6
 pokazivač, 8
 prost, 6
 realan broj, 6
 konstanta, 7
 short, 6
 short int, 6
 signed char, 6
 size_t, 117
 skalaran, 6
 složen, 6
 string, u jeziku C, 8
ofstream, klasa, 285
 konstruktor, 286
operator<<, funkcija, 105
operator>>, funkcija, 105
ostream, klasa, 285
ostringstream, klasa, 285
 pozicioniranjem unutar toka, 293
signalizacija grešaka, 293
<sstream>, zaglavlje, 285
<stdio.h>, zaglavlje, 31
stringstream, klasa, 285
u memoriji, 288
 istringstream, klasa
 konstruktor, 288
 ostringstream, klasa
 konstruktor, 288
 stringstream, klasa
 konstruktor, 288
 stvaranje, 288
 ulazni, 30
 ulazno-izlazna konverzija, 31
za datoteke, 286
tok podataka u memoriji, 285, 288
<cstdio>, zaglavlje, 286
failure, klasa za izuzetke, 294
<iomanip>, zaglavlje, 32
<iostream>, zaglavlje, 285
istream, klasa, 285
istringstream, klasa, 285
 konstruktor, 288
manipulatori, 32
ostream, klasa, 285
ostringstream, klasa, 285
 konstruktor, 288
pristup sadržaju, 288
<sstream>, zaglavlje, 285
<stdio.h>, zaglavlje, 286
stringstream, klasa, 285
 konstruktor, 288
stvaranje, 288
top, metoda, 251
trajan podatak, 16
 početna vrednost, 16
static, modifikator, 16
 zajedničko polje, 70
trajnost podataka, 16
 dinamički podatak, 17
privremen podatak, 16
prolazan podatak, 16
trajan podatak, 16
 zajedničko polje, 70

U

ugradena funkcija, 36
 inline, modifikator, 36
 mana, 36
 metoda klase, 54
ugradivanje funkcije u kod, 36
 inline, modifikator, 36
 mana, 36
 metode klase, 54
uklapanje
 kласа, 75
 naredbi if, 12
 prostora imena, 45
ulaz podataka, 285
 >>, operator, 105
 cin, glavni ulaz, 285
<cstdio>, zaglavlje, 286
<iostream>, zaglavlje, 47
istream, klasa, 105
<stdio.h>, zaglavlje, 286
ulazna konverzija, 31
 >>, operator, 31, 105, 290
 <iomanip>, zaglavlje, 32
 manipulatori, 32
 operator>>, funkcija, 105
ulazni iterator, 243
ulazni tok podataka, 30, 285
 >>, operator, 31, 105, 290
 cin, glavni ulaz, 30, 105
datoteka
 binarna, 286
 stvaranje, 286
 zajedničko polje, 70
failure, klasa za izuzetke, 294
fstream, klasa, 285
 konstruktor, 286
<fstream>, zaglavlje, 285
ifstream, klasa, 285
transform<>, funkcija, 261
true, konstanta, 20
true, logička vrednost u jeziku C, 6
trunc, konstanta, 287
try, naredba, 194
 catch, službena reč, 194
 uklapanje, 195
type_info, klasa, 158
 != operator, 158
 == operator, 158
before, metoda, 159
name, metoda, 159
typedef, naredba, 8
 automatsko izvršavanje, 22
 za upućivače, 25
typeid, operador, 158
<typeinfo>, zaglavlje, 157, 158
typename, u šablonu, 218

ulazni tok podataka (*nastavak*)
 <iomanip>, zaglavije, 32
 <iostream>, zaglavije, 31, 285
 <iostream.h>, zaglavije, 47
 istream, klasa, 285
 manipulatori, 32
 operator>>, funkcija, 105
 pozicioniranje unutar toka, 293
 signalizacija grešaka, 293
 <stdio.h>, zaglavije, 31
 u memoriji, 288
 ulazna konverzija, 31
 za datoteke, 286
 umetanje sadržaja datoteka, 17
 #include, direktiva, 17
 unarni operator, preklapanje, 102
 #undef, direktiva, 17
 unexpected, funkcija, 198
 unija, tip, 9, 77
 .. operator, 11
 {}, početna vrednost, 9
 ->, operator, 11
 bezimena, 23
 destruktur, 77
 doseg člana, 16
 identifikator
 u jeziku C, 9
 u jeziku C++, 22
 konstruktor, 77
 odnos prema klasi, 77
 ograničenja, 77, 143
 početna vrednost, 9
 polje unije, 9
 pristup članu, 11
 stvaranje, 77
 union, opis tipa, 9
 uništavanje, 77
 vrednost funkcije, 14
 union, opis tipa, 9
 unique, metoda, 250
 unique<>, funkcija, 266
 unique_copy<>, funkcija, 266
 uništavanje objekta, 66
 destruktur, 66
 univerzalni rukovalac izuzecima, 195
 UNIX, operativni sistem, 2
 CC, komanda, 3
 izvršavanje programa, 3
 povezivanje prevedenog oblika
 programa, 3
 prevedenje izvornog teksta
 programa, 3
 sastavljanje izvornog teksta
 programa, 2
 skretanje glavnog ulaza i izlaza, 3
 vi
 komanda, 3
 urednik teksta, 2

unsetf, metoda, 291
 unsigned, tip, 6
 char, 6
 int, 6
 long, 6
 long int, 6
 short, 6
 short int, 6
 unutrašnja klasa, 75
 doseg identifikatora unutrašnje
 klase, 75
 generička, 225
 unutrašnje povezivanje
 identifikatora, 16
 bezimeni prostor imena, 44
 static, modifikator, 16
 uporedivanje nizova, primer klase,
 232
 upotreba
 unutrašnje klase, 75
 zajedničkog člana klase, 70
 upper_bound, metoda, 253
 upper_bound<>, funkcija, 268
 uppercase, konstanta, 291
 upravljačka naredba, 12, 13
 break, 13
 continue, 13
 goto, 14
 povratak iz funkcije, 14
 return, 14
 skok
 iz upravljačke strukture, 13
 oznaka odredišta skoka, 14
 s proizvođnjim odredištem, 14
 upravljačka struktura, 12
 {}, sekvenca, 12
 ciklus, 13
 s izlazom na dnu, 13
 s izlazom na vrhu
 generalizovani, 13, 21
 osnovni, 13, 21
 do, naredba, 13
 for, naredba, 13, 21
 if, naredba, 12, 21
 sekvenca, 12
 selekcija, 12
 osnovna, 12, 21
 pomoću skretnice, 12, 21
 switch, naredba, 12, 21
 while, naredba, 13, 21
 upućivač, tip, 24
 &, modifikator, 24
 definisanje, 24
 dinamički tip upućivača, 153
 inicijalizacija, 24
 konverzija tipa, 152
 nadole, 152
 nagore, 152
 virtual, modifikator, 153

upućivač, tip (*nastavak*)
 na objekat, 55
 odnos prema pokazivaču, 24
 parametar funkcije, 26
 statički tip upućivača, 153
 vrednost funkcije, 26
 uređivanje nizova, primer klase, 232
 urednik teksta
 EDIT, 4
 NOTEPAD, 4
 using, naredba, 42
 uslovni izraz, 10
 kao tvrdost, 28
 uslovno prevodenje, 17
 #defined, operator, 17
 #elif, direktiva, 17
 #else, direktiva, 17
 #endif, direktiva, 17
 #if, direktiva, 17
 #ifndef, direktiva, 17
 #ifndef, direktiva, 17
 uslužne funkcije, 18
 <cstdlib>, zaglavije, 47
 <stdlib.h>, zaglavije, 18
 <utility>, zaglavije, 237, 238
 V
 veće, >, 9
 veće ili jednako, >=, 9
 <vector>, zaglavije, 245
 vector<>, klasa, 245
 konstruktor, 245
 vector<bool>, klasa, 258
 vektor, zbirka podataka, 245
 vektori, primer klase, 199
 veličina, podatka, 11
 vertikalna-tabulacija, beli znak, 6
 vi
 komanda, 3
 urednik teksta, 2
 vidljivost identifikatora, 16
 virtual, modifikator
 za klase, 148
 za metode, 153
 virtuelna metoda, 152
 apstraktna, 156
 čista, 156
 destruktur, 154
 dinamički tip pokazivača i
 upućivača, 153
 odnos prema preklapanju imena
 funkcija, 153
 polimorfizam, 154
 pozivanje, 153
 statički tip pokazivača i
 upućivača, 153
 virtual, modifikator, 153

virtuelna osnovna klasa, 148
 virtual, modifikator, 148
 višestruka tačnost realnih brojeva, 6
 višestruko nasleđivanje članova
 klasa, 142
 virtueina osnovna klasa, 148
 void, tip
 prazan niz parametara funkcije,
 14
 vrednost funkcije, 14
 void*, tip, 8
 volatile, modifikator, 7
 parametra funkcije, 14, 26
 za tekući objekat metode, 57
 vrednost funkcije, 14
 metode klase, 54
 pokazivač, 14
 struktura, 14
 tip, 14
 unija, 14
 upućivač, 26
 vremenski intervali, primer klase,
 120
 Zaglavije, bibliotečko (*nastavak*)
 <iostream>, 31, 47, 105, 285
 <iostream.h>, 47
 <list>, 249
 <map>, 252
 <math.h>, 18
 <new>, 35
 <queue>, 251
 <set>, 254
 <stack>, 251
 <stddef.h>, 117
 <stdio.h>, 18, 31, 286
 <stdlib.h>, 18
 <string>, 254
 <string.h>, 18
 <typeinfo>, 157, 158
 <utility>, 237, 238
 <vector>, 245
 zaglavije, pravilo obrazovanja, 161
 zajednički član klase, 69
 metoda, 70
 polje, 69
 definisanje, 70
 pristupanje, 70
 upotreba, 70
 static, modifikator, 69
 zamena leksičkih simbola, 17
 #define, direktiva, 17
 #undef, direktiva, 17
 zarez (,), operator, 10
 zaštićen član klase, 142
 protected, oznaka, 142
 zaštićena osnovna klasa, 143
 protected, modifikator, 143
 zaštićeno izvođenje klase, 143
 protected, modifikator, 143
 zatvaranje datoteke
 close, metoda, 287
 is_open, metoda, 287
 zavisnost, odnos medu klasama, 79
 dijagram klasa, 78
 zbirka podataka, 241
 algoritam, bibliotečki
 nad pojedinačnim podacima,
 260
 obrada uredenih
 sekvensijalnih zbirki,
 267
 obrada zbirki, 265
 pretraživanje zbirki, 263
 znak
 '\0', 8
 beli, 6
 <cctype>, zaglavije, 47
 <ctype.h>, zaglavije, 18
 znak, tip
 char, 6
 konstanta
 u jeziku C, 7
 u jeziku C++, 20
 znakovna konstanta
 u jeziku C, 7
 u jeziku C++, 20

Izdavač
AKADEMSKA MISAO

Bul. kralja Aleksandra 73, Beograd
tel./fax: (+381 11) 3218 354

office@akademiska-misao.co.yu
www.akademiska-misao.co.yu

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд



004.432.2C++
004.42.045

Krause, Laslo L.

Programski jezik C++ : sa rešenim zadacima / Laslo Kraus. - 7. izd. - Beograd : Akademika misao, 2007 (Beograd : Planeta print). - XI, 327 str. : graf. prikazi ; 24 cm

Tiraž 500. - Bibliografija: str. 305. - Registar.

ISBN 978-86-7466-288-5

a) Programske jezike "C++" b) Objektno orijentisano programiranje
COBISS.SR-ID 140175372
