Boris Motik i Julijan Šribar

Demistificirani C++

Dobro upoznajte protivnika
da biste njime ovladali

Zagreb, 1997.

Roditeljima i bratu
Boris

Roditeljima, supruzi, a naroèito Dori i Mateju
Julijan

Sadržaj

2. To C++ or not to C++? 10

2.1. Povijesni pregled razvoja programskih jezika 10
2.2. Osnovna svojstva jezika C++ 12
2.3. Usporedba s C-om 14
2.4. Zašto primjer iz knjige ne radi na mom raèunalu? 16
2.5. Literatura 16
2.6. Zahvale 18

3. “Oluja” kroz jezik C++ 19

3.1. Što je program i kako ga napisati 19
3.2. Moj prvi i drugi C++ program 23
3.3. Moj treæi C++ program 26
3.4. Komentari 28
3.5. Rastavljanje naredbi 29

4. Osnovni tipovi podataka 33

4.1. Identifikatori 33
4.2. Varijable, objekti i tipovi 35
4.3. Operator pridruživanja 37
4.4. Tipovi podataka i operatori 38
4.4.1. Brojevi 38
4.4.2. Aritmetièki operatori 43
4.4.3. Operator dodjele tipa 48
4.4.4. Dodjeljivanje tipa brojèanim konstantama 49
4.4.5. Simbolièke konstante 50
4.4.6. Kvalifikator `volatile` 52
4.4.7. Pobrojenja 53
4.4.8. Logièki tipovi i operatori 55
4.4.9. Poredbeni operatori 56
4.4.10. Znakovi 58
4.4.11. Bitovni operatori 60
4.4.12. Operatori pridruživanja (2 ½) 67
4.4.13. Alternativne oznake operatora 68
4.4.14. Korisnièki definirani tipovi i operatori 69
4.4.15. Deklaracija `typedef` 69
4.5. Operator `sizeof` 71
4.6. Operator razdvajanja 72
4.7. Hijerarhija i redoslijed izvoðenja operatora 72

5. Naredbe za kontrolu toka programa 77

5.1. Blokovi naredbi 77
5.2. Grananje toka naredbom `if` 79
5.3. Uvjetni operator `? :` 83
5.4. Grananje toka naredbom `switch` 84
5.5. Petlja `for` 86
5.6. Naredba `while` 90
5.7. Blok `do-while` 92
5.8. Naredbe `break` i `continue` 93
5.9. Ostale naredbe za skok 94
5.10. O strukturiranju izvornog kôda 95
5.11. Kutak za buduæe C++ “gurue” 97

6. Polja, pokazivaèi, reference 101

6.1. Polja podataka 102
6.1.1. Jednodimenzionalna polja 102
6.1.2. Dvodimenzionalna polja 108
6.2. Pokazivaèi 114
6.2.1. Nul-pokazivaèi 119
6.2.2. Kamo sa zvijezdom (petokrakom) 120
6.2.3. Tajna veza između pokazivaèa i polja 121
6.2.4. Aritmetièke operacije s pokazivaèima 124
6.2.5. Dinamièko alociranje memorije operatorom `new` 127
6.2.6. Dinamièka alokacija polja 129
6.2.7. Dinamièka alokacija višedimenzionalnih polja 133
6.2.8. Pokazivaèi na pokazivaèe 134
6.3. Nepromjenjivi pokazivaèi i pokazivaèi na nepromjenjive objekte 136
6.4. Znakovni nizovi 139
6.4.1. Polja znakovnih nizova 143
6.5. Reference 144
6.5.1. Reference za hakere 145
6.6. Nevolje s pokazivaèima 147
6.7. Skraæeno oznaèavanje izvedenih tipova 148

7. Funkcije 150

7.1. Što su i zašto koristiti funkcije 150
7.2. Deklaracija i definicija funkcije 152
7.2.1. Deklaracije funkcija u datotekama zaglavlja 155
7.3. Tip funkcije 158
7.4. Lista argumenata 161
7.4.1. Funkcije bez argumenata 161
7.4.2. Prijenos argumenata po vrijednosti 161
7.4.3. Pokazivaè i referenca kao argument 164
7.4.4. Promjena pokazivaèa unutar funkcije 166
7.4.5. Polja kao argumenti 168
7.4.6. Konstantni argumenti 172
7.4.7. Podrazumijevani argumenti 173

7.4.8. Funkcije s neodreðenim argumentima 176
7.5. Pokazivaèi i reference kao povratne vrijednosti 179
7.6. Život jednog objekta 181
7.6.1. Lokalni objekti 181
7.6.2. Globalni objekti 182
7.6.3. Statièki objekti u funkcijama 186
7.7. Umetnute funkcije 187
7.8. Preoptereæenje funkcija 189
7.9. Rekurzija 195
7.10. Pokazivaèi na funkcije 196
7.11. Funkcija `main()` 202
7.12. Standardne funkcije 204
7.12.1. Funkcije za rukovanje znakovnim nizovima 206
7.12.2. Funkcija `exit()` 210
7.13. Predlošci funkcija 211
7.14. Pogled na funkcije “ispod haube” 212

8. Klase i objekti 215

8.1. Kako prepoznati klase? 215
8.2. Deklaracija klase 218
8.2.1. Podatkovni èlanovi 219
8.2.2. Dohvaæanje èlanova objekta 220
8.2.3. Funkcijski èlanovi 221
8.2.4. Kljuèna rijeè `this` 224
8.2.5. Umetnuti funkcijski èlanovi 225
8.2.6. Dodjela prava pristupa 226
8.2.7. Formiranje javnog suèelja korištenjem prava pristupa 228
8.2.8. Prijatelji klase 230
8.3. Deklaracija objekata klase 232
8.4. Stvaranje i uništavanje objekata 233
8.4.1. Konstruktor 233
8.4.2. Podrazumijevani konstruktor 239
8.4.3. Poziv konstruktora prilikom definiranja objekata 240
8.4.4. Konstruktor kopije 241
8.4.5. Inicijalizacija referenci i konstantnih èlanova 243
8.4.6. Konstruktori i prava pristupa 245
8.4.7. Destruktor 246
8.4.8. Globalni i statièki objekti 248
8.5. Polja objekata 249
8.6. Konstantni funkcijski èlanovi 253
8.7. Funkcijski èlanovi deklarirani kao `volatile` 257
8.8. Statièki èlanovi klase 257
8.8.1. Statièki podatkovni èlanovi 257
8.8.2. Statièki funkcijski èlanovi 260
8.9. Podruèje klase 262
8.9.1. Razluèivanje podruèja 264
8.10. Ugniježðene klase 265
8.11. Lokalne klase 270

8.12. Pokazivaèi na èlanove klase 271
8.12.1. Pokazivaèi na podatkovne èlanove 272
8.12.2. Pokazivaèi na funkcijske èlanove 277
8.13. Privremeni objekti 280
8.13.1. Eksplicitno stvoreni privremeni objekti 281
8.13.2. Privremeni objekti kod prijenosa parametara u funkciju 284
8.13.3. Privremeni objekti kod vraæanja vrijednosti 288

9. Strukture i unije 292

9.1. Struktura ili klasa? 292
9.2. Unije 293
9.3. Polja bitova 297

10. Preoptereæenje operatora 299

10.1. Korisnièki definirane konverzije 299
10.1.1. Konverzija konstruktorom 300
10.1.2. Eksplicitni konstruktori 302
10.1.3. Operatori konverzije 302
10.2. Osnove preoptereæenja operatora 307
10.3. Definicija operatorske funkcije 309
10.3.1. Operator `=` 313
10.3.2. Operator `[]` 315
10.3.3. Operator `()` 317
10.3.4. Operator `->` 318
10.3.5. Prefiks i postfiks operatori `++` i `--` 321
10.3.6. Operatori `new` i `delete` 323
10.4. Opæe napomene o preoptereæenju operatora 328

11. Nasljeðivanje i hijerarhija klasa 330

11.1. Ima li klasa bogatog strica u *Ameriki*? 330
11.2. Specificiranje nasljeðivanja 335
11.3. Pristup naslijeðenim èlanovima 341
11.4. Nasljeðivanje i prava pristupa 343
11.4.1. Zaštiæeno pravo pristupa 344
11.4.2. Javne osnovne klase 345
11.4.3. Privatne osnovne klase 347
11.4.4. Zaštiæene osnovne klase 349
11.4.5. Posebne napomene o pravima pristupa 350
11.4.6. Iskljuèivanje èlanova 353
11.5. Nasljeðivanje i pripadnost 354
11.6. Inicijalizacija i uništavanje izvedenih klasa 355
11.7. Standardne pretvorbe i nasljeðivanje 357
11.8. Podruèje klase i nasljeðivanje 360
11.8.1. Razlika nasljeðivanja i preoptereæenja 361
11.8.2. Ugniježðeni tipovi i nasljeðivanje 362
11.9. Klase kao argumenti funkcija 363
11.9.1. Toèno podudaranje tipova 364

11.9.2. Standardne konverzije 364
11.9.3. Korisnièki definirane pretvorbe 366
11.10. Nasljeðivanje preoptereæenih operatora....368
11.11. Principi polimorfizma 371
11.11.1. Virtualni funkcijski èlanovi....374
11.11.2. Poziv virtualnih funkcijskih èlanova....378
11.11.3. Èiste virtualne funkcije 380
11.11.4. Virtualni destruktori 380
11.12. Virtualne osnovne klase....382
11.12.1. Deklaracija virtualnih osnovnih klasa 384
11.12.2. Pristup èlanovima virtualnih osnovnih klasa 385
11.12.3. Inicijalizacija osnovnih virtualnih klasa....387

12. Predlošci funkcija i klasa 390

12.1. Uporabna vrijednost predložaka 390
12.2. Predlošci funkcija....391
12.2.1. Definicija predloška funkcije 392
12.2.2. Parametri predloška funkcije 394
12.2.3. Instanciranje predloška funkcije....397
12.2.4. Eksplicitna instantacija predloška 402
12.2.5. Preoptereæivanje predložaka funkcija....403
12.2.6. Specijalizacije predložaka funkcija 405
12.2.7. Primjer predloška funkcije za *bubble sort* 406
12.3. Predlošci klasa 408
12.3.1. Definicija predloška klase 409
12.3.2. Instanciranje predložaka klasa 412
12.3.3. Eksplicitna instantacija predložaka klasa....416
12.3.4. Specijalizacije predložaka klasa 416
12.3.5. Predlošci klasa sa statièkim èlanovima 419
12.3.6. Konstantni izrazi kao parametri predložaka 420
12.3.7. Predlošci i ugniježðeni tipovi....422
12.3.8. Ugniježðeni predlošci....423
12.3.9. Predlošci i prijatelji klasa....426
12.3.10. Predlošci i nasljeðivanje 427
12.4. Mjesto instantacije 429
12.5. Realizacija klase `Lista` predloškom....431

13. Imenici 434

13.1. Problem podruèja imena....434
13.2. Deklaracija imenika 435
13.3. Pristup elementima imenika....437
13.3.1. Deklaracija `using` 439
13.3.2. Direktiva `using`....443

14. Rukovanje iznimkama 446

14.1. Što su iznimke? 446
14.2. Blokovi pokušaja i hvatanja iznimaka 448
14.3. Tijek obrade iznimaka....451

14.4. Detaljnije o kljuènoj rijeèi `catch` 453
14.5. Navoðenje liste moguæih iznimaka 458
14.6. Obrada pogrešaka konstruktora 460

15. Identifikacija tipa tijekom izvoðenja 463

15.1. Statièki i dinamièki tipovi 463
15.2. Operator `typeid` 465
15.3. Sigurna pretvorba 467
15.4. Ostali operatori konverzije 471
15.4.1. Promjena konstantnosti objekta 471
15.4.2. Statièke dodjele 471

16. Pretprocesorske naredbe 474

16.1. U poèetku bijaše pretprocesor 474
16.2. Naredba `#include` 475
16.3. Naredba `#define` 476
16.3.1. Trajanje definicije 477
16.3.2. Rezervirana makro imena 478
16.3.3. Makro funkcije 479
16.3.4. Operatori za rukovanje nizovima 480
16.4. Uvjetno prevoðenje 481
16.4.1. Primjena uvjetnog prevoðenja za pronalaženje pogrešaka 483
16.5. Ostale pretprocesorske naredbe 484
16.6. Ma èa æe meni pretprocesor? 485

17. Organizacija kôda u složenim programima 486

17.1. Zašto u više datoteka? 486
17.2. Povezivanje 487
17.2.1. Specifikatori `static` i `extern` 491
17.3. Datoteke zaglavlja 494
17.4. Organizacija predložaka 497
17.5. Primjer raspodjele deklaracija i definicija u više datoteka 501
17.6. Povezivanje s kôdom drugih programskih jezika 508
17.6.1. Poziv C funkcija iz C++ kôda 509
17.6.2. Ukljuèivanje asemblerskog kôda 511

18. Ulazni i izlazni tokovi 513

18.1. Što su tokovi 513
18.2. Biblioteka `iostream` 515
18.3. Stanje toka 516
18.4. Ispis pomoæu `cout` 517
18.4.1. Operator umetanja `<<` 517
18.4.2. Ispis korisnièki definiranih tipova 519
18.4.3. Ostali èlanovi klase `ostream` 520
18.5. Uèitavanje pomoæu `cin` 521

18.5.1. Uèitavanje pomoæu operatora >> 521
18.5.2. Uèitavanje korisnièki definiranih tipova....523
18.5.3. Uèitavanje znakovnih nizova 524
18.5.4. Ostali èlanovi klase `istream`....526
18.6. Kontrola uèitavanja i ispisa....529
18.6.1. Vezivanje tokova 529
18.6.2. Širina ispisa 530
18.6.3. Popunjavanje praznina 531
18.6.4. Zastavice za formatiranje....532
18.6.5. Formatirani prikaz realnih brojeva 535
18.6.6. Manipulatori....537
18.7. Datoteèni ispis i uèitavanje....541
18.7.1. Klase `ifstream` i `ofstream` 541
18.7.2. Otvaranje i zatvaranje datoteke....544
18.7.3. Klasa `fstream`....546
18.7.4. Odreðivanje i postavljanje položaja unutar datoteke 548
18.7.5. Binarni zapis i uèitavanje....550
18.8. Tokovi vezani na znakovne nizove....553
18.9. Ulijeva li se svaki tok u more? 554

19. Principi objektno orijentiranog dizajna.... 555

19.1. Zašto uopæe C++?....555
19.2. Objektna paradigma 556
19.3. Ponovna iskoristivost....558
19.4. Korak 1: Pronalaženje odgovarajuæe apstrakcije 559
19.5. Korak 2: Definicija apstrakcije....560
19.5.1. Definicija ekrana 560
19.5.2. Definicija prozora....562
19.5.3. Definicija izbornika....562
19.6. Korak 3: Definicija odnosa i veza izmeðu klasa....563
19.6.1. Odnosi objekata u korisnièkom suèelju 565
19.7. Korak 4: Definicija implementacijski zavisnih apstrakcija 569
19.8. Korak 5: Definicija suèelja 571
19.9. Korak 6: Implementacija 578

2. To C++ or not to C++?

Više nego bilo kada u povijesti, čovječanstvo se nalazi na razmeđi. Jedan put vodi u očaj i krajnje beznađe, a drugi u potpuno istrebljenje. Pomolimo se da ćemo imati mudrosti izabrati ispravno.

Woody Allen, "Side Effects" (1980)

Prilikom pisanja ove knjige mnogi su nas, s podsmijehom kao da gledaju posljednji primjerak snježnog leoparda, pitali: "No zašto se bavite C++ jezikom? To je kompliciran jezik, spor, nedovoljno efikasan za primjenu u komercijalnim programima. Na kraju, ja to sve mogu napraviti u obiènom C-u." Prije nego što poènemo objašnjavati pojedine znaèajke jezika, nalazimo važnim pokušati dati koliko-toliko suvisle odgovore na ta i sliène pitanja.

Dakle, osnovno pitanje je što C++ čini boljim i pogodnijim općenamjenskim jezikom za pisanje programa, od operacijskih sustava do baza podataka. Da bismo to razumjeli, pogledajmo kojim putem je tekao povijesni razvoj jezika. Na taj način će možda biti jasnija motivacija Bjarne Stroustrupa, "oca i majke" jezika C++.

2.1. Povijesni pregled razvoja programskih jezika

Prva raèunala koja su se pojavila bila su vrlo složena za korištenje. Njih su koristili iskljuèivo struènjaci koji su bili osposobljeni za komunikaciju s raèunalom. Ta komunikacija se sastojala od dva osnovna koraka: davanje uputa raèunalu i èitanje rezultata obrade. I dok se èitanje rezultata vrlo brzo uèinilo koliko-toliko snošljivim uvođenjem pisaèa na kojima su se rezultati ispisivali, unošenje uputa – programiranje – se sastojalo od mukotrpnog unosa niza nula i jedinica. Ti nizovi su davali raèunalu upute kao što su: "zbroji dva broja", "premjesti podatak s neke memorijske lokacije na drugu", "skoèi na neku instrukciju izvan normalnog slijeda instrukcija" i slièno. Kako je takve programe bilo vrlo složeno pisati, a još složenije èitati i ispravljati, ubrzo su se pojavili prvi programerski alati nazvani *asembleri* (engl. *assemblers*).

U asemblerskom jeziku svaka strojna instrukcija predstavljena je mnemonikom koji je razumljiv ljudima koji čitaju program. Tako se zbrajanje najčešće obavlja mnemonikom `ADD`, dok se premještanje podataka obavlja mnemonikom `MOV`. Time se postigla bolja čitljivost programa, no i dalje je bilo vrlo složeno pisati programe i ispravljati ih jer je bilo potrebno davati sve, pa i najmanje upute računalu za svaku pojedinu operaciju. Javlja se problem koji će kasnije, nakon niza godina, dovesti i do

pojave C++ programskog jezika: potrebno je razviti programerski alat koji će osloboditi programera rutinskih poslova te mu dopustiti da se usredotoči na problem koji rješava.

Zbog toga se pojavljuje niz viših programska jezika, koji preuzimaju na sebe neke "dosadne" programerske poslove. Tako je FORTRAN bio posebno pogodan za matematičke proračune, zatim BASIC koji se vrlo brzo učio, te COBOL koji je bio u pravilu namijenjen upravljanju bazama podataka.

Oko 1972. se pojavljuje jezik C, koji je direktna preteča današnjeg jezika C++. To je bio prvi jezik opće namjene te je postigao neviđen uspjeh. Više je razloga tome: jezik je bio jednostavan za učenje, omogućavao je modularno pisanje programa, sadržavao je samo naredbe koje se mogu jednostavno prevesti u strojni jezik, davao je brzi kôd. Jezik nije bio opterećen mnogim složenim funkcijama, kao na primjer *skupljanje smeća* (engl. *garbage collection*): ako je takav podsustav nekome trebao, korisnik ga je sam napisao. Jezik je omogućavao vrlo dobru kontrolu strojnih resursa te je na taj način omogućio programerima da optimiziraju svoj kôd. Do unatrag nekoliko godina, 99% komercijalnih programa bili su pisani u C-u, ponegdje dopunjeni odsječcima u strojnom jeziku kako bi se kritični dijelovi sustava učinili dovoljno brzima.

No kako je razvoj programske podrške napredovao, stvari su se i na području programskih jezika počele mijenjati. Složeni projekti od nekoliko stotina tisuća, pa i više redaka više nisu rijetkost, pa je zbog toga bilo potrebno uvesti dodatne mehanizme kojima bi se takvi programi učinili jednostavnijima za izradu i održavanje, te kojima bi se omogućilo da se jednom napisani kôd iskoristi u više različitih projekata.

Bjarne Stroustrup je započeo 1979. godine rad na jeziku "C s klasama" (engl. *C with Classes*). Prije toga, u *Computing Laboratory of Cambridge* on je radio na svom doktoratu te istraživao distribuirane sustave: granu računalne znanosti u kojoj se proučavaju modeli obrade podataka na više jedinica istodobno. Pri tome koristio se jezikom Simula, koji posjedovao neka važna svojstva koja su ga činila prikladnim za taj posao. Na primjer, Simula je posjedovala pojam klase: strukture podataka koje objedinjavaju podatke i operacije nad podacima. Korištenje klasa omogućilo je da se koncepti problema koji se rješava izraze direktno pomoću jezičnih konstrukcija. Dobiveni kôd je bio vrlo čitljiv i razumljiv, a g. Stroustrup je bio posebno fasciniran načinom na koji je sam programski jezik upućivao programera u razmišljanju o problemu. Također, jezik je posjedovao sustav tipizacije, koji je često pomagao korisniku u pronalaženju pogrešaka već prilikom prevođenja.

No naoko idealan u teoriji, jezik Simula je posrnuo u praksi: prevođenje je bilo iznimno dugotrajno, a dobiveni kôd se izvodio ekstremno sporo. Dobiveni program je bio neupotrebljiv, i da bi ipak pošteno zaradio svoj doktorat, gospodin Stroustrup se morao potruditi i ponovo napisati cjelokupan program u jeziku BCPL – jeziku niske razine koji je omogućio vrlo dobre performanse prevedenog programa. No iskustvo pisanja složenog programa u takvom jeziku je bilo užasno i g. Stroustrup je, po završetku svog posla na Cambridgeu, čvrsto sebi obećao da više nikada neće takav složen problem pokušati riješiti neadekvatnim alatima poput BCPL-a ili Simule.

Kada se 1979. zaposlio u *Bell Labs* u Murray Hillu, započeo je rad na onome što će kasnije postati C++. Pri tome je iskoristio svoje iskustvo stečeno prilikom rada na doktoratu te pokušao stvoriti univerzalni jezik koji će udovoljiti današnjim zahtjevima.

Pri tome je uzeo dobra svojstva niza jezika: Simula, Clu, Algol68 i Ada, a kao osnovu je uzeo jezik C.

2.2. Osnovna svojstva jezika C++

Èetiri su važna svojstva jezika C++ koja ga èine objektno orijentiranim: *enkapsulacija* (engl. *encapsulation*), *skrivanje podataka* (engl. *data hiding*), *nasljeðivanje* (engl. *inheritance*) i *polimorfizam* (engl. *polymorphism*). Sva ta tri svojstva doprinose ostvarenju takozvane objektno orijentirane paradigme programiranja.

Da bismo to bolje razumjeli, pogledajmo koji su se programski modeli koristili u prošlosti. Pri tome je svakako najvažniji model proceduralno strukturiranog programiranja.

Proceduralno programiranje se zasniva na promatranju programa kao niza jednostavnih programskih odsječaka: procedura. Svaka procedura je konstruirana tako da obavlja jedan manji zadatak, a cijeli se program sastoji od niza procedura koje međusobno sudjeluju u rješavanju zadatka.

Kako bi koristi od ovakve podjele programa bile što izraženije, smatrano je dobrom programerskom taktikom odvojiti proceduru od podataka koje ona obrađuje: time je bilo moguće pozvati proceduru za različite ulazne podatke i na taj način iskoristiti je na više mjesta. Strukturirano programiranje je samo dodatak na proceduralni model: ono definira niz osnovnih jezičnih konstrukcija, kao petlje, grananja i pozive procedura koje unose red u programe i čine samo programiranje daleko jednostavnijim.

Princip kojim bismo mogli obilježiti proceduralno strukturirani model jest *podijeli-pa-vladaj*: cjelokupni program je presložen da bi ga se moglo razumjeti pa se zbog toga on rastavlja na niz manjih zadataka – procedura – koje su dovoljno jednostavne da bi se mogle izraziti pomoću naredbi programskog jezika. Pri tome, pojedina procedura također ne mora biti riješena monolitno: ona može svoj posao obaviti kombinirajući rad niza drugih procedura.

Ilustrirat ćemo to primjerom: zamislimo da želimo izraditi kompleksan program za obradu trodimenzionalnih objekata. Kao jednu od mogućnosti koje moramo ponuditi korisnicima jest rotacija objekata oko neke točke u prostoru. Koristeći proceduralno programiranje, taj zadatak bi se mogao riješiti ovako:

1. Listaj sve objekte redom.
2. Za pojedini objekt odredi njegov tip.
3. Ovisno o tipu, pozovi ispravnu proceduru koja će izračunati novu poziciju objekta.
4. U skladu s tipom podataka ažuriraj koordinate objekta.

Operacije odreðivanja tipa, izraèunavanje nove pozicije objekta i ažuriranje koordinata se mogu dalje predstaviti pomoæu procedura koje sadržavaju niz jednostavnijih akcija.

Ovakav programski pristup je bio vrlo uspješan do kasnih osamdesetih, kada su njegovi nedostaci postajali sve očitiji. Naime, odvajanje podataka i procedura čini programski kôd težim za čitanje i razumijevanje. Prirodnije je o podacima razmišljati preko operacija koje možemo obaviti nad njima – u gornjem primjeru to znači da o

kocki ne razmišljamo pomoću koordinata njenih kutova već pomoću mogućih operacija, kao što je rotacija kocke.

Nadalje, pokazalo se složenim istodobno razmišljati o problemu i odmah strukturirati rješenje. Umjesto rješavanja problema, programeri su mnogo vremena provodili pronalazeći načine da programe usklade sa zadanom strukturom.

Također, današnji programi se pokreću pomoću miša, prozora, izbornika i dijaloga. Programiranje je *pogonjeno događajima* (engl. *event-driven*) za razliku od starog, sekvencijalnog načina. Proceduralni programi su korisniku u pojedinom trenutku prikazivali niz ekrana nudeći mu pojedine opcije u određenom redoslijedu. No *pogonjeno događajima* znači da se program ne odvija po unaprijed određenom slijedu, već se programom upravlja pomoću niza događaja. Događaja ima raznih: pomicanje miša, pritisak na tipku, izbor stavke iz izbornika i slično. Sada su sve opcije dostupne istodobno, a program postaje interaktivan, što znači da promptno odgovara na korisnikove zahtjeve i odmah (ovo ipak treba uvjetno shvatiti) prikazuje rezultat svoje akcije na zaslonu računala.

Kako bi se takvi zahtjevi jednostavnije proveli u praksi, razvijen je objektni pristup programiranju. Osnovna ideja je razbiti program u niz zatvorenih cjelina koje zatim međusobno surađuju u rješavanju problema. Umjesto specijaliziranih procedura koje barataju podacima, radimo s objektima koji objedinjavaju i operacije i podatke. Pri tome je važno što objekt radi, a ne kako on to radi. Jedan objekt se može izbaciti i zamijeniti drugim, boljim, ako oba rade istu stvar.

Ključ za postizanje takvog cilja jest spajanje podataka i operacija, poznato pod nazivom enkapsulacija. Također, podaci su privatni za svaki objekt te ne smiju biti dostupni ostalim dijelovima programa. To svojstvo se naziva skrivanje podataka. Svaki objekt svojoj okolini pruža isključivo podatke koji su joj potrebni da bi se objekt mogao iskoristiti. Korisnik se ne mora zamarati razmišljajući o načinu na koji objekt funkcionira – on jednostavno traži od objekta određenu uslugu.

Kada PC preprodavač, vlasnik poduzeća “Taiwan/tavan-Commerce” sklapa računalo, on zasigurno treba kućište (iako se i to ponekad pokazuje nepotrebnim). No to ne znači da će on morati početi od nule (miksajući atome željeza u čašici od Kinderlade); on će jednostavno otići kod susjednog dilera i kupiti gotovo kućište. Tako je i u programiranju: moguće je kupiti gotove programske komponente koje se zatim mogu iskoristiti u programu. Nije potrebno razumjeti kako komponenta radi – potrebno je jedino znati ju iskoristiti.

Također, kada projektanti u Renaultu žele izraditi novi model automobila, imaju dva izbora: ili mogu početi od nule i ponovo proračunavati svaki najmanji dio motora, šasije i ostalih dijelova, ili mogu jednostavno novi model bazirati na nekom starom modelu. Kako je kompaniji vrlo vjerojatno u cilju što brže razviti novi model kako bi pretekla konkurenciju, gotovo sigurno će jednostavno uzeti uspješan model automobila i samo izmijeniti neka njegova svojstva: promijenit će mu liniju, pojačati motor, dodati ABS kočnice. Slično je i s programskim komponentama: prilikom rješavanja nekog problema možemo uzdahnuti i početi kopati, ili možemo uzeti neku već gotovu komponentu koja je blizu rješenja i samo dodati nove mogućnosti. To se zove *ponovna iskoristivost* (engl. *reusability*) i vrlo je važno svojstvo. Za novu programsku

komponentu kaže se da je *naslijedila* (engl. *inherit*) svojstva komponente iz koje je izgrađena.

Korisnik koji kupuje auto sigurno neće biti presretan ako se njegov novi model razlikuje od starog po načinu korištenja: on jednostavno želi pritisnuti gas, a stvar je nove verzije automobila primjerice kraće vrijeme ubrzanja od 0 do 100 km/h. Slično je i s programskim komponentama: korisnik se ne treba opterećivati time koju verziju komponente koristi – on će jednostavno tražiti od komponente uslugu, a na njoj je da to obavi na adekvatan način. To se zove *polimorfizam* (engl. *polimorphism*).

Gore navedena svojstva zajedno sačinjavaju objektno orijentirani model programiranja. Evo kako bi se postupak rotiranja trodimenzionalnih likova proveo koristeći objekte:

1. Listaj sve objekte redom.
2. Zatraži od svakog objekta da se zarotira za neki kut.

Sada glavni program više ne mora voditi raèuna o tome koji se objekt rotira – on jednostavno samo zatraži od objekta da se zarotira. Sam objekt zna to uèiniti ovisno o tome koji lik on predstavlja: kocka æe se zarotirati na jedan naèin, a kubièni spline na drugi. Takoðer, ako se bilo kada kasnije program proširi novim likovima, nije potrebno mijenjani program koji rotira sve objekte – samo je za novi objekt potrebno definirati operaciju rotacije.

Dakle, ono što C++ jezik čini vrlo pogodnim jezikom opće namjene za izradu složenih programa jest mogućnost jednostavnog uvođenja novih tipova te naknadnog dodavanja novih operacija.

2.3. Usporedba s C-om

Mnogi okorjeli C programeri, koji sanjaju strukture i dok se voze u tramvaju ili razmišljaju o tome kako æe svoju novu rutinu riješiti pomoæu pokazivaèa na funkcije, dvoume se oko toga je li C++ doista dostojan njihovog kôda: mnogi su u strahu od nepoznatog jezika te se boje da æe im se njihov supermunjeviti program za raèunanje nekog fraktalnog skupa na novom jeziku biti sporiji od programa za zbrajanje dvaju jednoznamenkastih brojeva. Drugi se, pak, kunu da je C++ odgovor na sva njihova životna pitanja, te u fanatiènom zanosu umjesto Kristovog roðenja slave roðendan gospodina Stroustrupa.

Moramo odmah razočarati obje frakcije: niti jedna nije u pravu te je njihovo mišljenje rezultat nerazumijevanja nove tehnologije. Kao i sve drugo, objektna tehnologija ima svoje prednosti i mane, a također nije svemoguća te ne može riješiti sve probleme (na primjer, ona vam neće pomoći da opljačkate banku i umaknete Interpolu).

Kao prvo, C++ programi nisu nužno sporiji od svojih C ekvivalenata. No u pojedinim slučajevima oni doista mogu biti sporiji, a na programeru je da shvati kada je to dodatno usporenje prevelika smetnja da bi se toleriralo.

Nadalje, koncept klase i enkapsulacija uopće ne usporavaju dobiveni izvedbeni program. Dobiveni strojni kôd bi u mnogim slučajevima trebao biti potpuno istovjetan onome koji će se dobiti iz analognog C programa. Funkcijski članovi pristupaju

podatkovnim članovima objekata preko pokazivača, na sličan način na koji to korisnici proceduralne paradigme čine ručno. No C++ kôd će biti čitljiviji i jasniji te će ga biti lakše napisati i razumjeti. Ako pojedini prevoditelj i daje lošiji izvedbeni kôd, to je posljedica lošeg prevoditelja, a ne mana jezika.

Također, korištenje nasljeđivanja ne usporava dobiveni kôd ako se ne koriste virtualni funkcijski članovi i virtualne osnovne klase. Nasljeđivanje samo ušteđuje programeru višestruko pisanje kôda te olakšava ponovno korištenje već napisanih programskih odsječaka.

Virtualne funkcije i virtualne osnovne klase, naprotiv, mogu unijeti značajno usporenje u program. Korištenje virtualnih funkcija se obavlja tako da se prije poziva konzultira posebna tablica, pa je jasno da će poziv svake takve funkcije biti sporiji. Također, pristup članovima virtualnih osnovnih klasa se redovito obavlja preko jednog pokazivača više. Na programeru je da odredi hoće li koristiti "inkriminirana" svojstva jezika ili ne. Da bi se precizno ustanovilo kako djeluje pojedino svojstvo jezika na izvedbeni kôd, nije dobro nagađati i kriviti C++ za loše performanse, već izmjeriti vrijeme izvođenja te locirati problem.

No razmislimo o još jednom problemu: asembler je jedini jezik u kojemu programer točno zna što se dešava u pojedinom trenutku prilikom izvođenja programa. Kako je programiranje u asembleru bilo složeno, razvijeni su viši programski jezici koji to olakšavaju. Prevedeni C kôd također nije maksimalno brz – posebno optimiran asemblerski kôd će sigurno dati bolje rezultate. No pisanje takvog kôda danas jednostavno nije moguće: problemi koji se rješavaju su ipak previše složeni da bi se mogli rješavati tako da se pazi na svaki ciklus procesora. Složeni problemi zahtijevaju nove pristupe njihovom rješavanju: zbog toga imamo na raspolaganju nova računala s većim mogućnostima koja su sposobna učiniti gubitak performansi beznačajnim u odnosu na dobitak u brzini razvoja programa.

Slično je i s objektnom tehnologijom: možda će i dobiveni kôd biti sporiji i veći od ekvivalentnog C kôda, no jednostavnost njegove izrade će sigurno omogućiti da dobiveni program bude bolji po nizu drugih karakteristika: bit će jednostavnije izraditi program koji će biti lakši za korištenje, s više mogućnosti i slično. Uostalom, manje posla – veća zarada! (Raj zemaljski!) Tehnologija ide naprijed: dok se gubi neznatno na brzini i memorijskim zahtjevima, dobici su višestruki.

Također, C++ nije svemoćan. Korištenje objekata neće napisati pola programa umjesto vas: ako želite provesti crtanje objekata u tri dimenzije i pri tome ih realistično osjenčati, namučit ćete se pošteno koristite li C ili C++. To niti ne znači da će poznavanje objektne tehnologije jamčiti da ćete ju i ispravno primijeniti: ako se ne potrudite prilikom izrade klase te posao ne obavite u duhu objektnog programiranja, neće biti ništa od ponovne iskoristivosti kôda. Čak i ako posao obavite ispravno, to ne znači da jednog dana nećete naići na problem u kojem će jednostavno biti lakše zaboraviti sve napisano i početi od jajeta.

Ono što vam objektna tehnologija pruža jest mogućnost da manje pažnje obratite jeziku i načinu na koji ćete svoju misao izraziti, a da se usredotočite na ono što zapravo želite učiniti. U gornjem slučaju trodimenzionalnog crtanja objekata to znači da ćete manje vremena provesti razmišljajući gdje ste pohranili podatak o položaju kamere koji

vam baš sad treba, a više ćete razmišljati o tome kako da ubrzate postupak sjenčanja ili kako da ga učinite realističnijim.

Objektna tehnologija je pokušala dati odgovore na neke potrebe ljudi koji rješavaju svoje zadatke računalom; na vama je da procijenite koliko je to uspješno, a u svakom slučaju da prije toga pročitate ovu knjigu do kraja i preporučite ju prijateljima, naravno. *Jer tak' dobru i guba knjigu niste vidli već sto godina i baš vam je bilo fora ju čitat*.

2.4. Zašto primjer iz knjige ne radi na mom raèunalu?

Zahvaljujuæi velikoj popularnosti jezika C, programski jezik C++ se brzo proširio i vrlo su se brzo pojavili mnogi komercijalno dostupni prevoditelji. Od svoje pojave, jezik C++ se dosta mijenjao, a prevoditelji su "kaskali" za tim promjenama. Zbog toga se redovito dogaðalo da je program koji se dao korektno prevesti na nekom prevoditelju, bio neprevodiv veæ na sljedeæoj inaèici prevoditelja istog proizvoðaèa.

Pojavom završnog nacrta standarda je taj dinamički proces zaustavljen. Stoga bi bilo logično očekivati da se svi prevoditelji koji su se na tržištu pojavili nakon travnja 1995. ponašaju u skladu sa standardom. Nažalost, to nije slučaj, tako da vam se može lako dogoditi da i na najnovijem prevoditelju nekog proizvođača (imena nećemo spominjati) ne možete prevesti kôd iz knjige ili da vam prevedeni kôd radi drukčije od onoga što smo mi napisali.

Naravno (unaprijed se posipamo pepelom), ne otklanjamo mogućnost da smo i mi napravili pogrešku. Većinu primjera smo istestirali pomoću prevoditelja koji je uglavnom usklađen sa standardom, ... ali nikad se ne zna. Oznaku prevoditelja nećemo navoditi da nam netko ne bi prigovorio da objavljujemo (strane) plaćene reklame (a ne zato jer bi se radilo o piratskoj kopiji – kopija je legalno kupljena i licencirana).

2.5. Literatura

Tijekom pisanja knjige koristili smo sljedeæu literaturu (navodimo ju abecednim slijedom autora):

[ANSI95] *Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++*, Technical Report X3J16/95-0087, American National Standards Institute (ANSI), April 1995

[Barton94] John J. Barton, Lee R. Nackman: *Scientific and Engineering C++ – An Introduction with Advanced Techniques and Examples*, Addison-Wesley, Reading, MA, 1994, ISBN 0-201-53393-6

[Borland94] *Borland C++ 4.5 Programmer's Guide*, Borland International, Scotts Valley, CA

[Carroll95] Martin D. Carroll, Margaret A. Ellis: *Designing and Coding Reusable C++*, Addison-Wesley, Reading, MA, 1995, ISBN 0-201-51284-X

[Ellis90] Margaret A. Ellis, Bjarne Stroustrup: *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA, 1990, ISBN 0-201-51459-1

[Hanly95] Jeri R. Hanly, Elliot B. Koffman, Joan C. Horvath: *C Program Design for Engineers*, Addison-Wesley, Reading, MA, 1994, ISBN 0-201-59064-6

[Horstmann96] Cay S. Horstmann: *Mastering C++ – An Introduction to C++ and Object-Oriented Programming for C and Pascal Programmers (2nd Edition)*, John Wiley and Sons, New York, 1996, ISBN 0-471-10427-2

[Kernighan88] Brian Kernighan, Dennis Ritchie: *The C Programming Language (2nd Edition)*, Prentice-Hall, Englewood Cliffs, NJ, 1988, ISBN 0-13-937699-2

[Kukrika89] Milan Kukrika: *Programski jezik C*, Školska knjiga, Zagreb, 1989, ISBN 86-03-99627-X

[Liberty96] Jesse Liberty, J. Mark Hord: *Teach Yourself ANSI C++ in 21 Days*, Sams Publishing, Indianapolis, IN, 1996, ISBN 0-672-30887-6

[Lippman91] Stanley B. Lippman: *C++ Primer (2nd Edition)*, Addison-Wesley, Reading, MA, 1991, ISBN 0-201-53992-6

[Lippman96] Stanley B. Lippman: *Inside the C++ Object Model*, Addison-Wesley, Reading, MA, 1996, ISBN 0-201-83454-5

[Murray93] Robert B. Murray: *C++ Strategies and Tactics*, Addison-Wesley, Reading, MA, 1993, ISBN 0-201-56382-7

[Schildt90] Herbert Schildt: *Using Turbo C++*, Osborne/McGraw-Hill, Berkeley, CA, 1990, ISBN 0-07-881610-6

[Stroustrup91] Bjarne Stroustrup: *The C++ Programming Language (2nd Edition)*, Addison-Wesley, Reading, MA, 1991, ISBN 0-201-53992-6

[Stroustrup94] Bjarne Stroustrup: *The Design and Evolution of C++*, Addison-Wesley, Reading, MA, 1994, ISBN 0-201-54330-3

Ponegdje se pozivamo na neku od knjiga, navodeæi pripadajuæu, gore navedenu oznaku u uglatoj zagradi. Takoðer smo koristili èlanke iz èasopisa *C++ Report* u kojem se može naæi mnoštvo najnovijih informacija. Èasopis izlazi od 1989. godine, deset puta godišnje, a izdaje ga SIGS Publications Group, New York. Godišta 1991–1995 objavljena su na CDROM-u (SIGS Books, New York, ISBN 1-884842-24-0). Uz to, mnoštvo odgovora te praktiènih i korisnih rješenja može se naæi na Usenix konferencijama (*newsgroups*): *comp.lang.c++*, *comp.lang.c++.moderated i comp.std.c++*.

Od svih navedenih knjiga, "najreferentniji" je nacrt ANSI C++ standarda[†]. Može se naći na nekoliko Internet servera diljem svijeta i to u čistom tekstovnom formatu, PDF (Acrobat Reader) formatu ili u PostScript formatu. Broji oko 600 stranica formata A4 (oko 4MB komprimirane datoteke), a kao kuriozitet navedimo da je preko CARNet

[†] Iako se radi o *nacrtu* standarda, on u potpunosti definira sve karakteristike programskog jezika kakav æe on biti kada se (prema planu ANSI komiteta za standardizaciju jezika C++) u prosincu 1998. godine objavi konaèna verzija standarda. Sve promjene koje æe se dogaðati do tada odnosit æe se iskljuèivo na tekst standarda – sintaksa jezika i definicije biblioteka ostaju nepromijenjene.

mreže trebalo oko 6 sati vremena da se prebaci na naše računalo. Najnovija verzija nacrta standarda može se naručiti i poštom (po cijeni od 50 USD + poštarina) na sljedećoj adresi:

X3 Secretariat
CBEMA
1250 Eye Street NW
Suite 200
Washington, DC 20005

Naravno da ga ne preporuèujemo za uèenje jezika C++, a takoðer vjerujemo da neæe trebati niti iskusnijim korisnicima – Standard je prvenstveno namijenjen proizvoðaèima softvera. Svi noviji *prevoditelji* (engl. *compilers*) bi se trebali ponašati u skladu s tim standardom.

Zadnje poglavlje ove knjige posvećeno je principima objektno orijentiranog programiranja. Naravno da to nije dovoljno za ozbiljnije sagledavanje – detaljnije o objektno orijentiranom programiranju zainteresirani čitatelj može pročitati u referentnoj knjizi Grady Booch: *Object-Oriented Analysis and Design with Applications (2nd Edition)*, Benjamin/Cummings Publishing Co., Redwood City, CA, 1994, ISBN 0-8053-5340-2, kao i u nizu članaka istog autora u časopisu *C++ Report*.

2.6. Zahvale

Zahvaljujemo se svima koji su nam izravno ili posredno pomogli pri izradi ove knjige. Posebno se zahvaljujemo Branimiru Pejèinoviæu (*Portland State University*) koji nam je omoguæio da doðemo do Nacrta ANSI C++ standarda, Vladi Glaviniæu (*Fakultet elektrotehnike i raèunarstva*) koji nam je omoguæio da doðemo do knjiga [Stroustrup94] i [Carroll95] te nam je tijekom pripreme za tisak dao na raspolaganje laserski pisaè, te Zdenku Šimiæu (*Fakultet elektrotehnike i raèunarstva*) koji nam je, tijekom svog boravka u SAD, pomogao pri nabavci dijela literature.

Posebnu zahvalu upućujemo Ivi Mesarić koja je pročitala cijeli rukopis te svojim korisnim i konstruktivnim primjedbama znatno doprinijela kvaliteti iznesene materije. Također se zahvaljujemo Zoranu Kalafatiću (*Fakultet elektrotehnike i računarstva*) i Damiru Hodaku koji su čitali dijelove rukopisa i dali na njih korisne opaske.

Boris se posebno zahvaljuje gospođama Šribar na tonama kolača pojedenih za vrijeme dugih, zimskih noći čitanja i ispravljanja rukopisa, a koje su ga koštale kure mršavljenja.

I naravno, zahvaljujemo se Bjarne Stroustrupu i dečkima iz *AT&T*-a što su izmislili C++, naš najdraži programski jezik. Bez njih ne bi bilo niti ove knjige (ali možda bi bilo slične knjige iz FORTRAN-a).

3. “Oluja” kroz jezik C++

‘Što je to naredba?’
‘Da ugasim svjetiljku. Dobra večer.’ I on je ponovo upali.
‘Ne razumijem’ reče mali princ.
‘Nemaš što tu razumjeti’ reče noćobdija. ‘Naredba je naredba. Dobar dan.’ I on ugasi svoju svjetiljku.

Antoine de Saint-Exupéry (1900–1944), “Mali princ”

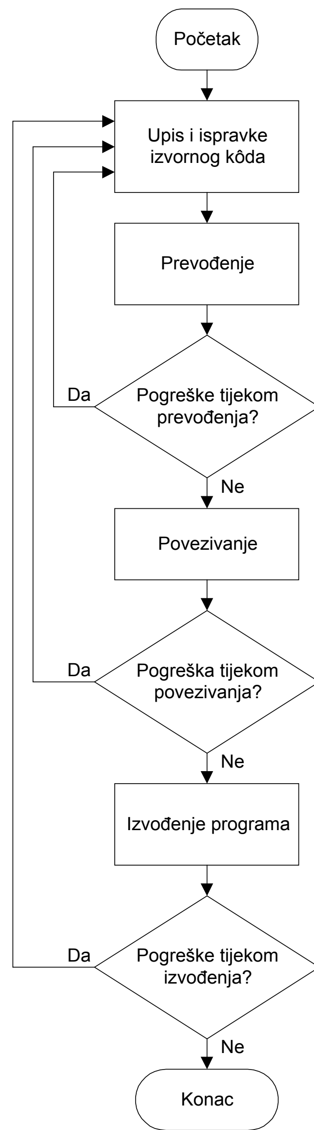
U uvodnom poglavlju proletjet æemo kroz osnovne pojmove vezane uz programiranje i upoznat æemo se s strukturom najjednostavnijih programa pisanih u programskom jeziku C++. Ovo poglavlje prvenstveno je namijenjeno èitateljicama i èitateljima koji nisu nikada pisali programe u bilo kojem višem programskom jeziku i onima koji su to možda radili, ali je “od tada prošla èitava vjeènost”.

3.1. Što je program i kako ga napisati

Elektronièka raèunala su danas postala pribor kojim se svakodnevno koristimo kako bismo si olakšali posao ili se zabavili. Istina, toènost prvog navoda æe mnogi poricati, istièuæi kao protuprimjer èinjenicu da im je za podizanje novca prije trebalo znatno manje vremena nego otkad su šalteri u banci kompjuterizirani. Ipak, èinjenica je da su mnogi poslovi danas nezamislivi bez raèunala; u krajnjoj liniji, dokaz za to je knjiga koju upravo èitate koja je u potpunosti napisana pomoæu raèunala.

Samo računalo, čak i kada se uključi u struju, nije kadro učiniti ništa korisno. Na današnjim računalima se ne može čak ni zagrijati prosječan ručak, što je inače bilo moguće na računalima s elektronskim cijevima. Ono što vam nedostaje jest pamet neophodna za koristan rad računala: programi, programi, … mnoštvo programa. Pod programom tu podrazumijevamo niz naredbi u strojnom jeziku koje procesor u vašem računalu izvodi i shodno njima obrađuje podatke, provodi matematičke proračune, ispisuje tekstove, iscrtava krivulje na zaslonu ili gađa vaš avion F-16 raketama srednjeg dometa. Pokretanjem programa s diska, diskete ili CD-ROM-a, program se učitava u radnu memoriju računala i procesor počinje s mukotrpnim postupkom njegova izvođenja.

Programi koje pokrećete na računalu su u *izvedbenom obliku* (engl. *executable*), razumljivom samo procesoru vašeg (i njemu sličnih) računala, sretnim procesorovim roditeljima negdje u Silicijskoj dolini i nekolicini zadrtih hakera širom svijeta koji još uvijek programiraju u strojnom kôdu. U suštini se strojni kôd sastoji od nizova binarnih znamenki: nula i jedinica. Budući da su današnji programi tipično duljine nekoliko megabajta, naslućujete da ih autori nisu pisali izravno u strojnom kôdu (kamo sreće da je tako – ne bi *Microsoft* tek tako svake dvije godine izbacivao nove *Windowse!*).



Slika 3.1. Tipičan razvoj programa

Gotovo svi današnji programi se pišu u nekom od *viših programskih jezika* (FORTRAN, BASIC, Pascal, C) koji su donekle razumljivi i ljudima (barem onima koji imaju nešto pojma o engleskom jeziku). Naredbe u tim jezicima se sastoje od mnemonika. Kombiniranjem tih naredbi programer slaže *izvorni kôd* (engl. *source code*) programa, koji se pomoću posebnih programa *prevoditelja* (engl. *compiler*) i *povezivača* (engl. *linker)* prevodi u izvedbeni kôd. Prema tome, pisanje programa u užem smislu podrazumijeva pisanje izvornog kôda. Međutim, kako pisanje kôda nije samo sebi svrhom, pod pisanjem programa u širem smislu podrazumijeva se i prevođenje, odnosno povezivanje programa u izvedbeni kôd. Stoga možemo govoriti o četiri faze izrade programa:

1. pisanje izvornog kôda
2. prevođenje izvornog kôda,
3. povezivanje u izvedbeni kôd te
4. testiranje programa.

Da bi se za neki program moglo reæi da je uspješno zgotovljen, treba uspješno proæi kroz sve èetiri faze.

Kao i svaki drugi posao, i pisanje programa iziskuje određeno znanje i vještinu. Prilikom pisanja programera vrebaju Scile i Haribde, danas poznatije pod nazivom pogreške ili *bugovi* (engl. *bug* - stjenica) u programu. Uoči li se pogreška u nekoj od faza izrade programa, izvorni kôd treba doraditi i ponoviti sve prethodne faze. Zbog toga postupak izrade programa nije pravocrtan, već manje-više podsjeća na mukotrpno petljanje u krug. Na slici 3.1 je shematski prikazan cjelokupni postupak izrade programa, od njegova začetka, do njegova okončanja. Analizirajmo najvažnije faze izrade programa.

Prva faza programa je pisanje izvornog kôda. U principu se izvorni kôd može pisati u bilo kojem programu za uređivanje teksta (engl. *text editor*), međutim velika većina današnjih prevoditelja i povezivača se isporučuje kao cjelina zajedno s ugrađenim programom za upis i ispravljanje izvornog kôda. Te programske cjeline poznatije su pod

nazivom *integrirane razvojne okoline* (engl. *integrated development environment, IDE*). Nakon što je (prema obično nekritičkom mišljenju programera) pisanje izvornog kôda završeno, on se pohrani u datoteku izvornog kôda na disku. Toj datoteci se obično daje nekakvo smisleno ime, pri čemu se ono za kôdove pisane u programskom jeziku C++ obično proširuje nastavkom `.cpp`, `.cp` ili samo `.c`, na primjer `pero.cpp`. Nastavak je potreban samo zato da bismo izvorni kôd kasnije mogli lakše pronaći.

Slijedi prevođenje izvornog kôda. U integriranim razvojnim okolinama program za prevođenje se pokreće pritiskom na neku tipku na zaslonu, pritiskom odgovarajuće tipke na tipkovnici ili iz nekog od *izbornika* (engl. *menu*) – ako prevoditelj nije integriran, poziv je nešto složeniji[†]. Prevoditelj tijekom prevođenja provjerava sintaksu napisanog izvornog kôda i u slučaju uočenih ili naslućenih pogrešaka ispisuje odgovarajuće poruke o pogreškama, odnosno upozorenja. Pogreške koje prijavi prevoditelj nazivaju se *pogreškama pri prevođenju* (engl. *compile-time errors*). Nakon toga programer će pokušati ispraviti sve navedene pogreške i ponovo prevesti izvorni kôd – sve dok prevođenje kôda ne bude uspješno okončano, neće se moći pristupiti povezivanju kôda. Prevođenjem izvornog dobiva se datoteka *objektnog kôda* (engl. *object code*), koja se lako može prepoznata po tome što obično ima nastavak `.o` ili `.obj` (u našem primjeru bi to bio `pero.obj`).

Nakon što su ispravljene sve pogreške uočene prilikom prevođenja i kôd ispravno preveden, pristupa se povezivanju objektnih kôdova u izvedbeni. U većini slučajeva objektni kôd dobiven prevođenjem programerovog izvornog kôda treba povezati s postojećim *bibliotekama* (engl. *libraries*). Biblioteke su datoteke u kojima se nalaze već prevedene gotove funkcije ili podaci. One se isporučuju zajedno s prevoditeljem, mogu se zasebno kupiti ili ih programer može tijekom rada sam razvijati. Bibliotekama se izbjegava opetovano pisanje vrlo često korištenih operacija. Tipičan primjer za to je biblioteka matematičkih funkcija koja se redovito isporučuje uz prevoditelje, a u kojoj su definirane sve funkcije poput trigonometrijskih, hiperbolnih, eksponencijalnih i sl. Prilikom povezivanja provjerava se mogu li se svi pozivi kôdova realizirati u izvedbenom kôdu. Uoči li povezivač neku nepravilnost tijekom povezivanja, ispisat će poruku o pogreški i onemogućiti generiranje izvedbenog kôda. Ove pogreške nazivaju se *pogreškama pri povezivanju* (engl. *link-time errors*) – sada programer mora prionuti ispravljanju pogrešaka koje su nastale pri povezivanju. Nakon što se isprave sve pogreške, kôd treba ponovno prevesti i povezati.

Uspješnim povezivanjem dobiva se izvedbeni kôd. Međutim, takav izvedbeni kôd još uvijek ne jamči da će program raditi ono što ste zamislili. Primjerice, može se dogoditi da program radi pravilno za neke podatke, ali da se za druge podatke ponaša nepredvidivo. U tom se slučaju radi o *pogreškama pri izvođenju* (engl. *run-time errors*). Da bi program bio potpuno korektan, programer treba istestirati program da bi uočio i ispravio te pogreške, što znači ponavljanje cijelog postupka u lancu 'ispravljanje izvornog kôda-prevođenje-povezivanje-testiranje'. Kod jednostavnijih programa broj ponavljanja će biti manji i smanjivat će se proporcionalno s rastućim iskustvom programera. Međutim, kako raste složenost programa, tako se povećava broj mogućih

[†] Ovdje neæemo opisivati konkretno kako se pokreæu postupci prevoðenja ili povezivanja, jer to varira ovisno o prevoditelju, odnosno povezivaèu.

pogrešaka i cijeli postupak izrade programa neiskusnom programeru može postati mukotrpan.

Za ispravljanje pogrešaka pri izvođenju, programeru na raspolaganju stoje programi za otkrivanje pogrešaka (engl. *debugger*). Radi se o programima koji omogućavaju prekid izvođenja izvedbenog kôda programa koji testiramo na unaprijed zadanim naredbama, izvođenje programa naredbu po naredbu, ispis i promjene trenutnih vrijednosti pojedinih podataka u programu. Najjednostavniji programi za otkrivanje pogrešaka ispisuju izvedbeni kôd u obliku strojnih naredbi. Međutim, većina današnjih naprednih programa za otkrivanje pogrešaka su *simbolički* (engl. *symbolic debugger*) – iako se izvodi prevedeni, strojni kôd, izvođenje programa se prati preko izvornog kôda pisanog u višem programskom jeziku. To omogućava vrlo lagano lociranje i ispravljanje pogrešaka u programu.

Osim pogrešaka, prevoditelj i povezivač redovito dojavljuju i upozorenja. Ona ne onemogućavaju nastavak prevođenja, odnosno povezivanja kôda, ali predstavljaju potencijalnu opasnost. Upozorenja se mogu podijeliti u dvije grupe. Prvu grupu čine upozorenja koja javljaju da kôd nije potpuno korektan. Prevoditelj ili povezivač će zanemariti našu pogrešku i prema svom nahođenju izgenerirati kôd. Drugu grupu čine poruke koje upozoravaju da "nisu sigurni je li ono što smo napisali upravo ono što smo željeli napisati", tj. radi se o dobronamjernim upozorenjima na zamke koje mogu proizići iz načina na koji smo program napisali. Iako će, unatoč upozorenjima, program biti preveden i povezan (možda čak i korektno), pedantan programer neće ta upozorenja nikada zanemariti – ona često upućuju na uzrok pogrešaka pri izvođenju gotovog programa. Za precizno tumačenje poruka o pogreškama i upozorenja neophodna je dokumentacija koja se isporučuje uz prevoditelj i povezivač.

Da zaključimo: "Što nam dakle treba za pisanje programa u jeziku C++?" Osim računala, programa za pisanje i ispravljanje teksta, prevoditelja i povezivača trebaju vam još samo tri stvari: interes, puno slobodnih popodneva i dooobra knjiga. Interes vjerojatno postoji, ako ste s čitanjem stigli čak do ovdje. Slobodno vrijeme će vam trebati da isprobate primjere i da se sami okušate u bespućima C++ zbiljnosti. Jer, kao što stari južnohrvatski izrijek kaže: "Nima dopisne škole iz plivanja". Stoga ne gubite vrijeme i odmah otkažite sudar curi ili dečku. A za dooobru knjigu…("*ta-ta-daaam*" – tu sada mi upadamo!).

3.2. Moj prvi i drugi C++ program

Bez mnogo okolišanja i filozofiranja, napišimo najjednostavniji program u jeziku C++:

```
int main() {
    return 0;
}
```

Utipkate li nadobudno ovaj program u svoje raèunalo, pokrenete odgovarajuæi prevoditelj, te nakon uspješnog prevođenja pokrenete program, na zaslonu raèunala sigurno neæete dobiti ništa! Nemojte se odmah hvatati za glavu, laæati telefona i zvati svoga dobavljaèa raèunala. Gornji program zaista ništa ne radi, a ako nešto i dobijete na zaslonu vašeg raèunala, znaèi da ste negdje pogriješili prilikom utipkavanja. Unatoè jalovosti gornjeg kôda, promotrimo ga, redak po redak.

U prvom retku je napisano `int main()`. `main` je naziv za glavnu funkciju[†] u svakom C++ programu. Izvođenje svakog programa počinje naredbama koje se nalaze u njoj.

> Svaki program napisan u C++ jeziku mora imati točno (ni manje ni više) jednu `main()` funkciju.

Pritom valja uoèiti da je to samo simbolièko ime koje daje do znanja prevoditelju koji se dio programa treba prvo poèeti izvoditi – ono nema nikakve veze s imenom izvedbenog programa koji se nakon uspješnog prevođenja i povezivanja dobiva. Željeno ime izvedbenog programa određuje sam programer: ako se korišteni prevoditelj pokreæe iz komandne linije, ime se navodi kao parametar u komandnoj liniji, a ako je prevoditelj ugrađen u neku razvojnu okolinu, tada se ime navodi kao jedna od opcija. Toène detalje definiranja imena izvedbenog imena èitateljica ili èitatelj naæi æe u uputama za prevoditelj kojeg koristi.

Riječ `int` ispred oznake glavne funkcije ukazuje na to da će `main()` po završetku izvođenja naredbi i funkcija sadržanih u njoj kao rezultat tog izvođenja vratiti cijeli broj (`int` dolazi od engleske riječi *integer* koja znači *cijeli broj*). Budući da se glavni program pokreće iz operacijskog sustava (DOS, UNIX, MS Windows), rezultat glavnog programa se vraća operacijskom sustavu. Najčešće je to kôd koji signalizira pogrešku nastalu tijekom izvođenja programa ili obavijest o uspješnom izvođenju.

Iza riječi `main` slijedi par otvorena-zatvorena zagrada. Unutar te zagrade trebali bi doći opisi podataka koji se iz operacijskog sustava prenose u `main()`. Ti podaci nazivaju se *argumenti* funkcije. Za funkciju `main()` to su parametri koji se pri pokretanju programa navode u komandnoj liniji iza imena programa, kao na primjer:

---

† U nekim programskim jezicima glavna funkcija se zove *glavni program*, a sve ostale funkcije *potprogrami*.

```
pkunzip -t mojzip
```

Ovom naredbom se pokreæe program `pkunzip` i pritom mu se predaju dva parametra: `-t` i `mojzip`. U našem C++ primjeru unutar zagrada nema ništa, što znaèi da ne prenosimo nikakve argumente. Tako æe i ostati do daljnjega, toènije do poglavlja 5.11 u kojem æemo detaljnije obraditi funkcije, a posebice funkciju `main()`.

Slijedi otvorena vitičasta zagrada. Ona označava početak *bloka* u kojem će se nalaziti naredbe glavne funkcije, dok zatvorena vitičasta zagrada u zadnjem retku označava kraj tog bloka. U samom bloku prosječni čitatelj uočit će samo jednu naredbu, `return 0`. Tom naredbom glavni program vraća pozivnom programu broj 0, a to je poruka operacijskom sustavu da je program uspješno okončan, štogod on radio.

Uočimo znak `;` (točka-zarez) iza naredbe `return 0`! On označava kraj naredbe te služi kao poruka prevoditelju da sve znakove koji slijede interpretira kao novu naredbu.

Znak `;` mora zakljuèivati svaku naredbu u jeziku C++.

Radi kratkoæe æemo u veæini primjera u knjizi izostavljati uvodni i zakljuèni dio glavne funkcije te æemo podrazumijevati da oni u konkretnom kôdu postoje.

Pogledajmo još na trenutak što bi se dogodilo da smo kojim slučajem napravili neku pogrešku prilikom upisivanja gornjeg kôda. Recimo da smo zaboravili desnu vitičastu zagradu na kraju kôda:

```
int main() {
    return 0;
```

Prilikom prevoðenja prevoditelj æe uoèiti da funkcija `main()` nije pravilno zatvorena, te æe ispisati poruku o pogreški oblika "Pogreška u prvi.cpp *xx*: složenoj naredbi nedostaje } u funkciji main()". U ovoj poruci je `prvi.cpp` ime datoteke u koju smo naš izvorni kôd pohranili, a *xx* je broj retka u kojem se pronaðena pogreška nalazi. Zaboravimo li napisati naredbu `return`:

```
int main() {
}
```

neki prevoditelji æe javiti upozorenje oblika "Funkcija bi trebala vratiti vrijednost". Iako se radi o pogreški, prevoditelj æe umetnuti odgovarajuæi kôd prema svom nahoðenju i prevesti program. Ne mali broj korisnika æe zbog toga zanemariti takva upozorenja. Meðutim, valja primijetiti da èesto taj umetnuti kôd ne odgovara onome što je programer zamislio. Zanemarivanjem upozorenja programer gubi pregled nad korektnošæu kôda, pa se lako može dogoditi da rezultirajuæi izvedbeni kôd daje na prvi pogled neobjašnjive rezultate.

Programi poput gornjeg nemaju baš neku praktičnu primjenu, te daljnju analizu gornjeg kôda prepuštamo poklonicima minimalizma. *We need some action, man!* Stoga pogledajmo sljedeći primjer:

```
#include <iostream.h>

int main() {
    cout << "Ja sam za C++!!! A vi?" << endl;
    return 0;
}
```

U ovom primjeru uoèavamo dva nova retka. U prvom retku nalazi se naredba `#include <iostream.h>` kojom se od prevoditelja zahtijeva da u naš program ukljuèi biblioteku `iostream`. U toj biblioteci nalazi se *izlazni tok* (engl. *output stream*) te funkcije koje omoguæavaju ispis podataka na zaslonu. Ta biblioteka nam je neophodna da bismo u prvom retku glavnoga programa ispisali tekst poruke. Naglasimo da `#include` nije naredba C++ jezika, nego se radi o *pretprocesorskoj naredbi*. Naletjevši na nju, prevoditelj æe prekinuti postupak prevođenja kôda u tekuæoj datoteci, skoèiti u datoteku `iostream.h`, prevesti ju, a potom se vratiti u poèetnu datoteku na redak iza naredbe `#include`. Sve pretprocesorske naredbe poèinju znakom `#`.

`iostream.h` je primjer *datoteke zaglavlja* (engl. *header file*, odakle i slijedi nastavak `.h`). U takvim datotekama se nalaze deklaracije funkcija sadržanih u odgovarajućim bibliotekama. Jedna od osnovnih značajki (zli jezici će reći mana) jezika C++ jest vrlo oskudan broj funkcija ugrađenih u sam jezik. Ta oskudnost olakšava učenje samog jezika, te bitno pojednostavnjuje i ubrzava postupak prevođenja. Za specifične zahtjeve na raspolaganju je veliki broj odgovarajućih biblioteka funkcija i klasa.

`cout` je ime izlaznog toka definiranog u biblioteci `iostream`, pridruženog zaslonu računala. Operatorom `<<` (dva znaka “manje od”) podatak koji slijedi upućuje se na izlazni tok, tj. na zaslon računala. U gornjem primjeru to je kratka promidžbena poruka:

```
Ja sam za C++!!! A vi?
```

Ona je u programu napisana unutar znakova navodnika, koji upuæuju na to da se radi o tekstu koji treba ispisati doslovce. Biblioteka `iostream` bit æe detaljnije opisana kasnije, u poglavlju 16.

Međutim, to još nije sve! Iza znakovnog niza ponavlja se operator za ispis, kojeg slijedi `endl`. `endl` je konstanta u biblioteci `iostream` koja prebacuje ispis u novi redak, to jest vraća *kurzor* (engl. *cursor*) na početak sljedećeg retka na zaslonu. Dovitljiva čitateljica ili čitatelj će sami zaključiti da bi se operatori za ispis mogli dalje nadovezivati u istoj naredbi:

```
cout << "Ja sam za C++!!! A vi?" << endl << "Ne, hvala!";
```

Zadatak. *Kako bi izgledao izvorni kôd ako bismo željeli ispis* `endl` *znaka staviti u posebnu naredbu? Dodajte u gornji primjer ispis poruke* `Imamo C++!!!` *u sljedećem retku (popratno sklapanje ruku iznad glave nije neophodno). Nemojte zaboraviti dodati i ispis* `endl` *na kraju tog retka!*

3.3. Moj treæi C++ program

Sljedeæi primjer je pravi mali dragulj interaktivnog programa:

```
#include <iostream.h>

int main() {
    int a, b, c;

    cout << "Upiši prvi broj:";
    cin >> a;                    // očekuje prvi broj

    cout << "Upiši i drugi broj:";
    cin >> b;                    // očekuje drugi broj

    c = a + b;                   // računa njihov zbroj
    // ispisuje rezultat:
    cout << "Njihov zbroj je: " << c << endl;
    return 0;
}
```

U ovom primjeru uoèavamo nekoliko novina. Prvo, to je redak

```
int a, b, c;
```

u kojem se deklariraju tri varijable `a`, `b` i `c`. Kljuènom rijeèi (*identifikatorom tipa*) `int` deklarirali smo ih kao cjelobrojne, tj. tipa `int`. Deklaracijom smo pridijelili simbolièko, nama razumljivo i lako pamtivo ime memorijskom prostoru u koji æe se pohranjivati vrijednosti tih varijabli. Naišavši na te deklaracije, prevoditelj æe zapamtiti njihova imena, te za njih rezervirati odgovarajuæi prostor u memoriji raèunala. Kada tijekom prevoðenja ponovno sretne varijablu s nekim od tih imena, prevoditelj æe znati da se radi o cjelobrojnoj varijabli i znat æe gdje bi se ona u memoriji trebala nalaziti. Osim toga, tip varijable odreðuje raspon dozvoljenih vrijednosti te varijable, te definira operacije nad njima. Opširnije o deklaracijama varijabli i o tipovima podataka govorit æemo u sljedeæem poglavlju.

Slijedi ispis poruke `Upiši prvi broj:`, čije značenje ne trebamo objašnjavati. Iza poruke nismo dodali ispis znaka `endl`, jer želimo da nam kurzor ostane iza poruke, u istom retku, spreman za unos prvog broja.

Druga novina je *ulazni tok* (engl. *input stream*) `cin` koje je zajedno s izlaznim tokom definiran u datoteci zaglavlja `iostream.h`. On služi za učitavanje podataka s konzole, tj. tipkovnice. Operatorom `>>` (dvostruki znak “veće od”) podaci s konzole

upućuju se u memoriju varijabli `a` (prvi broj), odnosno `b` (drugi broj). Naglasimo da učitavanje počinje tek kada se na tipkovnici pritisne tipka za novi redak, tj. tipka *Enter*. To znači da kada pokrenete program, nakon ispisane poruke možete utipkavati bilo što i to po potrebi brisati – tek kada pritisnete tipku *Enter*, program će analizirati unos te pohraniti broj koji je upisan u memoriju računala. Napomenimo da taj broj u našem primjeru ne smije biti veći od 32767 niti manji od –32768, jer je to dozvoljeni raspon cjelobrojnih `int` varijabli. Unos većeg broja neće imati nikakve dramatične posljedice na daljnji rad vašeg računala, ali će dati krivi rezultat na kraju računa.

> Svaki puta kada nam u programu treba ispis podataka na zaslonu ili unos podataka preko tipkovnice pomoæu ulazno-izlaznih tokova `cin` ili `cout`, treba ukljuèiti zaglavlje odgovarajuæe `iostream` biblioteke pretprocesorskom naredbom `#include`.

Radi sažetosti kôda, u veæini primjera koji slijede pretprocesorska naredba za ukljuèivanje `iostream` biblioteke neæe biti napisana, ali se ona podrazumijeva. Koriste li se ulazno-izlazni tokovi, njeno izostavljanje prouzroèit æe pogrešku kod prevođenja.

No, završimo s našim primjerom. Nakon unosa prvog broja, po istom principu se unosi drugi broj `b`, a zatim slijedi naredba za računanje zbroja

```
c = a + b;
```

Ona kaže raèunalu da treba zbrojiti vrijednosti varijabli `a` i `b`, te njihov zbroj pohraniti u varijablu `c`, tj. u memoriju na mjesto gdje je prevoditelj rezervirao prostor za tu varijablu.

U početku će čitatelj zasigurno imati problema s razlučivanjem operatora `<<` i `>>`. U vjeri da će olakšati razlikovanje operatora za ispis i učitavanje podataka, dajemo sljedeći naputak.

> Operatore `<<` i `>>` možemo shvatiti kao da pokazuju smjer prijenosa podataka [Lippman91].

Primjerice,

```
>> a
```

preslikava vrijednost u `a`, dok

```
<< c
```

preslikava vrijednost iz `c`.

3.4. Komentari

Na nekoliko mjesta u gornjem kôdu možemo uoèiti tekstove koji zapoèinju dvostrukim kosim crtama (`//`). Radi se o *komentarima*. Kada prevoditelj naleti na dvostruku kosu crtu, on æe zanemariti sav tekst koji slijedi do kraja tekuæeg retka i prevoðenje æe nastaviti u sljedeæem retku. Komentari dakle ne ulaze u izvedbeni kôd programa i služe programerima za opis znaèenja pojedinih naredbi ili dijelova kôda. Komentari se mogu pisati u zasebnom retku ili recima, što se obièno rabi za dulje opise, primjerice što odreðeni program ili funkcija radi:

```
//
// Ovo je moj treći C++ program, koji zbraja
// dva broja unesena preko tipkovnice, a zbroj
// ispisuje na zaslonu.
//                 Autor: N. N. Hacker III
//
```

Za kraæe komentare, na primjer za opise varijabli ili nekih operacija, komentar se piše u nastavku naredbe, kao što smo vidjeli u primjeru.

Uz gore navedeni oblik komentara, jezik C++ podržava i komentare unutar para znakova `/* */`. Takvi komentari započinju slijedom `/*` (kosa crta i zvjezdica), a završavaju slijedom `*/` (zvjezdica i kosa crta). Kraj retka ne znači podrazumijevani završetak komentara, pa se ovakvi komentari mogu protezati na nekoliko redaka izvornog kôda, a da se pritom znak za komentiranje ne mora ponavljati u svakom retku:

```
/*
    Ovakav način komentara
    preuzet je iz programskog
    jezika C.
*/
```

Stoga je ovakav naèin komentiranja naroèito pogodan za (privremeno) iskljuèivanje dijelova izvornog kôda. Ispred naredbe u nizu koji želimo iskljuèiti dodat æemo oznaku `/*` za poèetak komentara, a iza zadnje naredbe u nizu nadodat æemo oznaku `*/` za zakljuèenje komentara.

Iako komentiranje programa iziskuje dodatno vrijeme i napor, u kompleksnijim programima ono se redovito isplati. Dogodi li se da netko drugi mora ispravljati vaš kôd, ili (još gore) da nakon dugo vremena vi sami morate ispravljati svoj kôd, komentari će vam olakšati da proniknete u ono što je autor njime htio reći. Svaki ozbiljniji programer ili onaj tko to želi postati mora biti svjestan da će nakon desetak ili stotinjak napisanih programa početi zaboravljati čemu pojedini program služi. Zato je vrlo korisno na početku datoteke izvornog programa u komentaru navesti osnovne "generalije", na primjer ime programa, ime datoteke izvornog kôda, kratki opis onoga što bi program trebao raditi, funkcije i klase definirane u datoteci te njihovo značenje, autor(i) kôda, uvjeti pod kojima je program preveden (operacijski sustav, ime i oznaka prevoditelja), zabilješke, te naknadne izmjene:

```
/*********************************************************

Program:    Moj treći C++ program
Datoteka:   Treci.cpp
Funkcije:   main() - cijeli program je u jednoj datoteci
Opis:       Učitava dva broja i ispisuje njihov zbroj
Autori:     Boris (bm)
            Julijan (jš)
Okruženje:  Prozori95
            PHP C++ 4.5 prevoditelj
Zabilješke: Znam Boris da si ti protiv ovakvih komentara,
            ali sam ih morao staviti
Izmjene:    15.07.96.   (jš) prva inačica
            21.08.96.   (bm) svi podaci su iz float
                             promijenjeni u int

*********************************************************/
```

S druge strane, s kolièinom komentara ne treba pretjerivati, jer æe u protivnom izvorni kôd postati nepregledan. Naravno da æete izbjegavati komentare poput:

```
c = a + b;          // zbraja a i b
i++;                // uvećava i za 1
y = sqrt(x)         // poziva funkciju sqrt
```

Nudimo vam sljedeæe naputke gdje i što komentirati:

- Na poèetku datoteke izvornog kôda opisati sadržaj datoteke.
- Kod deklaracije varijabli, klasa i objekata obrazložiti njihovo znaèenje i primjenu.
- Ispred funkcije dati opis što radi, što su joj argumenti i što vraæa kao rezultat. Eventualno dati opis algoritma koji se primjenjuje.
- Dati sažeti opis na mjestima u programu gdje nije potpuno oèito što kôd radi.

Radi bolje razumljivosti, primjeri u knjizi bit æe redovito prekomentirani, tako da æe komentari biti pisani i tamo gdje je iskusnom korisniku jasno znaèenje kôda. Osim toga, redovito æemo ubacivati komentare uz naredbe za koje bi prevoditelj javio pogrešku.

3.5. Rastavljanje naredbi

Razmotrimo još dva problema važna svakoj poèetnici ili poèetniku: praznine u izvornom kôdu i produljenje naredbi u više redaka. U primjerima u knjizi intenzivno æemo koristiti praznine između imena varijabli i operatora, iako ih iskusniji programeri èesto izostavljaju. Tako smo umjesto

```
c = a + b;
```

mogli pisati

```
c=a+b;
```

Umetanje praznina doprinosi preglednosti kôda, a èesto je i neophodno da bi prevoditelj interpretirao djelovanje nekog operatora onako kako mi oèekujemo od njega. Ako je negdje dozvoljeno umetnuti prazninu, tada broj praznina koje se smiju umetnuti nije ogranièen, pa smo tako gornju naredbu mogli pisati i kao

```
c=          a    +            b       ;
```

što je još nepreglednije! Istaknimo da se pod prazninom ne podrazumijevaju iskljuèivo prazna mjesta dobivena pritiskom na razmaknicu (engl. *blanks*), veæ i praznine dobivene tabulatorom (engl. *tabs*) te znakove za pomak u novi red (engl. *newlines*).

Naredbe u izvornom kôdu nisu ograničene na samo jedan redak, već se mogu protezati na nekoliko redaka – završetak svake naredbe jednoznačno je određen znakom `;`. Stoga pisanje naredbe možemo prekinuti na bilo kojem mjestu gdje je dozvoljena praznina, te nastaviti pisanje u sljedećem retku, na primjer

```
c =
a + b;
```

Naravno da je ovakvim potezom kôd iz razmatranog primjera postao nepregledniji. Razdvajati naredbe u više redaka ima smisla samo kod vrlo dugaèkih naredbi, kada one ne stanu u jedan redak. Veæina klasiènih štampaèa i ekranskih editora podržava retke od 80 znakova, pa vam preporuèujemo da sve naredbe koje zauzimaju više od 80 znakova razbijete u više redaka. Zbog formata knjige duljine redaka u našim primjerima su manje.

Posebnu pažnju treba obratiti na razdvajanje znakovnih nizova. Ako bismo pokušali prevesti sljedeći primjer, dobit ćemo pogrešku prilikom prevođenja:

```
#include <iostream.h>

int main() {
    // pogreška: nepravilno razdvojeni znakovni niz
    cout << "Pojavom jezika C++ ostvaren je
             tisućljetni san svih programera" << endl;
    return 0;
}
```

Prevoditelj æe javiti pogrešku da znakovni niz `Pojavom ... ostvaren je` nije zakljuèen znakom dvostrukog navodnika, jer prevoditelj ne uoèava da se niz nastavlja u sljedeæem retku. Da bismo mu to dali do znanja, završetak prvog dijela niza treba oznaèiti lijevom kosom crtom \ (engl. *backslash*):

```
    cout << "Pojavom jezika C++ ostvaren je \
tisućljetni san svih programera" << endl;
```

pri èemu nastavak niza ne smijemo uvuæi, jer bi prevoditelj dotiène praznine prihvatio kao dio niza. Stoga je u takvim sluèajevima preglednije niz rastaviti u dva odvojena niza:

```
cout << "Pojavom jezika C++ ostvaren je "
        "tisućljetni san svih programera" << endl;
```

Pritom se ne smije zaboraviti staviti prazninu na kraju prvog ili poèetak drugog niza. Gornji niz mogli smo rastaviti na bilo kojem mjestu:

```
cout << "Pojavom jezika C++ ostvaren je tis"
        "ućljetni san svih programera" << endl;
```

ali je oèito takav pristup manje èitljiv.

Budući da znak `;` označava kraj naredbe, moguće je više naredbi napisati u jednom retku. Tako smo “Moj treći program” mogli napisati i na sljedeći način:

```
#include <iostream.h>

int main() {
    int a, b, c; cout << "Upiši prvi broj:"; cin >> a;
    cout << "Upiši i drugi broj:"; cin >> b; c = a + b;
    cout << "Njihov zbroj je: " << c << endl;
    return 0;
}
```

Program æe biti preveden bez pogreške i raditi æe ispravno. No, oèito je da je ovako pisani izvorni kôd daleko neèitljiviji – pisanje više naredbi u istom retku treba prakticirati samo u izuzetnim sluèajevima.

4. Osnovni tipovi podataka

Postoje dva tipa ljudi:
jedni koji nose pištolje, drugi koji kopaju...

Clint Eastwood, u filmu "Dobar, loš, zao"

Svaki program sadrži u sebi podatke koje obraðuje. Njih možemo podijeliti na nepromjenjive *konstante*, odnosno promjenjive *varijable* (*promjenjivice*). Najjednostavniji primjer konstanti su brojevi (5, 10, 3.14159). Varijable su podaci koji opæenito mogu mijenjati svoj iznos. Stoga se oni u izvornom kôdu predstavljaju ne svojim iznosom veæ simbolièkom oznakom, *imenom varijable*.

Svaki podatak ima dodijeljenu oznaku tipa, koja govori o tome kako se dotični podatak pohranjuje u memoriju računala, koji su njegovi dozvoljeni rasponi vrijednosti, kakve su operacije moguće s tim podatkom i sl. Tako razlikujemo cjelobrojne, realne, logičke, pokazivačke podatke. U poglavlju koje slijedi upoznat ćemo se ugrađenim tipovima podataka i pripadajućim operatorima.

4.1. Identifikatori

Mnogim dijelovima C++ programa (varijablama, funkcijama, klasama) potrebno je dati odreðeno ime. U tu svrhu koriste se identifikatori, koje možemo proizvoljno nazvati. Pri tome je bitno poštivati tri osnovna pravila:

1. Identifikator može biti sastavljen od kombinacije slova engleskog alfabeta (`A` - `Z`, `a` - `z`), brojeva (`0` - `9`) i znaka za podcrtavanje '_' (engl. *underscore*).
2. Prvi znak mora biti slovo ili znak za podcrtavanje.
3. Identifikator ne smije biti jednak nekoj od ključnih riječi (vidi tablicu 4.1) ili nekoj od alternativnih oznaka operatora (tablica 4.2). To ne znači da ključna riječ ne može biti dio identifikatora – `moj_int` je dozvoljeni identifikator. Također, treba izbjegavati da naziv identifikatora sadrži dvostruke znakove podcrtavanja (__) ili da započinje znakom podcrtavanja i velikim slovom, jer su takve oznake rezervirane za C++ implementacije i standardne biblioteke (npr. `__LINE__`, `__FILE__`).

Stoga si možemo pustiti mašti na volju pa svoje varijable i funkcije nazivati svakojako. Pritom je vjerojatno svakom jasno da je zbog razumljivosti kôda poželjno imena odabirati tako da odražavaju stvarno znaèenje varijabli, na primjer:

```
Pribrojnik1
Pribrojnik2
rezultat
```

Tablica 4.1. Kljuène rijeèi jezika C++

asm	else	operator	throw
auto	enum	private	true
bool	explicit	protected	try
break	extern	public	typedef
case	false	register	typeid
catch	float	reinterpret_cast	typename
char	for	return	union
class	friend	short	unsigned
const	goto	signed	using
const_cast	if	sizeof	virtual
continue	inline	static	void
default	int	static_cast	volatile
delete	long	struct	wchar_t
do	mutable	switch	while
double	namespace	template	
dynamic_cast	new	this	

Tablica 4.2. Alternativne oznake operatora

and	bitand	compl	not_eq	or_eq	xor_eq
and_eq	bitor	not	or	xor	

a izbjegavati imena poput

```
Snjeguljica_i_7_patuljaka
MojaPrivatnaVarijabla
FrankieGoesToHollywood
RockyXXVII
```

Unatoè èinjenici da æe neki prevoditelji prihvatiti i naše dijakritièke znakove u identifikatorima, zbog prenosivosti kôda preporuèuje se njihovo izbjegavanje. U protivnom se lako može dogoditi da vam program s varijablama `BežiJankec` ili `TeksaškiMasakrMotornjaèom` proðe kod jednog prevoditelja, a na drugom prouzroèi pogrešku prilikom prevođenja.

Valja uočiti da jezik C++ razlikuje velika i mala slova u imenima, tako da identifikatori

```
maliNarodiVelikeIdeje
MaliNarodiVelikeIdeje
malinarodiVELIKEIDEJE
```

predstavljaju tri razlièita naziva. Takoðer, iako ne postoji teoretsko ogranièenje na duljinu imena, svi prevoditelji razlikuju imena samo po prvih nekoliko znakova (pojam

“nekoliko” je ovdje prilièno rastezljiv), pa ako prevoditelj kojeg koristite uzima u obzir samo prvih 14 znakova, tada æe identifikatore

```
Snjeguljica_i_7_patuljaka
Snjeguljica_i_sedam_patuljaka
```

interpretirati kao iste, iako se razlikuju od petnaestog znaka (`'7'` odnosno `'s'`) na dalje. Naravno da dugaèka imena treba izbjegavati radi vlastite komocije, posebice za varijable i funkcije koje se èesto ponavljaju.

Ponekad je pogodno koristiti složena imena zbog boljeg razumijevanja kôda. Iz gornjih primjera čitateljica ili čitatelj mogli su razlučiti dva najčešća pristupa označavanju složenih imena. U prvom pristupu riječi od kojih je ime sastavljeno odvajaju se znakom za podcrtavanje, poput

```
Snjeguljica_te_sedam_patuljaka
```

(praznina umjesto znaka podcrtavanja između rijeèi oznaèavala bi prekid imena, što bi uzrokovalo pogrešku prilikom prevođenja). U drugom pristupu rijeèi se pišu spojeno, s velikim poèetnim slovom:

```
SnjeguljicaTeSedamPatuljaka
```

Mi æemo u primjerima preferirati potonji naèin samo zato jer tako oznaèene varijable i funkcije zauzimaju ipak nešto manje mjesta u izvornom kôdu.

Iako ne postoji nikakvo ograničenje na početno slovo naziva varijabli, većina programera preuzela je iz programskog jezika FORTRAN naviku da imena cjelobrojnih varijabli započinje slovima `i`, `j`, `k`, `l`, `m` ili `n`.

4.2. Varijable, objekti i tipovi

Bez obzira na jezik u kojem je pisan, svaki program sastoji se od niza naredbi koje mijenjaju vrijednosti objekata pohranjenih u memoriji raèunala. Raèunalo dijelove memorije u kojima su smješteni objekti razlikuje pomoæu pripadajuæe memorijske adrese. Da programer tijekom pisanja programa ne bi morao pamtiti memorijske adrese, svi programski jezici omoguæavaju da se vrijednosti objekata dohvaæaju preko simbolièkih naziva razumljivih inteligentnim biæima poput ljudi-programera. Gledano iz perspektive obiènog raèunala (lat. *computer vulgaris ex terae Tai-wan*), objekt je samo dio memorije u koji je pohranjena njegova vrijednost u binarnom obliku. *Tip* objekta, između ostalog, određuje raspored bitova prema kojem je taj objekt pohranjen u memoriju.

U programskom jeziku C++ pod objektima u užem smislu riječi obično se podrazumijevaju složeni tipovi podataka opisani pomoću posebnih “receptura” – klasa, što će biti opisano u kasnijim poglavljima. Jednostavni objekti koji pamte jedan cijeli ili realni broj se često nazivaju varijablama.

Da bi prevoditelj pravilno preveo naš izvorni C++ kôd u strojni jezik, svaku varijablu treba prije njena korištenja u kôdu *deklarirati*, odnosno jednoznačno odrediti njen tip. U "Našem trećem C++ programu" varijable su deklarirane u prvom retku glavne funkcije:

```
int a, b, c;
```

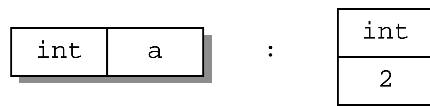
Tim deklaracijama je prevoditelju jasno dano do znanja da æe varijable `a`, `b` i `c` biti cjelobrojne – prevoditelj æe vrijednosti tih varijabli pohranjivati u memoriju prema svom pravilu za cjelobrojne podatke. Takoðer æe znati kako treba provesti operacije na njima. Prilikom deklaracije varijable treba takoðer paziti da se u istom dijelu programa (toènije *bloku naredbi*, vidi poglavlje 5.1) ne smiju deklarirati više puta varijable s istim imenom, èak i ako su one razlièitih tipova. Zato æe prilikom prevoðenja sljedeæeg kôda prevoditelj javiti pogrešku o višekratnoj deklaraciji varijable `a`:

```
int a, b, c;
int a;      // pogreška: ponovno korištenje naziva a
```

Varijable postaju realna zbiljnost tek kada im se pokuša pristupiti, na primjer kada im se pridruži vrijednost:

```
a = 2;
b = a;
c = a + b;
```

Prevoditelj tekar tada pridružuje memorijski prostor u koji se pohranjuju podaci. Postupak deklaracije varijable `a` i pridruživanja vrijednosti simbolièki možemo prikazati slikom 4.1. Lijevo od dvotoèke je kuæica koja simbolizira varijablu s njenim tipom i



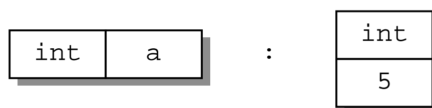
Slika 4.1. Deklaracija varijable i pridruživanje vrijednosti

imenom. Desno od dvotoèke je kuæica koja predstavlja objekt s njegovim tipom i konkretnom vrijednošæu. Ako nakon toga istoj varijabli `a` pridružimo novu vrijednost naredbom

```
a = 5;
```

tada se u memoriju raèunala na mjestu gdje je prije bio smješten broj 2 pohranjuje broj 5, kako je prikazano slikom 4.2.

Deklaracija varijable i pridruživanje vrijednosti mogu se obaviti u istom retku kôda.



Slika 4.2. Pridruživanje nove vrijednosti

Umjesto da pišemo poseban redak s deklaracijama varijabli `a`, `b` i `c` i zatim im pridružujemo vrijednosti, *inicijalizaciju* možemo provesti ovako:

```
int a = 2;
int b = a;
int c = a + b;
```

Za razliku od programskog jezika C u kojem sve deklaracije moraju biti na poèetku programa ili funkcije, prije prve naredbe, u C++ jeziku ne postoji takvo ogranièenje, pa deklaracija varijable može biti bilo gdje unutar programa. Štoviše, mnogi autori preporuèuju da se varijable deklariraju neposredno prije njihove prve primjene.

4.3. Operator pridruživanja

Operatorom pridruživanja mijenja se vrijednost nekog objekta, pri èemu tip objekta ostaje nepromijenjen. Najèešæi operator pridruživanja je znak jednakosti (`=`) kojim se objektu na lijevoj strani pridružuje neka vrijednost s desne strane. Ako su objekt s lijeve strane i vrijednost s desne strane razlièitih tipova, vrijednost se svodi na tip objekta prema definiranim pravilima konverzije, što æe biti objašnjeno u poglavlju 4.4 o tipovima podataka. Oèito je da s lijeve strane operatora pridruživanja mogu biti iskljuèivo promjenjivi objekti, pa se stoga ne može pisati ono što je inaèe matematièki korektno:

```
2 = 4 / 2          // pogreška!!!
3 * 4 = 12         // pogreška!!!
3.14159 = pi       // Bingooo!!! Treća pogreška!
```

Pokušamo li prevesti ovaj kôd, prevoditelj æe javiti pogreške. Objekti koji se smiju nalaziti s lijeve strane znaka pridruživanja nazivaju se *lvrijednosti* (engl. *lvalues*, kratica od *left-hand side values*). Pritom valja znati da se ne može svakoj lvrijednosti pridruživati nova vrijednost – neke varijable se mogu deklarirati kao konstantne (vidi poglavlje 4.4.4) i pokušaj promjene njihove vrijednosti prevoditelj æe naznaèiti kao pogrešku. Stoga se posebno govori o *promjenjivim lvrijednostima*. S desne strane operatora pridruživanja mogu biti i lvrijednosti i konstante.

Evo tipičnog primjera pridruživanja kod kojeg se ista varijabla nalazi i s lijeve i s desne strane znaka jednakosti:

```
int i = 5;
i = i + 3;
```

Naredbom u prvom retku deklarira se varijabla tipa `int`, te se njena vrijednost inicijalizira na 5. Dosljedno gledano, operator `=` ovdje nema znaèenje pridruživanja, jer se inicijalizacija varijable `i` provodi veæ tijekom prevođenja, a ne prilikom izvođenja programa. To znaèi da je broj 5 ugrađen u izvedbeni strojni kôd dobiven prevođenjem i povezivanjem programa. Obratimo, međutim, pažnju na drugu naredbu!

Matematički gledano, drugi redak u ovom primjeru je besmislen: nema broja koji bi bio jednak samom sebi uvećanom za 3! Ako ga pak gledamo kao uputu računalu što mu je činiti, onda taj redak treba početi čitati neposredno iza znaka pridruživanja: "uzmi vrijednost varijable `i`, dodaj joj broj 3....". Došli smo do kraja naredbe (znak `;`), pa se vraćamo na operator pridruživanja: "...i dobiveni zbroj s desne strane pridruži varijabli `i` koja se nalazi s lijeve strane znaka jednakosti". Nakon izvedene naredbe varijabla `i` imat će vrijednost 8.

Jezik C++ dozvoljava više operatora pridruživanja u istoj naredbi. Pritom pridruživanje ide od krajnjeg desnog operatora prema lijevo:

```
a = b = c = 0;
```

te ih je tako najsigurnije i èitati: "broj `0` pridruži varijabli `c`, èiju vrijednost pridruži varijabli `b`, èiju vrijednost pridruži varijabli `a`". Buduæi da se svakom od objekata lijevo od znaka `=` pridružuje neka vrijednost, svi objekti izuzev onih koji se nalaze desno od najdesnijeg znaka `=` moraju biti lvrijednosti:

```
a = b = c + d;      // OK!
a = b + 1 = c;      // pogreška: b + 1 nije lvrijednost
```

4.4. Tipovi podataka i operatori

U C++ jeziku ugrađeni su neki osnovni tipovi podataka i definirane operacije na njima. Za te su tipove precizno definirana pravila provjere i konverzije. *Pravila provjere tipa* (engl. *type-checking rules*) uoèavaju neispravne operacije na objektima, dok *pravila konverzije* određuju što æe se dogoditi ako neka operacija oèekuje jedan tip podataka, a umjesto njega se pojavi drugi. U sljedeæim poglavljima prvo æemo se upoznati s brojevima i operacijama na njima, da bismo kasnije prešli na pobrojenja, logièke vrijednosti i znakove.

4.4.1. Brojevi

U jeziku C++ ugrađena su u suštini dva osnovna tipa brojeva: cijeli brojevi (engl. *integers*) i realni brojevi (tzv. *brojevi s pomiènom decimalnom toèkom*, engl. *floating-point*). Najjednostavniji tip brojeva su cijeli brojevi – njih smo veæ upoznali u "Našem

treæem C++ programu”. Cjelobrojna varijabla deklarira se rijeèju `int` i njena æe vrijednost u memoriji raèunala obièno zauzeti dva *bajta* (engl. *byte*), tj. 16 bitova. Prvi bit je rezerviran za predznak, tako da preostaje 15 bitova za pohranu vrijednosti. Stoga se varijablom tipa `int` mogu obuhvatiti svi brojevi od najmanjeg broja (najveæeg negativnog broja)

$$-2^{15} = -32768$$

do najveæeg broja

$$2^{15} - 1 = 32767$$

Za veæinu praktiènih primjena taj opseg vrijednosti je dostatan. Međutim, ukaže li se potreba za veæim cijelim brojevima varijablu možemo deklarirati kao `long int`, ili kraæe samo `long`:

```
long int HrvataUDijasporiZaVrijemeIzbora = 3263456;
long ZrnacaPrasineNaMomRacunalu = 548234581;
    // mogao bih uzeti krpu za prašinu i svesti to na int!
```

Takva varijabla æe zauzeti u memoriji više prostora i operacije s njom æe dulje trajati.

Cjelobrojne konstante se mogu u izvornom kôdu prikazati u različitima brojevnim sustavima:

- dekadskom,
- oktalnom,
- heksadekadskom.

Dekadski prikaz je razumljiv veæini obiènih nehakerskih smrtnika koji se ne spuštaju na razinu raèunala. Međutim, kada treba raditi operacije nad pojedinim bitovima, oktalni i heksadekadski prikazi su daleko primjereniji.

U dekadskom brojevnom sustavu, koji koristimo svakodnevno, postoji 10 različitih znamenki (0, 1, 2,..., 9) čijom kombinacijom se dobiva bilo koji željeni višeznamenkasti broj. Pritom svaka znamenka ima deset puta veću težinu od znamenke do nje desno. U oktalnom brojevnom sustavu broj znamenki je ograničen na 8 (0, 1, 2,..., 7), tako da svaka znamenka ima osam puta veću težinu od svog desnog susjeda. Na primjer, 11 u oktalnom sustavu odgovara broju 9 u dekadskom sustavu ($11_8 = 9_{10}$). U heksadekadskom sustavu postoji 16 znamenki: 0, 1, 2,..., 9, A, B, C, D, E. Budući da za prikaz brojeva postoji samo 10 brojčanih simbola, za prikaz heksadekadskih znamenki iznad 9 koriste se prva slova engleskog alfabeta, tako da je $A_{16} = 10_{10}$, $B_{16} = 11_{10}$, itd. Oktalni i heksadekadski prikaz predstavljaju kompromis između dekadskog prikaza kojim se koriste ljudi i binarnog sustava kojim se podaci pohranjuju u memoriju računala. Naime, binarni prikaz pomoću samo dvije znamenke (0 i 1) iziskivao bi puno prostora u programskom kôdu, te bi bio nepregledan. Oktalni i heksadekadski prikazi iziskuju manje prostora i iskusnom korisniku omogućuju brzu pretvorbu brojeva iz dekadskog u binarni oblik i obrnuto.

Oktalne konstante se pišu tako da se ispred prve znamenke napiše broj `0` iza kojeg slijedi oktalni prikaz broja:

```
int SnjeguljicaIPatuljci = 010;   // odgovara dekadsko 8!
```

Heksadekadske konstante zapoèinju s `0x` ili `0X`:

```
int TucetPatuljaka = 0x0C;        // dekadsko 12
```

Slova za heksadekadske znamenke mogu biti velika ili mala.

Vodeća nula kod oktalnih brojeva uzrokom je čestih početničkih pogrešaka, jer korisnik obično smatra da će ju prevoditelj zanemariti i interpretirati broj kao obični dekadski.

Sve konstante koje zapoèinju s brojem `0` prevoditelj interpretira kao oktalne brojeve. To znaèi da æe nakon prevođenja `010` i `10` biti dva razlièita broja!

Još jednom uoèimo da æe bez obzira na brojevni sustav u kojem broj zadajemo, njegova vrijednost u memoriju biti pohranjena prema predlošku koji je određen deklaracijom pripadne varijable. To najbolje ilustrira sljedeæi primjer:

```
int i = 32;
cout << i << endl;

i = 040;                // oktalni prikaz broja 32
cout << i << endl;

i = 0x20;               // heksadekadski prikaz broja 32
cout << i << endl;
```

Bez obzira što smo varijabli `i` pridruživali broj 32 u tri razlièita prikaza, u sva tri sluèaja na zaslonu æe se ispisati isti broj, u dekadskom formatu.

Ako je potrebno računati s decimalnim brojevima, najčešće se koriste varijable tipa `float`:

```
float pi = 3.141593;
float brzinaSvjetlosti = 2.997925e8;
float nabojElektrona = -1.6E-19;
```

Prvi primjer odgovara uobièajenom naèinu prikaza brojeva s decimalnim zarezom. U drugom i treæem primjeru brojevi su prikazani u znanstvenoj notaciji, kao umnožak mantise i potencije na bazi 10 ($2{,}997925 \cdot 10^{8}$, odnosno $-1{,}6 \cdot 10^{-19}$). Kod prikaza u znanstvenoj notaciji, slovo 'e' koje razdvaja mantisu od eksponenta može biti veliko ili malo.

Praznine unutar broja, na primjer iza predznaka, ili između znamenki i slova '`e`' nisu dozvoljene.

Prevoditelj æe broj

```
float PlanckovaKonst = 6.626 e -34;     // pogreška
```

interpretirati samo do èetvrte znamenke. Prazninu koja slijedi prevoditelj æe shvatiti kao završetak broja, pa æe po nailasku na znak `e` javiti pogrešku.

Osim što je ograničen raspon vrijednosti koje se mogu prikazati `float` tipom (vidi tablicu 4.3), treba znati da je i broj decimalnih znamenki u mantisi također ograničen na 7 decimalnih mjesta†. Čak i ako se napiše broj s više znamenki, prevoditelj će zanemariti sve niže decimalne znamenke. Stoga će ispis varijabli

```
float pi_tocniji    = 3.141592654;
float pi_manjeTocan = 3.1415927;
```

dati potpuno isti broj! Usput spomenimo da su broj π i ostale znaèajnije matematièke konstante definirani u biblioteci `math.h`, i to na veæu toènost, tako da ukljuèivanjem te datoteke možete prikratiti muke traženja njihovih vrijednosti po raznim priruènicima i srednjoškolskim udžbenicima.

Ako točnost na sedam decimalnih znamenki ne zadovoljava ili ako se koriste brojevi veći od 10^{38} ili manji od 10^{-38}, tada se umjesto `float` mogu koristiti brojevi s dvostrukom točnošću tipa `double` koji pokrivaju opseg vrijednosti od $1.7 \cdot 10^{-308}$ do $1.7 \cdot 10^{308}$, ili `long double` koji pokrivaju još širi opseg od $3.4 \cdot 10^{-4932}$ do $1.1 \cdot 10^{4932}$. Brojevi veći od 10^{38} vjerojatno će vam zatrebati samo za neke astronomske proračune, ili ako ćete se baviti burzovnim mešetarenjem. Realno gledano, brojevi manji od 10^{-38} neće vam gotovo nikada trebati, osim ako želite izračunati vjerojatnost da dobijete glavni zgoditak na lutriji.

U tablici 4.3 dane su tipične duljine ugrađenih brojevnih tipova u bajtovima, rasponi vrijednosti koji se njima mogu obuhvatiti, a za decimalne brojeve i broj točnih decimalnih znamenki. Duljina memorijskog prostora i opseg vrijednosti koje određeni tip varijable zauzima nisu standardizirani i variraju ovisno o prevoditelju i o platformi za koju se prevoditelj koristi. Stoga čitatelju preporučujemo da za svaki slučaj konzultira dokumentaciju uz prevoditelj koji koristi. Relevantni podaci za cjelobrojne tipove mogu se naći u datoteci `limits.h`, a za decimalne tipove podataka u `values.h`.

Ako želite sami provjeriti veličinu pojedinog tipa, to možete učiniti i pomoću `sizeof` operatora:

```
cout << "Veličina cijelih brojeva: " << sizeof(int);
cout << "Veličina realnih brojeva: " << sizeof(float);
long double masaSunca = 2e30;
cout << "Veličina long double: " << sizeof(masaSunca);
```

† Broj toènih znamenki ovisi o prevoditelju i o raèunalu. U veæini sluèajeva broj toènih znamenki i dozvoljene pogreške pri osnovnim aritmetièkim operacijama ravnaju se po IEEE standardu 754 za prikaz brojeva s pomiènim decimalnim zarezom.

Tablica 4.3. Ugrađeni brojevni tipovi, njihove tipiène duljine i rasponi vrijednosti

tip konstante	bajtova	raspon vrijednosti	toènost
`char`	1		
`short int`	2	–32768 do 32767	
`int`	2 (4)	–32768 do 32767 –2147483648 do 2147483647	
`long int`	4	–2147483648 do 2147483647	
`float`	4	$-3{,}4\cdot10^{38}$ do $-3{,}4\cdot10^{-38}$ i $3{,}4\cdot10^{-38}$ do $3{,}4\cdot10^{38}$	7 dec. znamenki
`double`	8	$-1{,}7\cdot10^{308}$ do $-1{,}7\cdot10^{-308}$ i $1{,}7\cdot10^{-308}$ do $1{,}7\cdot10^{308}$	15 dec. znamenki
`long double`	10	$-1{,}1\cdot10^{4932}$ do $-3{,}4\cdot10^{-4932}$ i $3{,}4\cdot10^{-4932}$ do $1{,}1\cdot10^{4932}$	18 dec. znamenki

Svi navedeni brojevni tipovi mogu biti deklarirani i bez predznaka, dodavanjem rijeèi `unsigned` ispred njihove deklaracije:

```
unsigned int i = 40000;
unsigned long int li;
unsigned long double BrojZvijezdaUSvemiru;
```

U tom sluèaju æe prevoditelj bit za predznak upotrijebiti kao dodatnu binarnu znamenku, pa se najveæa moguæa vrijednost udvostruèuje u odnosu na varijable s predznakom, ali se naravno ogranièava samo na pozitivne vrijednosti. Pridružimo li `unsigned` varijabli negativan broj, ona æe poprimiti vrijednost nekog pozitivnog broja. Na primjer, programski slijed

```
unsigned int = -1;
cout << i << endl;
```

æe za varijablu `i` ispisati broj 65535 (uoèimo da je on za 1 manji od najveæeg dozvoljenog `unsigned int` broja 65536). Zanimljivo da prevoditelj za gornji kôd neæe javiti pogrešku.

Prevoditelj ne prijavljuje pogrešku ako se `unsigned` varijabli pridruži negativna konstanta – korisnik mora sam paziti da se to ne dogodi!

Stoga ni za živu glavu nemojte u programu za evidenciju stanja na tekuæem raèunu koristiti `unsigned` varijable. U protivnom se lako može dogoditi da ostanete bez èekovne iskaznice, a svoj bijes iskalite na raèunalu, ni krivom ni dužnom.

4.4.2. Aritmetièki operatori

Da bismo objekte u programu mogli mijenjati, na njih treba primijeniti odgovarajuæe operacije. Za ugraðene tipove podataka definirani su osnovni operatori, poput zbrajanja, oduzimanja, množenja i dijeljenja (vidi tablicu 4.4). Ti operatori se mogu podijeliti na *unarne*, koji djeluju samo na jedan objekt, te na *binarne* za koje su neophodna dva

Tablica 4.4. Aritmetièki operatori

	`+x`	unarni plus
	`-x`	unarni minus
unarni operatori	`x++`	uveæaj nakon
	`++x`	uveæaj prije
	`x--`	umanji nakon
	`--x`	umanji prije
	`x + y`	zbrajanje
	`x - y`	oduzimanje
binarni operatori	`x * y`	množenje
	`x / y`	dijeljenje
	`x % y`	modulo

objekta. Osim unarnog plusa i unarnog minusa koji mijenjaju predznak broja, u jeziku C++ definirani su još i unarni operatori za uveæavanje (*inkrementiranje*) i umanjivanje vrijednosti (*dekrementiranje*) broja. Operator `++` uveæat æe vrijednost varijable za 1, dok æe operator `--`) umanjiti vrijednost varijable za 1:

```
int i = 0;
++i;                            // uveća za 1
cout << i << endl;              // ispisuje 1
--i;                            // umanji za 1
cout << i << endl;              // ispisuje 0
```

Pritom valja uoèiti razliku izmeðu operatora kada je on napisan ispred varijable i operatora kada je on napisan iza nje. U prvom sluèaju (*prefiks* operator), vrijednost varijable æe se prvo uveæati ili umanjiti, a potom æe biti dohvaæena njena vrijednost. U drugom sluèaju (*postfiks* operator) je obrnuto: prvo se dohvati vrijednost varijable, a tek onda slijedi promjena. To najbolje doèarava sljedeæi kôd:

```
int i = 1;
cout << i << endl;      // ispiši za svaki slučaj
cout << (++i) << endl;  // prvo poveća, pa ispisuje 2
cout << i << endl;      // ispisuje opet, za svaki slučaj
cout << (i++) << endl;  // prvo ispisuje, a tek onda uveća
cout << i << endl;      // vidi, stvarno je uvećao!
```

Na raspolaganju imamo pet binarnih aritmetièkih operatora: za zbrajanje, oduzimanje, množenje, dijeljenje i *modulo* operator:

```
float a = 2.;
float b = 3.;
cout << (a + b) << endl;    // ispisuje 5.
cout << (a - b) << endl;    // ispisuje -1.
cout << (a * b) << endl;    // ispisuje 6.
cout << (a / b) << endl;    // ispisuje 0.666667
```

Operator modulo kao rezultat vraæa ostatak dijeljenja dva cijela broja:

```
int i = 6;
int j = 4;
cout << (i % j) << endl;    // ispisuje 2
```

On se vrlo èesto koristi za ispitivanje djeljivosti cijelih brojeva: ako su brojevi djeljivi, ostatak nakon dijeljenja æe biti nula.

Za razliku od većine matematički orijentiranih jezika, jezik C++ nema ugrađeni operator za potenciranje, već programer mora sam pisati funkciju ili koristiti funkciju `pow()`. iz standardne matematičke biblioteke deklarirane u datoteci zaglavlja `math.h`.

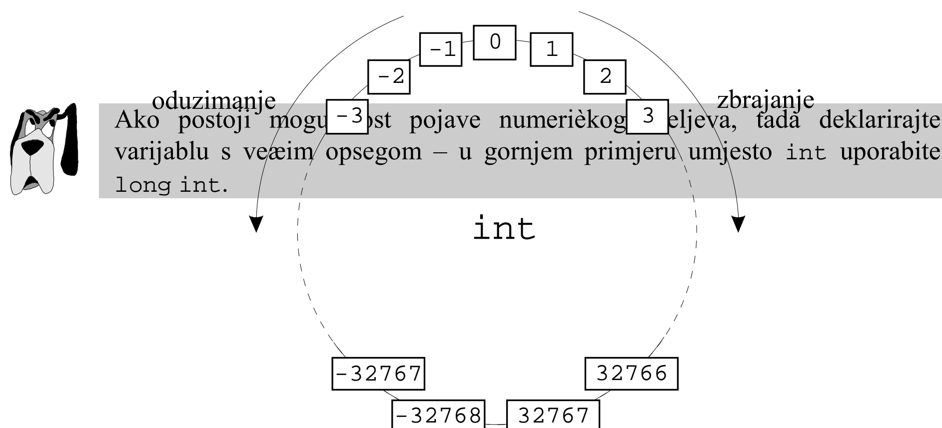
Valja uočiti dva suštinska problema vezana uz aritmetičke operatore. Prvi problem vezan je uz pojavu numeričkog preljeva, kada uslijed neke operacije rezultat nadmaši opseg koji dotični tip objekta pokriva. Drugi problem sadržan je u pitanju "kakvog će tipa biti rezultat binarne operacije s dva broja različitih tipova?".

Razmotrimo prvo pojavu brojčanog preljeva. U donjem programu će prvo biti ispisan broj 32767, što je najveći mogući broj tipa `int`. Izvođenjem naredbe u trećem retku, na zaslonu računala će se umjesto očekivanih 32768 ispisati –32768, tj. najveći negativni `int`!

```
int i = 32766;
cout << (++i) << endl;
cout << (++i) << endl;
```

Uzrok tome je preljev podataka do kojeg došlo zbog toga što oèekivani rezultat više ne stane u 15 bita predviđenih za `int` varijablu. Podaci koji su se "prelili" ušli su u bit za predznak (zato je rezultat negativan), a raspored preostalih 15 bitova daje broj koji odgovara upravo onom što nam je raèunalo ispisalo. Slièno æe se dogoditi oduzmete li od najveæeg negativnog broja neki broj. Ovo se može ilustrirati brojevnom kružnicom

Ako postoji mogu[illegible]st pojave numerièkog [illegible]eljeva, tada deklarirajte varijablu s veæim opsegom – u gornjem primjeru umjesto `int` uporabite `long int`.



Slika 4.3. Prikaz preljeva na brojevnoj kružnici

na kojoj zbrajanje odgovara kretanju po kružnici u smjeru kazaljke na satu, a oduzimanje odgovara kretanju u suprotnom smjeru (slika 4.3).
Gornja situacija može se poistovjetiti s kupovinom automobila na sajmu rabljenih automobila. Naravno da auto star 10 godina nije prešao samo 5000 kilometara koliko piše na brojaèu kilometara (kilometer-cajgeru), veæ je nakon 99999 kilometara brojaè ponovno krenuo od 00000. Buduæi da brojaè ima mjesta za 5 znamenki najviša znamenka se izgubila! Suštinska razlika je jedino u tome da je kod automobila do preljeva došlo "hardverskom" intervencijom prethodnog vlasnika, dok u programu do preljeva obièno dolazi zbog nepažnje programera pri izradi programa.

Drugo pitanje koje se nameće jest kakvog će biti tipa rezultat binarne operacije na dva broja. Za ugrađene tipove točno su određena pravila *uobičajene aritmetičke konverzije*. Ako su oba operanda istog tipa, tada je i rezultat tog tipa, a ako su operandi različitih tipova, tada se oni prije operacije svode na *zajednički tip* (to je obično složeniji tip), prema sljedećim pravilima:

1. Ako je jedan od operanada tipa `long double`, tada se i drugi operand pretvara u `long double`.
2. Inaèe, ako je jedan od operanada tipa `double`, tada se i drugi operand pretvara u `double`.
3. Inaèe, ako je jedan od operanada tipa `float`, tada se i drugi operand pretvara u `float`.
4. Inaèe, se provodi *cjelobrojna promocija* (engl. *integral promotion*) oba operanda (ovo je bitno samo za operande tipa `bool`, `wchar_t` i pobrojenja, tako da æemo o cjelobrojnoj promociji govoriti u odgovarajuæim poglavljima o tim tipovima).
5. Potom, ako je jedan od operanada `unsigned long`, tada se i drugi operand pretvara u `unsigned long`.
6. U protivnom, ako je jedan od operanada tipa `long int`, a drugi operand tipa `unsigned int`, ako `long int` može obuhvatiti sve vrijednosti `unsigned int`,

`unsigned int` se pretvara u `long int`; inaèe se oba operanda pretvaraju u `unsigned long int`.

7. Inaèe, ako je jedan od operanada tipa `long`, tada se i drugi operand pretvara u `long`.
8. Inaèe, ako je jedan od operanada `unsigned`, tada se i drugi operand pretvara u `unsigned`.

Na primjer, izvoðenje kôda

```
int i = 3;
float a = 0.5;
cout << (a * i) << endl;
```

uzrokovat æe ispis broja `1.5` na zaslonu, jer je od tipova `int` i `float` složeniji tip `float`. Cjelobrojnu varijablu `i` prevoditelj prije množenja pretvara u `float` (prema pravilu u toèki 3), tako da se provodi množenje dva broja s pomiènom toèkom, pa je i rezultat tipa `float`. Da smo umnožak prije ispisa pridružili nekoj cjelobrojnoj varijabli

```
int c = a * i;
cout << c << endl;
```

dobili bismo ispis samo cjelobrojnog dijela rezultata, tj. broj `1`. Decimalni dio rezultata se gubi prilikom pridruživanja umnoška cjelobrojnoj varijabli `c`.

Problem konverzije brojevnih tipova najjače je izražen kod dijeljenja cijelih brojeva, što početnicima (ali i nepažljivim C++ "guruima") zna prouzročiti dosta glavobolje. Pogledajmo sljedeći jednostavni primjer:

```
int Brojnik = 1;
int Nazivnik = 4;
float Kvocijent = Brojnik / Nazivnik;
cout << Kvocijent << endl;
```

Suprotno svim pravilima zapadnoeuropske klasiène matematike, na zaslonu æe se kao rezultat ispisati `0.`, tj. najobiènija nula! Koristi li vaše raèunalo aboriðanski brojevni sustav, koji poznaje samo tri broja (*jedan*, *dva* i *puno*)? Iako smo rezultat dijeljenja pridružili `float` varijabli, pri izvoðenju programa to pridruživanje slijedi tek nakon što je operacija dijeljenja dva cijela broja bila završena (ili, u duhu južnoslavenske narodne poezije, *Kasno* `float` *na Dijeljenje stiže!*). Buduæi da su obje varijable, `Brojnik` i `Nazivnik` cjelobrojne, prevoditelj provodi cjelobrojno dijeljenje u kojem se zanemaruju decimalna mjesta. Stoga je rezultat cjelobrojni dio kvocijenta varijabli `Brojnik` i `Nazivnik` (0.25), a to je 0. Slièna je situacija i kada dijelimo cjelobrojne konstante:

```
float DiskutabilniKvocijent = 3 / 2;
```

Brojeve 3 i 2 prevoditelj æe shvatiti kao cijele, jer ne sadrže decimalnu toèku. Zato æe primijeniti cjelobrojni operator `/`, pa æe rezultat toga dijeljenja biti cijeli broj 1.

Da bi se izbjegle sve nedoumice ovakve vrste, dobra je programerska navada sve konstante koje trebaju biti s pomiènom decimalnom toèkom (`float` i `double`) pisati s decimalnom toèkom, èak i kada nemaju decimalnih mjesta:

Evo pravilnog naèina da se provede željeno dijeljenje:

```
float TocniKvocijent = 3. / 2.;
```

Dovoljno bi bilo decimalnu toèku staviti samo uz jedan od operanada – prema pravilima aritmetièke konverzije i drugi bi operand bio sveden na `float` tip. Iako je kôd ovako dulji, izaziva manje nedoumica i stoga svakom poèetniku preporuèujemo da ne štedi na decimalnim toèkama.

Neupućeni početnik postavit će logično pitanje zašto uopće postoji razlika između cjelobrojnog dijeljenja i dijeljenja decimalnih brojeva. Za programera bi bilo najjednostavnije kada bi se oba operanda bez obzira na tip, prije primjene operatora svela na `float` (ili još bolje, na `double`), na takve modificirane operande primijenio željeni operator, a dobiveni rezultat se pretvorio u zajednički tip. U tom slučaju programer uopće ne bi trebao paziti kojeg su tipa operandi – rezultat bi uvijek bio korektan (osim ako nemate procesor s pogreškom, što i nije nemoguće). Nedostatak ovakvog pristupa jest njegova neefikasnost. Zamislimo da treba zbrojiti dva broja tipa `int`! U gornjem "programer-ne-treba-paziti" pristupu, izvedbeni kôd dobiven nakon prevođenja trebao bi prvo jedan cjelobrojni operand pretvoriti u `float`. Budući da se različiti tipovi podataka pohranjuju u memoriju računala na različite načine, ta pretvorba nije trivijalna, već iziskuje određeni broj strojnih instrukcija. Istu pretvorbu treba ponoviti za drugi operand. Već sada uočavamo dvije operacije konverzije tipa koje kod izravnog zbrajanja cijelih brojeva ne postoje! Štoviše, samo zbrajanje se provodi na dva `float`-a, koji u memoriji zauzimaju veći prostor od `int`-a. Takvo zbrajanje iziskivat će puno više strojnih instrukcija i strojnog vremena, ne samo zbog veće duljine `float` brojeva u memoriji, već i zbog složenijeg načina na koji su oni pohranjeni. Za mali broj operacija korisnik zasigurno ne bi osjetio razliku u izvođenju programa s izravno primijenjenim operatorima i operatorima *à la* "programer-ne-treba-paziti". Međutim, u složenim programima s nekoliko tisuća ili milijuna operacija, ta razlika može biti zamjetna, a često i kritična. U krajnjoj liniji, shvatimo da prevoditelj poput vjernog psa nastoji što brže ispuniti gospodarevu zapovijed, ne pazeći da li će pritom protrčati kroz upravo uređen susjedin cvjetnjak ili zagaziti u svježe betoniran pločnik.

Zadatak. *Što će ispisati sljedeće naredbe:*

```
int a = 10;
float b = 10.;

cout << a / 3 << endl;
cout << b / 3 << endl;
```

Zadatak. *Da li će postojati razlika pri ispisu u donjim naredbama (varijable* `a` *i* `b` *deklarirane su u prethodnom zadatku):*

```
float c = a / 3;
cout << c * b << endl;
c = a / 3.;
cout << c * b << endl;
c = b * a;
cout << c / 3 << endl;
```

4.4.3. Operator dodjele tipa

Što uèiniti želimo li podijeliti dvije cjelobrojne varijable, a da pritom ne izgubimo decimalna mjesta? Dodavanje decimalne toèke iza imena varijable nema smisla, jer æe prevoditelj javiti pogrešku. Za eksplicitnu promjenu tipa varijable valja primijeniti operator *dodjele tipa* (engl. *type cast*, kraæe samo *cast*):

```
int Brojnik = 1;
int Nazivnik = 3;
float TocniKvocijent = (float)Brojnik / (float)Nazivnik;
```

Navoðenjem kljuènih rijeèi `float` u zagradama ispred operanada njihove vrijednosti se pretvaraju u decimalne brojeve prije operacije dijeljenja, tako da je rezultat korektan. I ovdje bi bilo dovoljno operator dodjele tipa primijeniti samo na jedan operand – prema pravilima aritmetièke konverzije i drugi bi operand bio sveden na `float` tip. Da ne bi bilo zabune, same varijable `Brojnik` i `Nazivnik` i nadalje ostaju tipa `int`, tako da æe njihovo naknadno dijeljenje

```
float OpetKriviKvocijent = Brojnik / Nazivnik;
```

opet kao rezultat dati kvocijent cjelobrojnog dijeljenja.

Jezik C++ dozvoljava i funkcijski oblik dodjele tipa u kojem se tip navodi ispred zagrade, a ime varijable u zagradi:

```
float TocniKvocijent2 = float(Brojnik) / float(Nazivnik);
```

Iako je ovakav oblik "logièniji" i za poèetnika prihvatljiviji, on nije primjenjiv na pokazivaèe (odjeljak 4.2), tako da ne preporuèujemo njegovu uporabu.

Operator dodjele tipa može se koristiti i u obrnutom slučaju, kada želimo iz decimalnog broja izlučiti samo cjelobrojne znamenke.

Zadatak. *Odredite koje će od navedenih naredbi za ispis:*

```
int a = 10000;
int b = 10;
```

```
long c = a * b;

cout << c << endl;
cout << (a * b) << endl;
cout << ((float)a * b) << endl;
cout << (long)(a * b) << endl;
cout << (a * (long)b) << endl;
```

dati ispravan umnožak, tj. 100000.

Zadatak. *Odredite što će se ispisati na zaslonu po izvođenju sljedećeg kôda:*

```
float a = 2.71;
float b = (int)a;
cout << b << endl;
```

4.4.4. Dodjeljivanje tipa brojèanim konstantama

Kada se u kôdu pojavljuju brojèane konstante, prevoditelj ih pohranjuje u formatu nekog od osnovnih tipova. Tako s brojevima koji sadrže decimalnu toèku ili slovo `e`, odnosno `E`, prevoditelj barata kao s podacima tipa `double`, dok sve ostale brojeve tretira kao `int`. Operatore dodjele tipa moguæe primijeniti i na konstante, na primjer:

```
cout << (long)10 << endl;            // long int
cout << (unsigned)60000 << endl;     // unsigned int
```

Èešæe se za specificiranje konstanti koriste *sufiksi*, posebni znakovi kojima se eksplicitno odreðuje tip brojèane konstante (vidi tablicu 4.5). Tako æe sufiks `l`, odnosno `L` cjelobrojnu konstantu pretvoriti u `long`, a konstantu s decimalnom toèkom u `double`:

```
long double a = 1.602e-4583L / 645672L;
            // (long double) / long
```

dok æe sufiksi `u`, odnosno `U` cjelobrojne konstante pretvoriti u `unsigned int`:

Tablica 4.5. Djelovanje sufiksa na brojèane konstante

broj ispred sufiksa	sufiks	rezultirajuæi tip
cijeli		`int`
	`L, l`	`long int`
	`U, u`	`unsigned int`
decimalni		`double`
	`F, f`	`float`
	`L, l`	`long double`

```
cout << 65000U << endl;     // unsigned int
```

Sufiks `f`, odnosno `F` æe konstantu s decimalnom toèkom pretvoriti u `float`:

```
float b = 1.234F;
```

Velika i mala slova sufiksa su potpuno ravnopravna. U svakom sluèaju je preglednije koristiti veliko slovo `L`, da bi se izbjegla zabuna zbog slKènosti slova `l` i broja 1. Također, sufiksi `L` (`l`) i `U` (`u`) za cjelobrojne podatke mogu se međusobno kombinirati u bilo kojem redoslijedu (`LU`, `UL`, `Ul`, `uL`, `ul`, `lu`...) – rezultat æe uvijek biti `unsigned long int`.

Sufiksi se rijetko koriste, budući da u većini slučajeva prevoditelj obavlja sam neophodne konverzije, prema već spomenutim pravilima. Izuzetak je, naravno pretvorba `double` u `float` koju provodi sufiks `F`.

4.4.5. Simbolièke konstante

U programima se redovito koriste simbolièke velièine èija se vrijednost tijekom izvođenja ne želi mijenjati. To mogu biti fizièke ili matematièke konstante, ali i parametri poput maksimalnog broja prozora ili maksimalne duljine znakovnog niza, koji se namještaju prije prevođenja kôda i ulaze u izvedbeni kôd kao konstante.

Zamislimo da za neki zadani polumjer želimo izračunati i ispisati opseg kružnice, površinu kruga, oplošje i volumen kugle. Pri računanju sve četiri veličine treba nam Ludolfov broj π=3,14159...:

```
float Opseg = 2. * r * 3.14159265359;
float Povrsina = r * r * 3.14159265359;
double Oplosje = 4. * r * r * 3.14159265359;
double Volumen = 4. / 3. * r * r * r * 3.14159265359;
```

Naravno da bi jednostavnije i pouzdanije bilo definirati zasebnu varijablu koja æe sadržavati broj π:

```
double pi = 3.14159265359;
float Opseg = 2. * r * pi;
float Povrsina = r * r * pi;
double Oplosje = 4. * r * r * pi;
double Volumen = 4. / 3. * r * r * r * pi;
```

Manja je vjerojatnost da æemo pogriješiti prilikom utipkavanja samo jednog broja, a osim toga, ako se (kojim sluèajem, jednog dana) promijeni vrijednost broja π, bit æe lakše ispraviti ga kada je definiran samo na jednom mjestu, nego raditi pretraživanja i zamjene brojeva po cijelom izvornom kôdu.

Pretvaranjem konstantnih veličina u varijable izlažemo ih pogibelji od nenamjerne promjene vrijednosti. Nakon izvjesnog vremena jednostavno zaboravite da dotična varijabla predstavlja konstantu, te negdje u kôdu dodate naredbu

```
pi = 2 * pi;
```

kojom ste promijenili vrijednost varijable `pi`. Prevoditelj vas neæe upozoriti (*"Opet taj prokleti kompjutor!"*) i dobit æete da zemaljska kugla ima dva puta veæi volumen nego što ga je imala prošli puta kada ste koristili isti program, ali bez inkriminirane naredbe. Koristite li program za odreðivanje putanje svemirskog broda, ovakva pogreška sigurno æe rezultirati odašiljanjem broda u bespuæa svemirske zbiljnosti.

Da bismo izbjegli ovakve peripetije, na raspolaganju nam je kvalifikator `const` kojim se prevoditelju daje na znanje da varijabla mora biti nepromjenjiva. Na svaki pokušaj promjene vrijednosti takve varijable, prevoditelj će javiti pogrešku:

```
const double pi = 3.14159265359;
pi = 2 * pi;                        // sada je to pogreška!
```

Drugi èesto korišteni pristup zasniva se na pretprocesorskoj naredbi `#define`:

```
#define PI 3.14159265359
```

Ona daje uputu prevoditelju da, prije nego što zapoène prevoðenje izvornog kôda, sve pojave prvog niza (`PI`) zamijeni drugim nizom znakova (`3.14159265359`). Stoga æe prevoditelj naredbe

```
double Volumen = 4. / 3. * r * r * r * PI;
PI = 2 * PI;                        // pogreška
```

interpretirati kao da se u njima umjesto simbola `PI` nalazi odgovarajuæi broj. U drugoj naredbi æe javiti pogrešku, jer se taj broj našao s lijeve strane operatora pridruživanja. Valja napomenuti da se ove zamjene neæe odraziti u izvornom kôdu i on æe nakon prevoðenja ostati nepromijenjen.

Na prvi pogled nema razlike između pristupa – oba pristupa će osigurati prijavu pogreške pri pokušaju promjene konstante. Razlika postaje očita tek kada pokušate program ispraviti koristeći program za simboličko lociranje pogrešaka (engl. *debugger*, doslovni prijevod bio bi *istjerivač stjenica*, odnosno *stjenični terminator*). Ako ste konstantu definirali pretprocesorskom naredbom, njeno ime neće postojati u simboličkoj tablici koju prevoditelj generira, jer je prije prevođenja svaka pojava imena nadomještena brojem. Ne možete čak ni provjeriti da li ste možda pogriješili prilikom poziva imena.

Konstante definirane pomoæu deklaracije `const` dohvatljive su i iz programa za simbolièko lociranje pogrešaka.

Napomenimo da vrijednost simbolièke konstante mora biti inicijalizirana prilikom deklaracije. U protivnom æemo dobiti poruku o pogreški:

```
const float mojaMalaKonstanta;  // pogreška
```

Prilikom inicijalizacije vrijednosti nismo ogranièeni na brojeve – možemo pridružiti i vrijednost neke prethodno definirane varijable. Vrijednost koju pridružujemo konstanti ne mora biti èak ni zapisana u kôdu:

```
float a;
cin >> a;
const float DvijeTrecine = a;
```

4.4.6. Kvalifikator `volatile`

U prethodnom odsjeèku smo deklarirali konstantne objekte èime smo ih zaštitili od neovlaštenih promjena. Svi ostali objekti su promjenjivi (engl. *volatile*). Promjenjivost objekta se može naglasiti tako da se ispred tipa umetne kljuèna rijeè `volatile`:

```
volatile int promjenjivica;
```

Time se prevoditelju daje na znanje da se vrijednost varijable može promijeniti njemu nedokuèivim naèinima, te da zbog toga mora iskljuèiti sve optimizacije kôda prilikom pristupa.

Valja razjasniti koji su to načini promjene koji su izvan prevoditeljevog znanja. Na primjer, ako razvijamo sistemski program, vrijednost memorijske adrese se može promijeniti unutar obrade *prekida* (engl. *interrupt*) – time prekidna rutina može signalizirati programu da je određeni uvjet zadovoljen. U tom slučaju prevoditelj ne smije optimizirati pristup navedenoj varijabli. Na primjer:

```
int izadjiVan = 0;
while (!izadjiVan) {
    // čekaj dok se vrijednost izadjiVan ne postavi na 1
}
```

U gornjem primjeru prevoditelj analizom petlje može zakljuèiti da se varijabla `izadjiVan` ne mijenja unutar petlje, te æe optimizirati izvođenje tako da se vrijednost varijable `izadjiVan` uopæe ne testira. Prekidna rutina koja æe eventualno postaviti vrijednost `izadjiVan` na 1 zbog toga neæe okonèati petlju. Da bismo to sprijeèili, `izadjiVan` moramo deklarirati kao `volatile`:

```
volatile int izadjiVan;
```

4.4.7. Pobrojenja

Ponekad su varijable u programu elementi pojmovnih skupova, tako da je takvim skupovima i njihovim elementima zgodno pridijeliti lakopamtiva imena. Za takve sluèajeve obièno se koriste *pobrojani tipovi* (engl. *enumerated types*):

```
enum dani {ponedjeljak, utorak, srijeda,
           cetvrtak, petak, subota, nedjelja};
```

Ovom deklaracijom uvodi se novi tip podataka `dani`, te sedam nepromjenjivih identifikatora (`ponedjeljak`, `utorak`,...) toga tipa. Prvom identifikatoru prevoditelj pridjeljuje vrijednost 0, drugom 1, itd. Sada možemo definirati varijablu tipa `dani`, te joj pridružiti neku od vrijednosti iz niza:

```
dani HvalaBoguDanasJe = petak;
dani ZakajJaNeVolim = ponedjeljak;
```

Naredbom

```
cout << HvalaBoguDanasJe << endl;
```

na zaslonu se ispisuje cjelobrojni ekvivalent za `petak`, tj. broj 4.

Varijable tipa `dani` mogli smo deklarirati i neposredno uz definiciju pobrojanog tipa:

```
enum dani{ponedjeljak, utorak, srijeda, cetvrtak,
          petak, subota, nedjelja} ThankGodItIs, SunnyDay;

ThankGodItIs = petak;
SunnyDay = nedjelja;
```

U pobrojanim nizovima podrazumijevana vrijednost prve varijable je 0. Želimo li da niz poèinje nekom drugom vrijednošæu, treba eksplicitno pridijeliti željenu vrijednost:

```
enum dani {ponedjeljak = 1, utorak, srijeda,
           cetvrtak, petak, subota, nedjelja};
```

Identifikator `utorak` poprima vrijednost 2, `srijeda` 3, itd. U ovom sluèaju, kôd

```
dani ZecUvijekDolaziU = nedjelja;
cout << ZecUvijekDolaziU << endl;
```

na zaslonu ispisuje broj 7.

Eksplicitno pridjeljivanje može se primijeniti i na bilo koji od ostalih članova, s time da slijedeći član, ako njegova vrijednost nije eksplicitno definirana, ima za 1 veću vrijednost:

```
enum likovi {kruznica = 0,
             trokut = 3,
             pravokutnik = 4,
             kvadrat = 4,
             cetverokut = 4,
             peterokut,
             sedmerokut = cetverokut + 3};

cout << kruznica << endl;       // ispisuje 0
cout << trokut << endl;         // ispisuje 3
cout << kvadrat << endl;        // ispisuje 4
cout << peterokut << endl;      // ispisuje 5
cout << sedmerokut << endl;     // ispisuje 7
```

Ako nam ne treba više varijabli tog tipa, ime tipa iza rijeèi `enum` može se izostaviti:

```
enum {NE = 0, DA} YesMyBabyNo, ShouldIStayOrShouldIGo;
enum {TRUE = 1, FALSE = 0};
```

Pobrojane vrijednosti mogu se koristiti u aritmetièkim izrazima. U takvim sluèajevima se prvo provodi *cjelobrojna promocija* (engl. *integral promotion*) pobrojane vrijednosti – ona se pretvara u prvi moguæi cjelobrojni tip naveden u slijedu: `int`, `unsigned int`, `long` ili `unsigned long`. Na primjer:

```
enum {DVOSTRUKI = 2, TROSTRUKI};
int i = 5;
float pi = 3.14159;
cout << DVOSTRUKI * i << endl;      // ispisuje 10
cout << TROSTRUKI * pi << endl;     // ispisuje 9.42477
```

Međutim, pobrojanim tipovima ne mogu se pridruživati cjelobrojne vrijednosti, pa æe prevoditelj za sljedeæi primjer javiti pogrešku:

```
dani VelikiPetak = petak;
dani DolaziZec = VelikiPetak + 2;       // pogreška
```

Prilikom zbrajanja pobrojeni tip `VelikiPetak` svodi se na tip zajednièki sa cijelim brojem 2 (a to je `int`). Dobiveni rezultat tipa `int` treba pridružiti pobrojenom tipu, što rezultira pogreškom prilikom prevođenja[†].

[†] Programski jezik C dozvoljava da se pobrojanom tipu pridruži `int` vrijednost. Zato, radi prenosivosti kôda pisanog u programskom jeziku C, te kôda pisanog u starijim varijantama jezika C++, ANSI/ISO C++ standard dozvoljava da se u nekim implementacijama prevoditelja omoguæi to pridruživanje, uz napomenu da to može prouzroèiti neželjene efekte.

4.4.8. Logièki tipovi i operatori

Logièki podaci su takvi podaci koji mogu poprimiti samo dvije vrijednosti, na primjer: da/ne, istina/laž, dan/noæ. Jezik C++ za prikaz podataka logièkog tipa ima ugraðen tip `bool`, koji može poprimiti vrijednosti `true` (engl. *true – toèno*) ili `false` (engl. *false – pogrešno*)[†]:

```
bool JeLiDanasNedjelja = true;
bool SunceSije = false;
```

Pri ispisu logièkih tipova, te pri njihovom korištenju u aritmetièkim izrazima, logièki tipovi se pravilima cjelobrojne promocije (vidi prethodno poglavlje) pretvaraju u `int`: `true` se pretvara u cjelobrojni 1, a `false` u 0. Isto tako, logièkim varijablama se mogu pridruživati aritmetièki tipovi: u tom sluèaju se vrijednosti razlièite od nule pretvaraju u `true`, a nula se pretvara u `false`.

Gornja svojstva tipa `bool` omogućavaju da se za predstavljanje logičkih podataka koriste i cijeli brojevi. Broj 0 u tom slučaju odgovara logičkoj neistini, a bilo koji broj različit od nule logičkoj istini. Iako je za logičku istinu na raspolaganju vrlo široki raspon cijelih brojeva (zlobnici bi rekli da ima više istina), ona se ipak najčešće zastupa brojem 1. Ovakvo predstavljanje logičkih podataka je vrlo često, budući da je tip `bool` vrlo kasno uveden u standard jezika C++.

Tablica 4.6. Logièki operatori

`!x`	logièka negacija
`x && y`	logièki i
`x || y`	logièki ili

Za logičke podatke definirana su svega tri operatora: `!` (*logička negacija*), `&&` (*logički i*), te `||` (*logički ili*) (tablica 4.6). Logička negacija je unarni operator koji mijenja logičku vrijednost varijable: istinu pretvara u neistinu i obrnuto. Logički *i* daje kao rezultat istinu samo ako su oba operanda istinita; radi boljeg razumijevanja u tablici 4.7 dani su rezultati logičkog *i* za sve moguće vrijednosti oba operanda. Logički *ili* daje istinu ako je bilo koji od operanada istinit (vidi tablicu 4.8).

Razmotrimo primjenu logičkih operatora na sljedećem primjeru:

```
enum Logicki{NEISTINA, ISTINA, DRUGAISTINA = 124};
Logicki a = NEISTINA;
```

Tablica 4.7. Stanja za logièki *i*

a		*b*		*a* .i. *b*	
toèno	(1)	toèno	(1)	toèno	(1)
toèno	(1)	pogrešno	(0)	pogrešno	(0)
pogrešno	(0)	toèno	(1)	pogrešno	(0)
pogrešno	(0)	pogrešno	(0)	pogrešno	(0)

[†] Rijeè `bool` dolazi od prezimena engleskog matematièara Georgea Boolea (1815–1864), utemeljitelja logièke algebre.

Tablica 4.8. Stanja za logièki *ili*

a		*b*		*a* .ili. *b*	
toèno	(1)	toèno	(1)	toèno	(1)
toèno	(1)	pogrešno	(0)	toèno	(1)
pogrešno	(0)	toèno	(1)	toèno	(1)
pogrešno	(0)	pogrešno	(0)	pogrešno	(0)

```
Logicki b = ISTINA;
Logicki c = DRUGAISTINA;

cout << "a = " << a << ", b = " << b
     << ", c = " << c << endl;

cout << "Suprotno od a = " << !a << endl;
cout << "Suprotno od b = " << !b << endl;
cout << "Suprotno od c = " << !c << endl;

cout << "a .i. b = " << (a && b) << endl;
cout << "a .ili. c = " << (a || c) << endl;
```

Unatoè tome da smo varijabli `c` pridružili vrijednost `DRUGAISTINA = 124`, njenom logièkom negacijom dobiva se 0, tj. logièka neistina. Operacija `a`-*logièki i*-`b` daje neistinu (broj 0), jer operand `a` ima vrijednost *pogrešno*, dok `a`-*logièki ili*-`c` daje istinu, tj. 1, jer operand `c` ima logièku vrijednost *toèno*.

Logički operatori i operacije s njima uglavnom se koriste u naredbama za grananje toka programa, pa ćemo ih tamo još detaljnije upoznati.

Budući da su logički podaci tipa `bool` dosta su kasno uvršteni u standard C++ jezika, stariji prevoditelji ne podržavaju taj tip podataka. Želite li na starijem prevoditelju prevesti program koji je pisan u novoj verziji jezika, tip `bool` možete simulirati tako da na početak programa dodate sljedeće pobrojenje:

```
enum bool {false, true};
```

4.4.9. Poredbeni operatori

Osim aritmetièkih operacija, jezik C++ omoguæava i usporedbe dva broja (vidi tablicu 4.9). Kao rezultat usporedbe dobiva se tip `bool`: ako je uvjet usporedbe zadovoljen, rezultat je `true`, a ako nije rezultat je `false`. Tako æe se izvoðenjem kôda

```
cout << (5 > 4) << endl;    // je li 5 veće od 4?
cout << (5 >= 4) << endl;   // je li 5 veće ili jednako 4?
cout << (5 < 4) << endl;    // je li 5 manje od 4?
cout << (5 <= 4) << endl;   // je li 5 manje ili jednako 4?
```

Tablica 4.9. Poredbeni operatori

`x < y`	manje od
`x <= y`	manje ili jednako
`x > y`	veæe od
`x >= y`	veæe ili jednako
`x == y`	jednako
`x != y`	razlièito

```
cout << (5 == 4) << endl;  // je li 5 jednako 4?
cout << (5 != 4) << endl;  // je li 5 različito od 4?
```

na zaslonu ispisati brojevi 1, 1, 0, 0, 0 i 1.

Poredbeni operatori (ponekad nazvani i *relacijski operatori* od engl. *relational operators*) se koriste pretežito u naredbama za grananje toka programa, gdje se, ovisno o tome je li neki uvjet zadovoljen, izvođenje programa nastavlja u različitim smjerovima. Stoga ćemo poredbene operatore podrobnije upoznati kod naredbi za grananje, kasnije u ovom poglavlju.

Uoèimo suštinsku razliku između jednostrukog znaka jednakosti (`=`) koji je simbol za pridruživanje, te dvostrukog znaka jednakosti (`==`) koji je operator za usporedbu!

Što æe se dogoditi ako umjesto operatora pridruživanja `=`, pogreškom u nekom izrazu napišemo operator usporedbe `==`? Na primjer:

```
int a = 3;
a == 5;
```

U drugoj naredbi æe se umjesto promjene vrijednosti varijable `a`, ona usporediti s brojem 5. Rezultat te usporedbe je `false`, odnosno 0, ali to ionako nema nikakvog znaèaja, jer se rezultat usporedbe u ovom sluèaju ne pridružuje niti jednoj varijabli – naredba nema nikakvog efekta. A što je najgore, prevoditelj neæe prijaviti pogrešku, veæ možda samo upozorenje! Kod složenijeg izraza poput

```
a = 1.23 * (b == c + 5);
```

to upozorenje æe izostati, jer ovdje poredbeni operator može imati smisla. Za poèetnike jedno od nezgodnih svojstava jezika C++ jest da trpi i besmislene izraze, poput:

```
i + 5;
```

ili

```
i == 5 == 6;
```

Bolji prevoditelji æe u gornjim primjerima prilikom prevoðenja dojaviti upozorenje o tome da kôd nema efekta.

4.4.10. Znakovi

Znakovne konstante tipa `char` pišu se uglavnom kao samo jedan znak unutar jednostrukih znakova navodnika:

```
char SlovoA = 'a';
cout << 'b' << endl;
```

Za znakove koji se ne mogu prikazati na zaslonu koriste se *posebne sekvence* (engl. *escape sequence*) koje poèinju lijevom kosom crtom (engl. *backslash*) (tablica 4.10). Na primjer, kôd

```
cout << '\n';        // znak za novi redak
```

uzrokovat æe pomak kurzora na poèetak sljedeæeg retka, tj. ekvivalentan je ispisu konstante `endl`.

Tablica 4.10. Posebni znakovi

`\n`	novi redak
`\t`	horizontalni tabulator
`\v`	vertikalni tabulator
`\b`	pomak za mjesto unazad (*backspace*)
`\r`	povrat na poèetak retka (*carriage return*)
`\f`	nova stranica (*form feed*)
`\a`	zvuèni signal (*alert*)
`\\`	kosa crta ulijevo (*backslash*)
`\?`	upitnik
`\'`	jednostruki navodnik
`\"`	dvostruki navodnik
`\0`	završetak znakovnog niza
`\ddd`	znak èiji je kôd zadan oktalno s 1, 2 ili 3 znamenke
`\xddd`	znak èiji je kôd zadan heksadekadski

Znakovne konstante najèešæe se koriste u *znakovnim nizovima* (engl. *strings*) za ispis tekstova, te æemo ih tamo detaljnije upoznati. Za sada samo spomenimo da se znakovni nizovi sastoje od nekoliko znakova unutar dvostrukih navodnika. Zanimljivo je da se `char` konstante i varijable mogu usporeðivati, poput brojeva:

```
cout << ('a' < 'b') << endl;
cout << ('a' < 'B') << endl;
cout << ('A' > 'a') << endl;
cout << ('\'' != '\"') << endl; // usporedba jednostrukog
                                // i dvostrukog navodnika
```

pri èemu se u biti uporeðuju njihovi brojèani ekvivalenti u nekom od standarda. Najrašireniji je ASCII niz (kratica od *American Standard Code for Information Interchange*) u kojem su svim ispisivim znakovima, brojevima i slovima engleskog alfabeta pridruženi brojevi od 32 do 127, dok specijalni znakovi imaju kôdove od 0 do 31 ukljuèivo. U prethodnom primjeru, prva naredba ispisuje 1, jer je ASCII kôd malog slova `a` (97) manji od kôda za malo slovo `b` (98). Druga naredba ispisuje 0, jer je ASCII kôd slova B jednak 66 (nije važno što je B po abecedi iza a!). Treæa naredba ispisuje 0, jer za veliko slovo `A` kôd iznosi 65. ASCII kôdovi za jednostruki navodnik i dvostruki navodnik su 39 odnosno 34, pa zakljuèite sami što æe èetvrta naredba ispisati.

Znakovi se mogu uspoređivati sa cijelim brojevima, pri čemu se oni pretvaraju u cjelobrojni kôdni ekvivalent. Naredbom

```
cout << (32 == ' ') << endl;    // usporedba broja 32 i
                                // praznine
```

ispisat æe se broj 1, jer je ASCII kôd praznine upravo 32.

Štoviše, na znakove se mogu primjenjivati i svi aritmetički operatori. Budući da se pritom znakovi cjelobrojnom promocijom pretvaraju u cijele brojeve, za njih vrijede ista pravila kao i za operacije sa cijelim brojevima. Ilustrirajmo to sljedećim primjerom:

```
char a = 'h';               // ASCII kôd 104
char b;

cout << (a + 1) << endl;    // ispisuje 105

b = a + 1;
cout << b << endl;          // ispisuje 'i'
```

Kod prvog ispisa znakovna varijabla `a` (koja ima vrijednost slova `h`) se, zbog operacije sa cijelim brojem 1, pretvara se u ASCII kôd za slovo `h`, tj. u broj 104, dodaje joj se 1, te ispisuje broj 105. Druga naredba za ispis daje slovo `i`, jer se broj 105 pridružuje znakovnoj varijabli `b`, te se 105 ispisuje kao ASCII znak, a ne kao cijeli broj.

Zadatak. *Razmislite i provjerite što će se ispisati izvođenjem sljedećeg kôda:*

```
char a = 'a';
char b = a;
int asciiKod = a;
cout << a << endl;
cout << b << endl;
```

```
cout << 'b' << endl;
cout << asciiKod << endl;
```

Za pohranjivanje znakova koji ne stanu u tip `char` može se koristiti tip `wchar_t`. Duljina tipa `wchar_t` nije definirana standardom; na prevoditelju koji smo koristili pri pisanju knjige ona je iznosila dva bajta, tako da smo u jedan znak tipa `wchar_t` mogli smjestiti dva znaka:

```
wchar_t podebljiZnak = L'ab';
```

Slovo `L` ispred pridružene eksplicitno odreðuje da je znak koji slijedi tipa `wchar_t`.

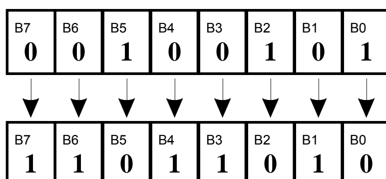
4.4.11. Bitovni operatori

Velika prednost jezika C++ je što omoguæava izravne operacije na pojedinim bitovima podataka. Na raspolaganju je šest operatora: bitovni komplement, bitovni *i*, bitovni *ili*, bitovni *iskljuèivi ili*, bitovni pomak udesno i bitovni pomak ulijevo (tablica 4.11). Valja napomenuti da su bitovni operatori definirani samo za cjelobrojne (`int`, `long int`) operande.

Tablica 4.11. Bitovni operatori

`~i`	komplement
`i & j`	binarni i
`i | j`	binarni ili
`i ^ j`	iskljuèivi ili
`i << n`	pomakni ulijevo
`i >> n`	pomakni udesno

Operator komplementiranja ~ (tilda) je unarni operator koji mijenja stanja pojedinih bitova, tj. sve bitove koji su jednaki nuli postavlja u 1, a sve bitove koji su 1 postavlja u 0. Tako će binarni broj 00100101_2 komplementiranjem prijeći u 11011010_2, kako je prikazano na slici 4.4.



~00100101 = 11011010

Slika 4.4. Bitovni komplement

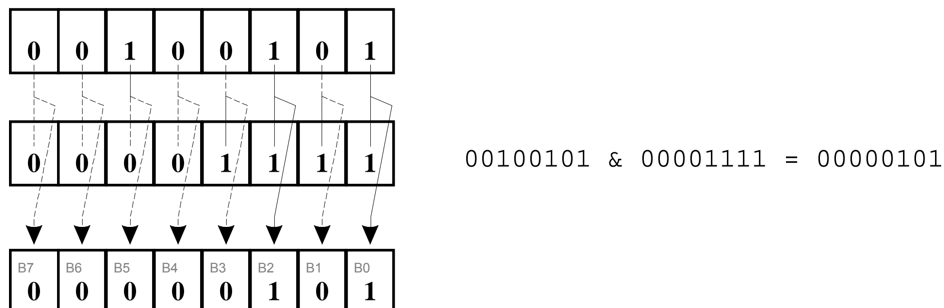
Zbog toga æe se izvoðenjem kôda

```
unsigned char a = 0x25;     // 00100101 u heksadec.prikazu
unsigned char b = ~a;       // pridruži bitovni komplement

cout << hex << (int)a << endl;
                            // ispisuje a u hex-formatu
cout << (int)b << endl;     // ispisuje b u hex-formatu
```

ispisati heksadekadski broj `25` i njegov bitovni komplement, heksadekadski broj `da` (tj. 11011010 u binarnom prikazu). U gornjem kôdu uoèavamo `hex` manipulator za izlazni tok, kojim smo osigurali da ispis svih brojeva koji slijede bude u heksadekadskom formatu. Manipulator `hex` definiran je u `iostream` biblioteci. Operator dodjele tipa `(int)` pri ispisu je neophodan, jer bi se bez njega umjesto heksadekadskih brojeva ispisali pripadajuæi ASCII znakovi.

Poneki čitatelj će se sigurno upitati zašto smo u gornjem primjeru varijable `a` i `b` deklarirali kao `unsigned char`. Varijabla tipa `char` ima duljinu samo jednog bajta, tj. zauzima 8 bitova, tako da će varijabla `a` nakon pridruživanja vrijednosti `0x25` u memoriji imati zaista oblik 00100101. Prilikom komplementiranja, ona se proširuje u `int`, tj. dodaje joj se još jedan bajt nula (ako `int` brojevi zauzimaju dva bajta), tako da ona postaje 00000000 00100101. Komplement tog broja je 11111111 11011010 ($FFDA_{16}$), ali kako se taj rezultat pridružuje `unsigned char` varijabli `b`, viši bajt (lijevih 8 bitova) se gubi, a preostaje samo niži bajt (DA_{16}). Da smo varijablu `b` deklarirali kao `int`, lijevih 8 bitova bi ostalo, te bismo na zaslonu dobili ispis heksadekadskog broja `ffda`. Pažljivi čitatelj će odmah primijetiti da je za varijablu `a` potpuno svejedno da li je deklarirana kao `char` ili kao `int` – ona se ionako prije komplementiranja pretvara u `int`. Nju smo deklarirali kao `unsigned char` samo radi dosljednosti s varijablom `b`.

Operator `&` (bitovni *i*) postavlja u 1 samo one bitove koji su kod oba operanda jednaki 1 (slika 4.5); matematičkom terminologijom rečeno, traži se *presjek* bitova dvaju brojeva.



Slika 4.5. Bitovni operator *i*

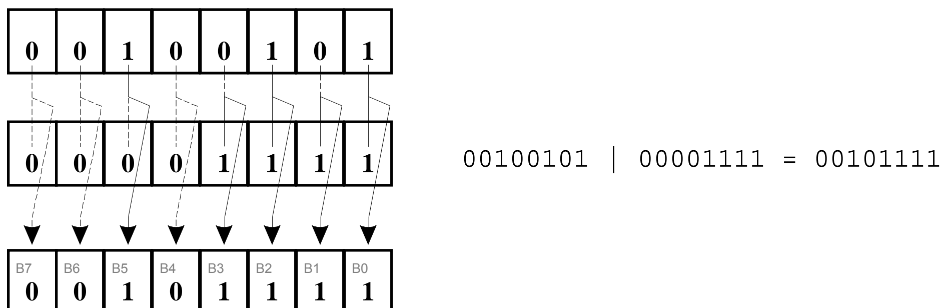
Bitovni *i* se najèešæe koristi kada se žele pojedini bitovi broja postaviti u 0 (*obrisati*) ili ako se žele izdvojiti samo neki bitovi broja. Drugi operand je pritom maska koja odreðuje koji æe se bitovi postaviti u nulu, odnosno koji æe se bitovi izluèiti. Za postavljanje odreðenih bitova u nulu svi bitovi maske na tim mjestima moraju biti jednaki 0. Ako u gornjem primjeru uzmemo da je maska donji operand, tj. 00001111_2, tada možemo reæi da smo kao rezultat dobili gornji broj u kojem su èetiri najznaèajnija (tj. lijeva) bita obrisana. Èetiri preostala bita su ostala nepromijenjena, jer maska na tim mjestima ima jedinice, iz èega odmah slijedi zakljuèak da u maski za izluèivanje svi

bitovi koje želimo izluèiti iz prvog broja moraju biti jednaki 1. Ako u gornjem primjeru uzmemo da je maska donji operand, kao rezultat smo dobili stanja èetiri najniža bita gornjeg operanda. Zbog svojstva komutativnosti možemo reæi i da je prvi broj maska pomoæu koje brišemo, odnosno vadimo odreðene bitove drugog broja. Izvorni kôd za gornju operaciju mogli bismo napisati kao:

```
int a = 0x0025;
int maska = 0x000f;

cout << hex << (a & maska) << endl;
```

Izvoðenjem se ispisuje broj 5.

Operator | (bitovni *ili*) postavlja u 1 sve one bitove koji su u bilo kojem od operanada jednaki 1 (slika 4.6); matematičkim rječnikom traži se *unija* bitova dvaju brojeva.



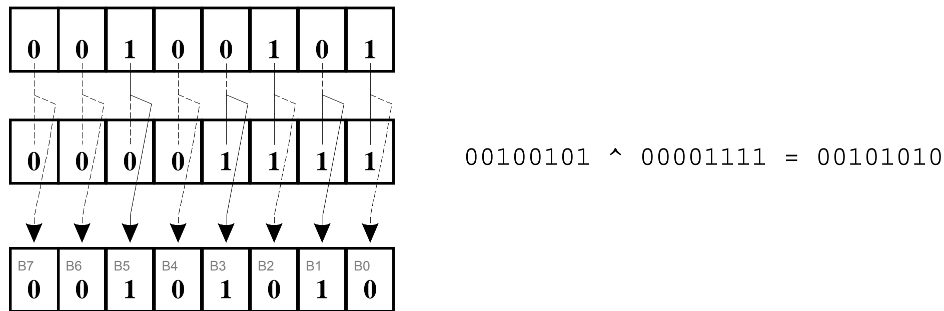
Slika 4.6. Bitovni operator *ili*

Bitovni *ili* se najèešæe koristi za postavljanje odreðenih bitova u 1. Drugi operand je pritom maska koja mora imati 1 na onim mjestima koja želimo u broju postaviti. U gornjem primjeru postavili smo sva èetiri najniža bita lijevog broja u 1. Stoga æe niz naredbi:

```
int a = 0x0025;
int maska = 0x000f;

cout << hex << (a | maska) << endl;
```

na zaslonu ispisati heksadekadski broj 2F.

Operator ^ (*isključivi ili*, ponekad nazvan i *ex-ili*, engl. *exclusive or*) postavlja u 1 samo one bitove koji su kod oba operanda međusobno različiti (slika 4.7). On se uglavnom koristi kada se žele promijeniti stanja pojedinih bitova. Odgovarajuća maska mora u tom slučaju imati 1 na mjestima bitova koje želimo promijeniti:



Slika 4.7. Bitovni operator *isključivo ili*

```
int a = 0x0025;
int maska = 0x000f;
cout << hex << (a ^ maska) << endl;     // ispisuje 0x2a
```

Zanimljivo je uoèiti da ponovna primjena maske vraæa broj u izvornu vrijednost:

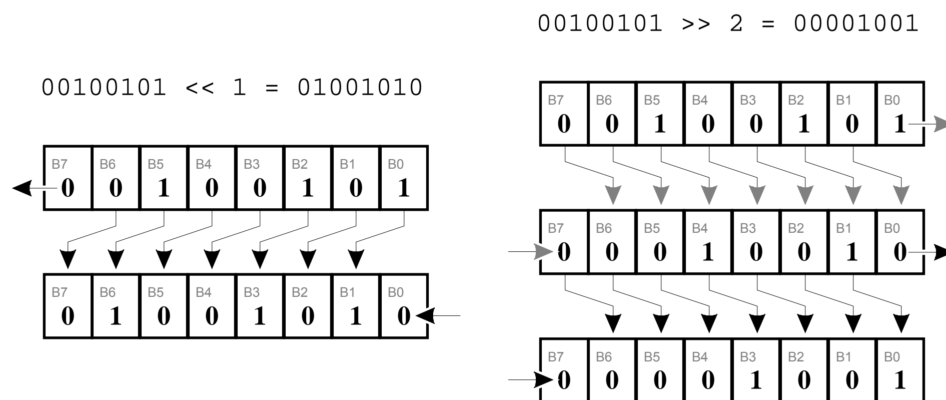
```
cout << hex << (a ^ maska ^ maska) << endl;
                                    // ispisuje 0x25
```

Ova èinjenica èesto se koristi za jednostavnu zaštitu podataka ili kôda od nepoželjnog èitanja, odnosno korištenja: primjenom *iskljuèivog ili* na podatke pomoæu tajne maske podaci se šifriraju. Ponovnom primjenom *iskljuèivog ili* s istom maskom podaci se vraæaju u izvorni oblik.

Operatori << (pomak ulijevo, engl. *shift left*) i >> (pomak udesno, engl. *shift right*) pomiču sve bitove lijevog operanda za broj mjesta određen desnim operandom, kako je prikazano na slici 4.8.

Pri pomaku ulijevo najznačajniji (tj. krajnji lijevi) bitovi se gube, dok najmanje značajni bitovi poprimaju vrijednost 0. Slično, pri pomaku udesno, gube se najmanje značajni bitovi, a najznačajniji bitovi poprimaju vrijednost 0. Uočimo da pomak bitova u cijelim brojevima za jedno mjesto ulijevo odgovara množenju broja s 2 – situacija je potpuno identična dodavanju nula iza dekadskog broja, samo što je kod dekadskog broja baza 10, a u binarnoj notaciji je baza 2. Slično, pomak bitova za jedno mjesto udesno odgovara dijeljenju cijelog broja s 2:

```
int a = 20;
cout << (a << 1) << endl;   // pomak za 1 bit ulijevo -
                            //   ispisuje 40
cout << (a >> 2) << endl;   // pomak za 2 bita udesno -
                            //   ispisuje 5
```



Slika 4.8. Bitovni pomaci ulijevo, odnosno udesno

Meðutim, pri korištenju pomaka valja biti krajnje oprezan, jer se kod cjelobrojnih konstanti s predznakom najviši bit koristi za predznak. Ako je broj negativan, tada se pomakom ulijevo bit predznaka æe se izgubiti, a na njegovo mjesto æe doæi najviša binarna znamenka. Kod pomaka udesno bit predznaka ulazi na mjesto najviše znamenke. Istina, neki prevoditelji vode raèuna o bitu predznaka, ali se zbog prenosivosti kôda ne treba previše oslanjati na to.

Mogućnost izravnog pristupa pojedinim bitovima često se koristi za stvaranje skupova logičkih varijabli pohranjenih u jednom cijelom broju, pri čemu pojedini bitovi tog broja predstavljaju zasebne logičke varijable. Takvim pristupom se štedi na memoriji, jer umjesto da svaka logička varijabla zauzima zasebne bajtove, u jednom je bajtu pohranjeno osam logičkih varijabli. Ilustrirajmo to primjerom u kojem se definiraju parametri za prijenos podataka preko serijskog priključka na računalu. Radi lakšeg praćenja, izvorni kôd ćemo raščlaniti na segmente, koje ćemo analizirati zasebno.

Za serijsku komunikaciju treba, osim brzine prijenosa izražene u baudima, definirati broj bitova po podatku (7 ili 8), broj stop-bitova (1 ili 2), te paritet (parni paritet, neparni paritet ili bez pariteta). Uzmimo da na raspolaganju imamo osam brzina prijenosa: 110, 150, 300, 600, 1200, 2400, 4800 i 9600 bauda. Svakoj toj brzini pridružit ćemo po jedan broj u nizu od 0 do 7:

```
enum {Baud110 = 0, Baud150, Baud300, Baud600,
      Baud1200, Baud2400, Baud4800, Baud9600};
```

Buduæi da imamo brojeve od 0 do 7, možemo ih smjestiti u tri najniža bita B0 - B2 (vidi sliku 4.9). Podatak o broju bitova po podatku pohranit æemo u B3; ako se prenosi 7 bitova po podatku bit B3 je jednak 0, a ako se prenosi 8 bitova po podatku onda je 1. Binarni broj koji ima treæi bit postavljen odgovara broju 08 u heksadekadskom prikazu ($0000\ 1000_2 = 08_{16}$):

```
enum {Bitova7 = 0x00, Bitova8 = 0x08};
```

Broj stop-bitova pohranit æemo u B4 i B5. Ako se traži jedan stop-bit, tada æe B4 biti 1, a za dva stop-bita B5 je jednak 1:

```
enum {StopB1 = 0x10, StopB2 = 0x20};
```

Konaèno, podatke o paritetu æemo pohraniti u dva najviša bita (B6 - B7):

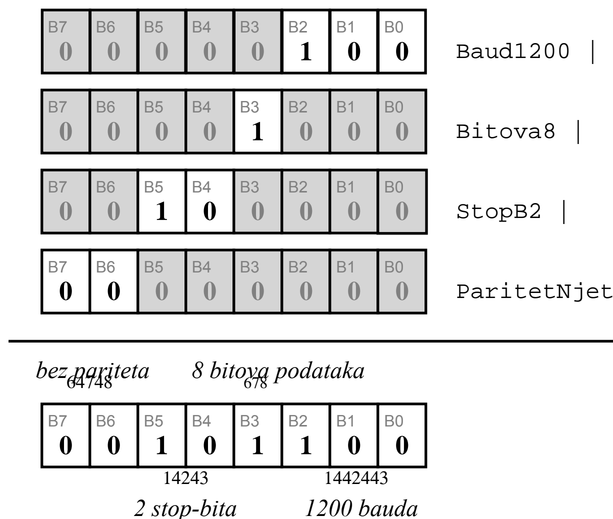
```
enum {ParitetNjet = 0x00, ParitetNeparni = 0x40,
      ParitetParni = 0x80};
```

Uoèimo da smo sva gornja pobrojenja mogli sažeti u jedno.

Želimo li sada definirati cjelobrojnu varijablu `SerCom` u kojoj će biti sadržani svi parametri, primijenit ćemo bitovni operator *ili*:

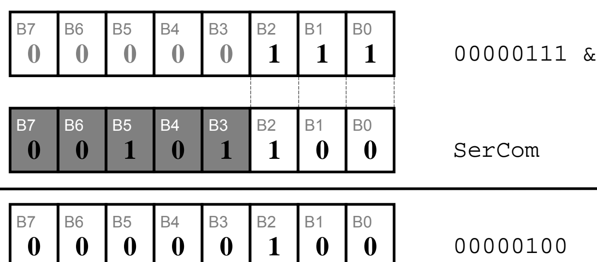
```
int SerCom = Baud1200 | Bitova8 | StopB2 | ParitetNjet;
```

kojim se određeni bitovi postavljaju u stanje 1. Rezultirajuæi raspored bitova prikazan je na slici 4.9.



Slika 4.9. Primjer rasporeda bitovnih parametara za serijsku komunikaciju

Kako iz nekog zadanog bajta s parametrima izluèiti relevantne informacije? Za to trebamo na cijeli bajt primijeniti masku kojom æemo odstraniti sve nezanimljive bitove. Na primjer, za brzinu prijenosa važni su nam samo bitovi B0 - B3, tako da æemo bitovnim *i* operatorom s brojem koji u binarnom prikazu ima 1 na tim mjestima (tj. s 07 heksadekadsko), izluèiti podatak o brzini (vidi sliku 4.10):



Slika 4.10. Primjena bitovne maske i operatora *i*

```
int Speed = SerCom & 0x07;          // dobiva se 4
```

Prenosi li se osam bitova podataka provjerit æemo sljedeæom bitovnom *ili* operacijom:

```
int JeLiOsamBita = SerCom & 0x08;   // dobiva se 8
```

buduæi da 8_{16} ima u binarnom prikazu 1 samo na mjestu B3. Ako je bit B3 postavljen, tj. ako ima vrijednost 1, varijabla `JeLiOsamBita` æe poprimiti vrijednost dekadsko (i heksadekadsko!) 8; u protivnom æe biti 0.

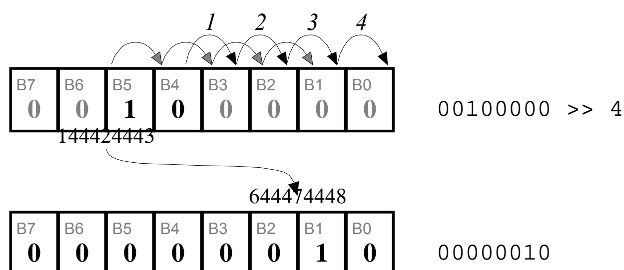
Analogno, za izlučivanje broja stop-bitova poslužit ćemo se maskom 30_{16} koja na mjestima B4 i B5 ima 1:

```
int BrojStopBita = SerCom & 0x30;   // dobiva se 32
```

U našem primjeru kao rezultat æemo dobiti 32. Ako rezultantu bitovnu strukturu pomaknemo udesno za 4 mjesta tako da B4 i B5 doðu na B0 odnosno B1 (slika 4.11):

```
BrojStopBita = BrojStopBita >> 4;
```

dobit æemo upravo broj stop-bitova, tj. broj 2.



Slika 4.11. Pomak udesno za četiri bita

Zadatak. *Napišite kôd u kojem će se* `int` *varijabla* `nekiBroj` *množiti s 4 pomakom bitova ulijevo. Prije pomaka pohranite bit predznaka u cjelobrojnu varijablu* `predznak`*, a nakon pomaka vratite predznak rezultatu! Napravite to isto i za dijeljenje s 2.*

4.4.12. Operatori pridruživanja (2 ½)

Osim veæ obraðenog operatora `=`, jezik C++ za aritmetièke i bitovne operatore podržava i operatore *obnavljajuæeg pridruživanja* (engl. *update assignment*) koji se sastoje se od znaka odgovarajuæeg aritmetièkog ili bitovnog operatora i znaka jednakosti. Operatori obnavljajuæeg pridruživanja omoguæavaju kraæi zapis naredbi. Na primjer, naredba

```
a += 5;
```

ekvivalentna je naredbi

```
a = a + 5;
```

Tablica 4.12. Operatori pridruživanja

=	+=	-=	*=	/=	%=	>>=	<<=	^=	&=	|=

U tablici 4.12 dani su svi operatori pridruživanja. Primjenu nekolicine operatora ilustrirat æemo sljedeæim kôdom

```
int n = 10;

n += 5;                // isto kao: n = n + 5
cout << n << endl;     // ispisuje: 15
n -= 20;               // isto kao: n = n - 20
cout << n << endl;     // ispisuje: -5
n *= -2;               // isto kao: n = n * (-2)
cout << n << endl;     // ispisuje: 10
n %= 3;                // isto kao: n = n % 3
cout << n << endl;     // ispisuje 1
```

Pri korištenju operatora obnavljajuæeg pridruživanja valja znati da operator pridruživanja ima niži prioritet od svih aritmetièkih i bitovnih operatora. Stoga, želimo li naredbu

```
a = a - b - c;
```

napisati kraæe, neæemo napisati

```
a -= b - c;          // to je zapravo a = a - (b - c)
```

veæ kao

```
a -= b + c;          // a = a - (b + c)
```

Operator -= ima niži prioritet od ostalih aritmetièkih operatora. Izraz s desne strane je zapravo izraz u zagradi ispred znaka oduzimanja.

4.4.13. Alternativne oznake operatora

U skladu sa standardima Meðunarodne organizacija za standardizaciju (ISO), specijalni znakovi poput [,], {, }, |, \, ~ i ^ nadomješteni su u pojedinim europskim jezicima nacionalnim znakovima kojih nema u engleskom alfabetu. Tako su u hrvatskoj inaèici ISO standarda (udomaæenoj pod nazivom CROSCII) ti znakovi nadomješteni slovima Š, Æ, š, æ, ð, Ð, è, odnosno È. Oèito je da æe na raèunalu koje podržava samo takav sustav znakova biti nemoguæe pregledno napisati èak i najjednostavniji C++ program. Na primjer, na zaslonu bi kôd nekog trivijalnog programa mogao izgledati ovako (onaj tko "dešifrira" kôd, zaslužuje bronèani Velered Bjarnea Stroustrupa):

```
int main() š
    int a = 5;
    char b = 'Ð0';
    int c = ča Č b;
    return 0;
ć
```

Buduæi da prevoditelj interpretira kôdove pojedinih znakova, a ne njihove grafièke prezentacije na zaslonu, program æe biti preveden korektno. Meðutim, za èovjeka je kôd neèitljiv. Osim toga, ispis programa neæete moæi poslati svom prijatelju u Njemaèkoj, koji ima svojih briga jer umjesto vitièastih zagrada ima znakove ü i ß.

Da bi se izbjegla ograničenja ovakve vrste, ANSI komitet je predložio alternativne oznake za operatore koji sadrže "problematične" znakove (tablica 4.13). Uz to, za istu

Tablica 4.13. Alternativne oznake operatora

osnovna	*alternativa*	*osnovna*	*alternativa*	*osnovna*	*alternativa*
{	<%	&&	and	~	compl
}	%>	||	or	!=	not_eq
[	<:	!	not	&=	and_eq
]	:>	&	bitand	|=	or_eq
#	%:	|	bitor	^=	xor_eq
##	%:%:	^	xor		

Tablica 4.14. Trigraf nizovi

trigraf	zamjena za	trigraf	zamjena za	trigraf	zamjena za
`??=`	#	`??(`	[	`??<`	{
`??/`	\	`??)`	]	`??>`	}
`??'`	^	`??!`	|	`??-`	~

namjenu je u jeziku C++ dozvoljena i upotreba nizova od tri znaka (engl. *trigraph*), naslijeđenih iz programskog jezika C (tablica 4.14). Srećom, operacijski sustavi koji za predstavljanje znakova koriste neki 8-bitni standard (npr. kodna stranica 852 pod DOS-om ili kodna stranica 1250 pod Windows-ima) omogućavaju istovremeno korištenje nacionalnih znakova i specijalnih znakova neophodnih za pisanje programa u jeziku C++, tako da većini korisnika tablice 4.13 i 4.14 neće nikada zatrebati. Ipak za ilustraciju pogledajmo kako bi izgledao neki program napisan pomoću alternativnih oznaka (čitatelju prepuštamo da "dešifrira" kôd):

```
%:include <iostream.h>

int main() <%
    bool prva = true;
    bool druga = false;
    bool treca = prva and druga;
    cout << treca << endl;
    cout << not treca << endl;
    cout << prva or druga << endl;
    return 1;
%>
```

4.4.14. Korisnièki definirani tipovi i operatori

Osim standardno ugraðenih tipova podataka koje smo u prethodnim odjeljcima upoznali, te operatora koji su za njih definirani, u programskom jeziku C++ na raspolaganju su izvedeni tipovi podataka kao što su polja, pokazivaèi i reference (njih æemo upoznati u 4. poglavlju). Meðutim, ono što programski jezik C++ èini naroèito moænim su *klase* koje omoguæavaju uvoðenje potpuno novih, *korisnièki definirani tipova podataka*. Tako programer više nije sputan osnovnim tipovima koji su ugraðeni u jezik, veæ ih može po volji dopunjavati i definirati operacije na novostvorenim tipovima. Detaljnije o klasama bit æe govora u narednim poglavljima.

4.4.15. Deklaracija `typedef`

Ključna rijeè `typedef` omoguæava uvoðenje novog imena za veæ postojeæi ugraðeni ili korisnièki definirani tip podataka. Na primjer, deklaracijom:

```
typedef float broj;
```

identifikator `broj` postaje sinonimom za tip `float`. Nakon gornje deklaracije "novopeèeni" identifikator tipa `broj` može se ravnopravno koristiti u deklaracijama objekata:

```
broj pi = 3.14159;
```

Buduæi da deklaracija `typedef` ne uvodi novi tip podataka, niti mijenja standardna pravila pretvorbe podataka, sljedeæa pridruživanja su dozvoljena i neæe prouzroèiti nikakve promjene u toènosti:

```
float a = pi;
typedef float pliva;
pliva = pi;
```

Osnovni razlog za primjenu deklaracije `typedef` jesu jednostavnije promjene kôda. Pretpostavimo da smo napisali program za neki numerièki proraèun tako da su nam svi brojèani podaci tipa `float`. Nakon nekog vremena utvrdili smo da nam `float` ne zadovoljava uvijek glede toènosti, te da ponekad neke brojeve moramo deklarirati kao `double`. U najgorem sluèaju to znaèi pretraživanje kôda i ruènu zamjenu odgovarajuæih deklaracija `float` u deklaracije `double` ili obrnuto (ako se predomislimo).

```
// prije promjene:                    // nakon promjene:
float a, b;                           double a, b;
float k, h;                           float k, h;
// ...                                // ...
float x, y;                           double x, y;
```

Kada je broj objekata mali to i nije teško, ali za veliki broj deklaracija te zamjene mogu biti naporne i podložne pogreškama. Posao æemo si bitno olakšati, ako u gornjem primjeru dodamo deklaraciju `typedef` za podatke èiji tip æemo povremeno mijenjati:

```
typedef float brojevi;
brojevi a, b;
float k, h;
// ...
brojevi x, y;
```

Želimo li sada promijeniti tipove, dovoljno je samo gornju deklaraciju `typedef` zamijeniti sljedeæom:

```
typedef double brojevi;
```

Sada æe svi podaci tipa `brojevi` biti prevedeni kao `double` podaci, bez potrebe daljnjih izmjena u izvornom kôdu.

Osim u ovakvim jednostavnim slučajevima, ključna riječ `typedef` se često koristi prilikom deklaracija pokazivača i referenci na objekte, pokazivača na funkcije i kod deklaracija polja. Naime, sintaksa kojom se ti tipovi opisuju može često biti vrlo složena. Zbog toga, ako često koristimo pokazivač na neki tip programski kôd može postati vrlo nečitljiv. U tom slučaju, jednostavnije je definirati pomoću ključne riječi `typedef` novi tip koji označava često korišteni tip "pokazivač na objekt". Takav tip postaje ravnopravan svim ostalim ugrađenim tipovima, te se može pojaviti na svim mjestima na kojima se može pojaviti ugrađeni tip: prilikom deklaracije objekata, specificiranja parametara funkcije, kao parametar `sizeof` operatoru i sl. Primjena `typedef` ključne riječi na takve tipove bit će objašnjena u kasnijim poglavljima.

4.5. Operator `sizeof`

Operator `sizeof` je unarni operator koji kao rezultat daje broj bajtova što ih operand zauzima u memoriji raèunala:

```
cout << "Duljina podataka tipa int je " << sizeof(int)
     << " bajtova" << endl;
```

Valja naglasiti da standard jezika C++ ne definira velièinu bajta, osim u smislu rezultata što ga daje `sizeof` operator; tako je `sizeof(char)` jednak 1. Naime, duljina bajta (broj bitova koji èine bajt) ovisi o arhitekturi raèunala. Mi æemo u knjizi uglavnom podrazumijevati da bajt sadrži 8 bitova, što je najèešæi sluèaj u praksi.

Operand `sizeof` operatora može biti identifikator tipa (npr. `int`, `float`, `char`) ili konkretni objekt koji je već deklariran:

```
float f;
int i;
cout << sizeof(f) << endl;      // duljina float
cout << sizeof(i) << endl;      // duljina int
```

Operator `sizeof` se može primijeniti i na izraze, koji se u tom sluèaju ne izraèunavaju veæ se odreðuje duljina njegova rezultata. Zbog toga æe sljedeæe naredbe ispisati duljine `float`, odnosno `int` rezultata:

```
float f;
int i;
cout << sizeof(f * i) << endl;          // duljina float
cout << sizeof((int)(i * f)) << endl;   // duljina int
```

Operator `sizeof` se može primijeniti i na pokazivaèe, reference, polja, korisnièki definirane klase, strukture, unije i objekte (s kojima æemo se upoznati u sljedeæim poglavljima). Ne može se primijeniti na funkcije (na primjer, da bi se odredilo njihovo zauzeæe memorije), ali se može primijeniti na pokazivaèe na funkcije. U svim sluèajevima on vraæa ukupnu duljinu tih objekata izraženu u bajtovima.

Rezultat operatora `sizeof` je tipa `size_t`, cjelobrojni tip bez predznaka koji ovisi o implementaciji prevoditelja, definiran u zaglavlju `stddef.h` (o zaglavljima će biti riječi u kasnijim poglavljima).

Operator `sizeof` se uglavnom koristi kod dinamičkog alociranja memorijskog prostora kada treba izračunati koliko memorije treba osigurati za neki objekt, o čemu će biti govora u kasnije u knjizi.

4.6. Operator razdvajanja

Operator razdvajanja `,` (zarez) koristi se za razdvajanje izraza u naredbama. Izrazi razdvojeni zarezom se izvode postepeno, s lijeva na desno. Tako æe nakon naredbe

```
i = 10, i + 5;
```

varijabli `i` biti pridružena vrijednost 15. Prilikom korištenja operatora razdvajanja u složenijim izrazima valja biti vrlo oprezan, jer on ima najniži prioritet (vidi sljedeæi odjeljak o hijerarhiji operatora). To se posebice odnosi na pozive funkcija u kojima se zarez koristi za razdvajanje argumenata. Ovaj operator se vrlo èesto koristi u sprezi s uvjetnim operatorom (poglavlje 5.3) te za razdvajanje više izraza u parametrima `for` petlje (poglavlje 5.5).

4.7. Hijerarhija i redoslijed izvođenja operatora

U matematici postoji utvrđena hijerarhija operacija prema kojoj neke operacije imaju prednost pred drugima. Podrazumijevani slijed operacija je slijeva nadesno, ali ako se dvije operacije razlièitog prioriteta naðu jedna do druge, prvo se izvodi operacija s višim prioritetom. Na primjer u matematièkom izrazu

$$a + b \cdot c / d$$

množenje broja b s brojem c ima prednost pred zbrajanjem s brojem a, tako da se ono izvodi prvo. Umnožak se zatim dijeli s d i tek se tada pribraja broj a.

I u programskom jeziku C++ definirana je hijerarhija operatora. Prvenstveni razlog tome je kompatibilnost s matematičkom hijerarhijom operacija, što omogućava pisanje računskih izraza na gotovo identičan način kao u matematici. Stoga gornji izraz u jeziku C++ možemo pisati kao

```
y = a + b * c / d;
```

Redoslijed izvoðenja operacija æe odgovarati matematièki oèekivanom. Operacije se izvode prema hijerarhiji operacija, poèevši s operatorima najvišeg prioriteta. Ako dva susjedna operatora imaju isti prioritet, tada se operacije izvode prema slijedu izvoðenja operatora. U tablici 4.15 na stranici 73 dani su svi operatori svrstani po hijerarhiji od najvišeg do najnižeg. Operatori s istim prioritetom smješteni su u zajednièke blokove.

Tablica 4.15. Hijerarhija i pridruživanje operatora

<table>
<tr><th>operator</th><th>znaèenje</th><th>pridruživanje</th><th>prioritet</th></tr>
<tr><td><code>::</code></td><td>globalno podruèje</td><td>s desna na lijevo</td><td>najviši</td></tr>
<tr><td><code>::</code></td><td>podruèje klase</td><td>s lijeva na desno</td><td></td></tr>
<tr><td><code>-> .</code></td><td>izbor èlana</td><td>s lijeva na desno</td><td></td></tr>
<tr><td><code>[]</code></td><td>indeksiranje</td><td>s lijeva na desno</td><td></td></tr>
<tr><td><code>()</code></td><td>poziv funkcije</td><td>s lijeva na desno</td><td></td></tr>
<tr><td><code>++</code></td><td>uveæaj nakon</td><td>s lijeva na desno</td><td></td></tr>
<tr><td><code>--</code></td><td>umanji nakon</td><td>s lijeva na desno</td><td></td></tr>
<tr><td><code>sizeof</code></td><td>velièina objekta</td><td>s desna na lijevo</td><td></td></tr>
<tr><td><code>++</code></td><td>uveæaj prije</td><td>s desna na lijevo</td><td></td></tr>
<tr><td><code>--</code></td><td>umanji prije</td><td>s desna na lijevo</td><td></td></tr>
<tr><td><code>* & + - ! ~</code></td><td>unarni operatori</td><td>s desna na lijevo</td><td></td></tr>
<tr><td><code>new</code></td><td>stvori objekt</td><td>s desna na lijevo</td><td></td></tr>
<tr><td><code>delete</code></td><td>izbriši objekt</td><td>s desna na lijevo</td><td></td></tr>
<tr><td><code>typeid</code></td><td>tip objekta</td><td>s desna na lijevo</td><td></td></tr>
<tr><td><code>()</code></td><td>dodjela tipa</td><td>s desna na lijevo</td><td></td></tr>
<tr><td><code>->* .*</code></td><td>pokazivaèi na èlan</td><td>s lijeva na desno</td><td></td></tr>
<tr><td><code>* / %</code></td><td>množenja</td><td>s lijeva na desno</td><td></td></tr>
<tr><td><code>+ -</code></td><td>zbrajanja</td><td>s lijeva na desno</td><td></td></tr>
<tr><td><code><< >></code></td><td>bitovni pomaci</td><td>s lijeva na desno</td><td></td></tr>
<tr><td><code>< > <= >=</code></td><td>poredbeni operatori</td><td>s lijeva na desno</td><td></td></tr>
<tr><td><code>== !=</code></td><td>operatori jednakosti</td><td>s lijeva na desno</td><td></td></tr>
<tr><td><code>&</code></td><td>bitovni i</td><td>s lijeva na desno</td><td></td></tr>
<tr><td><code>^</code></td><td>bitovno iskljuèivo ili</td><td>s lijeva na desno</td><td></td></tr>
<tr><td><code>|</code></td><td>bitovni ili</td><td>s lijeva na desno</td><td></td></tr>
<tr><td><code>&&</code></td><td>logièki i</td><td>s lijeva na desno</td><td></td></tr>
<tr><td><code>||</code></td><td>logièki ili</td><td>s lijeva na desno</td><td></td></tr>
<tr><td><code>?:</code></td><td>uvjetni izraz</td><td>s desna na lijevo</td><td></td></tr>
<tr><td><code>= *= /= += -= &=</code>
<code>^= |= %= >>= <<=</code></td><td>pridruživanja</td><td>s desna na lijevo</td><td></td></tr>
<tr><td><code>,</code></td><td>razdvajanje</td><td>s lijeva na desno</td><td>najniži</td></tr>
</table>

Striktno definiranje hijerarhije i slijeda izvođenja je neophodno i zato jer se neki znakovi koriste za više namjena. Tipičan primjer je znak `-` (minus) koji se koristi kao binarni operator za oduzimanje, kao unarni operator za promjenu predznaka, te u operatoru za umanjivanje (`--`). Takva višeznačnost može biti uzrokom čestih pogrešaka.

Ilustrirajmo to sljedeæim primjerom. Neka su zadana dva broja *a* i *b*; želimo od broja *a* oduzeti broj *b* prethodno umanjen za 1. Neopreznom programeru može se dogoditi da umjesto

```
c = a - --b;
```

za to napiše naredbu

```
c = a---b;
```

Što æe stvarno ta naredba uraditi pouzdano æemo saznati ako ispišemo vrijednosti svih triju varijabli nakon naredbe:

```
int main() {
    int a = 2;
    int b = 5;
    int c;

    c = a---b;
    cout << "a = " << a << ", b = " << b
         << ", c = " << c << endl;
    return 1;
}
```

Izvoðenjem ovog programa na zaslonu æe se ispisati

```
a = 1, b = 5, c = -3
```

što znaèi da je prevoditelj naredbu interpretirao kao

```
c = a-- - b;
```

tj. uzeo je vrijednost varijable `a`, od nje oduzeo broj `b` te rezultat pridružio varijabli `c`, a varijablu `a` je umanjio (ali tek nakon što je upotrijebio njenu vrijednost).

Kao i u matematici, okruglim zagradama se može zaobići ugrađena hijerarhija operatora, budući da one imaju viši prioritet od svih operatora. Tako će se u kôdu

```
d = a * (b + c);
```

prvo zbrojiti varijable `b` i `c`, a tek potom æe se njihov zbroj množiti s varijablom `a`. Èesto je zgodno zagrade koristiti i radi èitljivosti kôda kada one nisu neophodne. Na primjer, u gore razmatranom primjeru mogli smo za svaki sluèaj pisati

```
c = a - (--b);
```

Dobra je navada stavljati zagrade i praznine svugdje gdje postoji dvojba o hijerarhiji operatora i njihovom pridruživanju. Time kôd postaje pregledniji i razumljiviji. Pritom valja paziti da broj lijevih zagrada u naredbi mora biti jednak broju desnih zagrada – u protivnom æe prevoditelj javiti pogrešku.

5. Naredbe za kontrolu toka programa

'Tko kontrolira prošlost,' glasio je slogan
Stranke, 'kontrolira i budućnost: tko kontrolira
sadašnjost kontrolira prošlost.'

George Orwell (1903–1950), "1984"

U veæini realnih problema tok programa nije pravocrtan i jedinstven pri svakom izvođenju. Redovito postoji potreba da se pojedini odsjeèci programa ponavljaju – programski odsjeèci koji se ponavljaju nazivaju se *petljama* (engl. *loops*). Ponavljanje može biti unaprijed određeni broj puta, primjerice želimo li izraèunati umnožak svih cijelih brojeva od 1 do 100. Međutim, broj ponavljanja može ovisiti i o rezultatu neke operacije. Kao primjer za to uzmimo uèitavanje podataka iz neke datoteke – datoteka se uèitava podatak po podataka, sve dok se ne uèita znak koji oznaèava kraj datoteke (*end-of-file*). Duljina datoteke pritom može varirati za pojedina izvođenja programa.

Gotovo uvijek se javlja potreba za grananjem toka, tako da se ovisno o postavljenom uvjetu u jednom slučaju izvodi jedan dio programa, a u drugom slučaju drugi dio. Na primjer, želimo izračunati realne korijene kvadratne jednadžbe. Prvo ćemo izračunati diskriminantu – ako je diskriminanta veća od nule, izračunat ćemo oba korijena, ako je jednaka nuli izračunat ćemo jedini korijen, a ako je negativna ispisat ćemo poruku da jednadžba nema realnih korijena. Grananja toka i ponavljanja dijelova kôda omogućavaju posebne naredbe za kontrolu toka.

5.1. Blokovi naredbi

Dijelovi programa koji se uvjetno izvode ili èije se izvođenje ponavlja grupiraju se u *blokove naredbi* – jednu ili više naredbi koje su omeđene parom otvorena-zatvorena vitièasta zagrada `{}`. Izvana se taj blok ponaša kao jedinstvena cjelina, kao da se radi samo o jednoj naredbi. S blokom naredbi susreli smo se veæ u prvom poglavlju, kada smo opisivali strukturu glavne funkcije `main()`. U prvim primjerima na stranicama 23 i 26 cijeli program sastojao se od po jednog bloka naredbi. Blokovi naredbi se redovito pišu uvuèeno. To uvlaèenje radi se iskljuèivo radi preglednosti, što je naroèito važno ako imamo blokove ugniježđene jedan unutar drugog.

Važno svojstvo blokova jest da su varijable deklarirane u bloku vidljive samo unutar njega. Zbog toga će prevođenje kôda

```
#include <iostream.h>

int main() {
    {                        // početak bloka naredbi
```

```
        int a = 1;          // lokalna varijabla u bloku
        cout << a << endl;
    }                       // kraj bloka naredbi
    return 0;
}
```

proæi uredno, ali ako naredbu za ispis lokalne varijable `a` prebacimo izvan bloka

```
#include <iostream.h>

int main() {
    {
        int a = 1;
    }
    cout << a << endl;  // greška, jer a više ne postoji!
    return 0;
}
```

dobit æemo poruku o pogreški da varijabla `a` koju pokušavamo ispisati ne postoji. Ovakvo ogranièenje *podruèja* lokalne varijable (engl. *scope* – domet, doseg) omoguæava da se unutar blokova deklariraju varijable s istim imenom kakvo je veæ upotrijebljeno izvan bloka. Lokalna varijabla æe unutar bloka zakloniti istoimenu varijablu prethodno deklariranu izvan bloka, u što se najbolje možemo uvjeriti sljedeæim primjerom

```
#include <iostream.h>

int main() {
    int a = 5;
    {
        int a = 1;
        cout << a << endl;      // ispisuje 1
    }
    cout << a << endl;          // ispisuje 5
    return 0;
}
```

Prva naredba za ispis dohvatit æe lokalnu varijablu `a = 1`, buduæi da ona ima prednost pred istoimenom varijablom `a = 5` koja je deklarirana ispred bloka. Po izlasku iz bloka, lokalna varijabla `a` se gubi te je opet dostupna samo varijabla `a = 5`. Naravno da bi ponovna deklaracija istoimene varijable bilo u vanjskom, bilo u unutarnjem bloku rezultirala pogreškom tijekom prevođenja. Podruèjem dosega varijable pozabavit æemo se detaljnije u kasnijim poglavljima.

Ako se blok u naredbama za kontrolu toka sastoji samo od jedne naredbe, tada se vitičaste zagrade mogu i izostaviti.

5.2. Grananje toka naredbom `if`

Naredba `if` omoguæava uvjetno grananje toka programa ovisno o tome da li je ili nije zadovoljen uvjet naveden iza kljuène rijeèi `if`. Najjednostavniji oblik naredbe za uvjetno grananje je:

```
if ( logički_izraz )
    // blok_naredbi
```

Ako je vrijednost izraza iza rijeèi `if` logièka istina (tj. bilo koji broj razlièit od nule), izvodi se blok naredbi koje slijede iza izraza. U protivnom se taj blok preskaèe i izvoðenje nastavlja od prve naredbe iza bloka. Na primjer:

```
if (a < 0) {
    cout << "Broj a je negativan!" << endl;
}
```

U sluèaju da blok sadrži samo jednu naredbu, vitièaste zagrade koje omeðuju blok mogu se i izostaviti, pa smo gornji primjer mogli pisati i kao:

```
if (a < 0)
    cout << "Broj a je negativan!" << endl;
```

ili

```
if (a < 0) cout << "Broj a je negativan!" << endl;
```

Zbog preglednosti kôda i nedoumica koje mogu nastati prepravkama, poèetniku preporuèujemo redovitu uporabu vitièastih zagrada i pisanje naredbi u novom retku.

U protivnom se može dogoditi da nakon dodavanja naredbe u blok programer zaboravi omeðiti blok zagradama i time dobije nepredviðene rezultate:

```
if (a < 0)
    cout << "Broj a je negativan!" << endl;
    cout << "Njegova apsolutna vrijednost je " << -a
         << endl;
```

Druga naredba ispod `if` uvjeta izvest æe se i za pozitivne brojeve, jer više nije u bloku, pa æe se za pozitivne brojeve ispisati pogrešna apsolutna vrijednost! Inaèe, u primjerima æemo izbjegavati pisanje vitièastih zagrada gdje god je to moguæe radi uštede na prostoru.

Želimo li da se ovisno u rezultatu izraza u `if` uvjetu izvode dva nezavisna programska odsječka, primijenit ćemo sljedeći oblik uvjetnog grananja:

```
if ( logički_izraz )
    // prvi_blok_naredbi
else
    // drugi_blok_naredbi
```

Kod ovog oblika, ako izraz u `if` uvjetu daje rezultat razlièit od nule, izvest æe se prvi blok naredbi. Po završetku bloka izvoðenje programa nastavlja se od prve naredbe iza drugog bloka. Ako izraz daje kao rezultat nulu, preskaèe se prvi blok, a izvodi samo drugi blok naredbi, nakon èega se nastavlja izvoðenje naredbi koje slijede. Evo jednostavnog primjera u kojem se raèunaju presjecišta pravca s koordinatnim osima. Pravac je zadan jednadžbom $a \cdot x + b \cdot y + c = 0$.

```
#include <iostream.h>

int main() {
    float a, b, c;                  // koeficijenti pravca
    cin >> a >> b >> c;             // učitaj koeficijente
    cout << "Koeficijenti: " << a << ","
         << b << "," << c << endl;  // ispiši ih

    cout << "Presjecište s apscisom: ";
    if (a != 0)
        cout << -c / a << ", ";
    else
        cout << "nema, ";       // pravac je horizontalan

    cout << "presjecište s ordinatom: ";
    if (b != 0)
        cout << -c / b << endl;
    else
        cout << "nema" << endl; // pravac je vertikalan

    return 0;
}
```

Blokovi `if` naredbi se mogu nadovezivati:

```
if ( logički_izraz1 )
    // prvi_blok_naredbi
else if ( logički_izraz2 )
    // drugi_blok_naredbi
else if ( logički_izraz3 )
    // treći_blok_naredbi
.
```

```
else
    // zadnji_blok_naredbi
```

Ako je `logièki_izraz1` toèan, izvest æe se prvi blok naredbi, a zatim se izvoðenje nastavlja od prve naredbe iza zadnjeg `else` bloka u nizu, tj. iza bloka `zadnji_blok_naredbi`. Ako `logièki_izraz1` nije toèan, izraèunava se `logièki_izraz2` i ovisno o njegovoj vrijednosti izvodi se `drugi_blok_naredbi`, ili se program nastavlja iza njega. Ilustrirajmo to primjerom u kojem tražimo realne korijene kvadratne jednadžbe:

```
#include <iostream.h>

int main() {
    float a, b, c;

    cout << "Unesi koeficijente kvadratne jednadžbe:"
         << endl;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    cout << "c = ";
    cin >> c;

    float diskr = b * b - 4. * a * c;   // diskriminanta

    cout << "Jednadžba ima ";
    if (diskr == 0)
        cout << "dvostruki realni korijen." << endl;
    else if (diskr > 0)
        cout << "dva realna korijena." << endl;
    else
        cout << "dva kompleksna korijena." << endl;

    return 0;
}
```

Blokovi `if` naredbi mogu se ugnježðivati jedan unutar drugoga. Ilustrirajmo to primjerom u kojem gornji kôd poopæujemo i na sluèajeve kada je koeficijent *a* kvadratne jednadžbe jednak nuli, tj. kada se kvadratna jednadžba svodi na linearnu jednadžbu:

```
#include <iostream.h>

int main() {
    float a, b, c;

    cout << "Unesi koeficijente kvadratne jednadzbe:"
         << endl;
```

```
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    cout << "c = ";
    cin >> c;
    if (a) {
        float diskr = b * b - 4. * a * c;
        cout << "Jednadžba ima ";
        if (diskr == 0)
            cout << "dvostruki realni korijen." << endl;
        else if (diskr > 0)
            cout << "dva realna korijena." << endl;
        else
            cout << "kompleksne korijene." << endl;
    }
    else
        cout << "Jednadžba je linearna." << endl;

    return 0;
}
```

Za logièki izraz u prvom `if` uvjetu postavili smo samo vrijednost varijable `a` – ako æe ona biti razlièita od nule, uvjet æe biti zadovoljen i izvest æe se naredbe u prvom `if` bloku. Taj blok sastoji se od niza `if-else` blokova identiènih onima iz prethodnog primjera. Ako poèetni uvjet nije zadovoljen, tj. ako varijabla `a` ima vrijednost 0, preskaèe se cijeli prvi blok i ispisuje poruka da je jednadžba linearna. Uoèimo u gornjem primjeru dodatno uvlaèenje ugniježðenih blokova.

Pri definiranju logičkog izraza u naredbama za kontrolu toka početnik treba biti oprezan. Na primjer, želimo li da se dio programa izvodi samo za određeni opseg vrijednosti varijable `b`, naredba

```
if (-10 < b < 0) //...
```

neæe raditi onako kako bi se prema ustaljenim matematièkim pravilima oèekivalo. Ovaj logièki izraz u biti se sastoji od dvije usporedbe: prvo se ispituje je li –10 manji od `b`, a potom se rezultat te usporedbe usporeðuje s 0, tj.

```
if ((-10 < b) < 0) //...
```

Kako rezultat prve usporedbe može biti samo `true` ili `false`, odnosno 1 ili 0, druga usporedba dat æe uvijek logièku neistinu. Da bi program poprimio željeni tok, usporedbe moramo razdvojiti i logièki izraz formulirati ovako: "ako je –10 manji od `b` *i* ako je `b` manji od 0":

```
if (-10 < b && b < 0) //...
```

Druga nezgoda koja se može dogoditi jest da se umjesto operatora za usporedbu `==`, u logièkom izrazu napiše operator pridruživanja `=`. Na primjer:

```
if (k = 0)  // pridruživanje, a ne usporedba!!!
    k++;
else
    k = 0;
```

Umjesto da se varijabla `k` usporeðuje s nulom, njoj se u logièkom izrazu pridjeljuje vrijednost 0. Rezultat logièkog izraza jednak je vrijednosti varijable `k` (0 odnosno `false`), tako da se prvi blok naredbi nikada ne izvodi. Bolji prevoditelj æe na takvom mjestu korisniku prilikom prevoðenja dojaviti upozorenje.

5.3. Uvjetni operator `? :`

Iako ne spada meðu naredbe za kontrolu toka programa, uvjetni operator po strukturi je slièan `if-else` bloku, tako da ga je zgodno upravo na ovom mjestu predstaviti. Sintaksa operatora uvjetnog pridruživanja je:

```
uvjet ? izraz1 : izraz2 ;
```

Ako izraz *`uvjet`* daje logièku istinu, izraèunava se *`izraz1`*, a u protivnom *`izraz2`*. U primjeru

```
x = (x < 0) ? -x : x;   // x = abs(x)
```

ako je `x` negativan, izraèunava se prvi izraz, te se varijabli `x` na lijevoj strani znaka jednakosti pridružuje njegova pozitivna vrijednost, tj. varijabla `x` mijenja svoj predznak. Naprotiv, ako je `x` pozitivan, tada se izraèunava drugi izraz i varijabli `x` na lijevoj strani pridružuje njegova nepromijenjena vrijednost.

Izraz u uvjetu mora kao rezultat vraćati aritmetički tip ili pokazivač. Alternativni izrazi desno od znaka upitnika moraju davati rezultat istog tipa ili se moraju dati svesti na isti tip preko ugrađenih pravila konverzije.

Uvjetni operator koristite samo za jednostavna ispitivanja kada naredba stane u jednu liniju. U protivnom kôd postaje nepregledan.

Koliko èitatelja odmah shvaæa da u sljedeæem primjeru zapravo raèunamo korijene kvadratne jednadžbe?

```
((diskr = b * b -4 * a * c) >= 0) ?
    (x1 = (-b + diskr) / 2 / a, x2 = (-b + diskr) / 2 / a)  :
    (cout << "Ne valja ti korijen!", x1 = x2 = 0);
```

5.4. Grananje toka naredbom `switch`

Kada izraz uvjeta daje više razlièitih rezultata, a za svaki od njih treba provesti razlièite odsjeèke programa, tada je umjesto `if` grananja èesto preglednije koristiti `switch` grananje. Kod tog grananja se prvo izraèunava neki izraz koji daje cjelobrojni rezultat. Ovisno o tom rezultatu, tok programa se preusmjerava na neku od grana unutar `switch` bloka naredbi. Opæenita sintaksa `switch` grananja izgleda ovako:

```
switch ( cjelobrojni_izraz ) {
  case konstantan_izraz1 :
    // prvi_blok_naredbi
  case konstantan_izraz2 :
    // drugi_blok_naredbi
    break;
  case konstantan_izraz3 :
  case konstantan_izraz4 :
    // treći_blok_naredbi
    break;
  default:
    // četvrti_blok_naredbi
}
```

Prvo se izraèunava *`cjelobrojni_izraz`*, koji mora davati cjelobrojni rezultat. Ako je rezultat tog izraza jednak nekom od konstantnih izraza u `case` uvjetima, tada se izvode sve naredbe koje slijede pripadajuæi `case` uvjet sve do prve `break` naredbe. Nailaskom na `break` naredbu, izvoðenje kôda u `switch` bloku se prekida i nastavlja se od prve naredbe iza `switch` bloka. Ako `izraz` daje rezultat koji nije naveden niti u jednom od `case` uvjeta, tada se izvodi blok naredbi iza kljuène rijeèi `default`. Razmotrimo tokove programa za sve moguæe sluèajeve u gornjem primjeru. Ako *`cjelobrojni_izraz`* kao rezultat daje:

- *`konstantan_izraz1`*, tada æe se prvo izvesti *`prvi_blok_naredbi`*, a zatim *`drugi_blok_naredbi`*. Nailaskom na naredbu `break` prekida se izvoðenje naredbi u `switch` bloku. Program iskaèe iz bloka i nastavlja od prve naredbe iza bloka.
- *`konstantan_izraz2`*, izvodi se *`drugi_blok_naredbi`*. Nailaženjem na naredbu `break` prekida se izvoðenje naredbi u `switch` bloku i program nastavlja od prve naredbe iza bloka.
- *`konstantan_izraz3`* ili *`konstantan_izraz4`* izvodi se *`treæi_blok_naredbi`*. Naredbom `break` prekida se izvoðenje naredbi u `switch` bloku i program nastavlja od prve naredbe iza bloka.
- Ako rezultat nije niti jedan od navedenih *`konstantnih_izraza`*, izvodi se *`èetvrti_blok_naredbi`* iza `default` naredbe.

Evo i konkretnog primjera `switch` grananja (algoritam je prepisan iz priruènika za jedan stari programirljivi kalkulator):

```
#include <iostream.h>

int main() {
    cout << "Upiši datum u formatu DD MM GGGG:";
    int dan, mjesec;
    long int godina;
    cin >> dan >> mjesec >> godina;

    long datum;
    if (mjesec < 3) {
        datum = 365 * godina + dan + 31 * (mjesec - 1)
                + (godina - 1) / 4
                - 3 * ((godina - 1) / 100 + 1) / 4;
    }
    else {
        // uočimo operator dodjele tipa (int):
        datum = 365 * godina + dan + 31 * (mjesec - 1)
                - (int)(0.4 * mjesec + 2.3) + godina / 4
                - 3 * (godina / 100 + 1) / 4;
    }

    cout << dan << "." << mjesec << "." << godina
         << ". pada u ";

    switch (datum % 7) {
      case 0:
        cout << "subotu." << endl;
        break;
      case 1:
        cout << "nedjelju." << endl;
        break;
      case 2:
        cout << "ponedjeljak." << endl;
        break;
      case 3:
        cout << "utorak." << endl;
        break;
      case 4:
        cout << "srijedu." << endl;
        break;
      case 5:
        cout << "četvrtak." << endl;
        break;
      default:
        cout << "petak." << endl;
    }
    return 0;
}
```

Algoritam se zasniva na cjelobrojnim dijeljenjima, tako da sve varijable treba deklarirati kao cjelobrojne. Štoviše, `godina` se mora definirati kao `long int`, jer se ona u raèunu

množi s 365. U protivnom bi vrlo vjerojatno došlo do brojèanog preljeva, osim ako bismo se ograniièili na datume iz života Kristovih suvremenika. Uoèimo u gornjem kôdu operator dodjele tipa `(int)`

```
/*...*/  (int)(0.4 * mjesec + 2.3)  /*...*/
```

kojim se rezultat množenja i zbrajanja brojeva s pomiènim zarezom pretvara u cijeli broj, tj. odbacuju decimalna mjesta. U `switch` naredbi se rezultat prethodnih raèuna normira na neki od sedam dana u tjednu pomoæu operatora `%` (*modulo*).

Blok `default` se smije izostaviti, ali ga je redovito zgodno imati da bi kroz njega program prošao za vrijednosti koje nisu obuhvaćene `case` blokovima. Ovo je naročito važno tijekom razvijanja programa, kada se u `default` blok može staviti naredba koja će ispisivati upozorenje da je *`cjelobrojni_izraz`* u `switch` poprimio neku nepredviđenu vrijednost.

5.5. Petlja `for`

Èesto u programima treba ponavljati dijelove kôda. Ako je broj ponavljanja poznat prije ulaska u petlju, najprikladnije je koristiti `for` petlju. To je najopæenitija vrsta petlje i ima sljedeæi oblik:

```
for ( početni_izraz ; uvjet_izvođenja ; izraz_prirasta )
    // blok_naredbi
```

Postupak izvoðenja `for`-bloka je sljedeæi:

1. Izraèunava se *`poèetni_izraz`*. Najèešæe je to pridruživanje poèetne vrijednosti brojaèu kojim æe se kontrolirati ponavljanje petlje.
2. Izraèunava se *`uvjet_izvođenja`*, izraz èiji rezultat mora biti tipa `bool`. Ako je rezultat jednak logièkoj neistini, preskaèe se *`blok_naredbi`* i program se nastavlja prvom naredbom iza bloka.
3. Ako je *`uvjet_izvođenja`* jednak logièkoj istini, izvodi se *`blok_naredbi`*.
4. Na kraju se izraèunava *`izraz_prirasta`* (npr. poveæavanje brojaèa petlje). Program se vraæa na poèetak petlje, te se ona ponavlja od toèke 2.

Programski odsjeèak se ponavlja sve dok *`uvjet_izvođenja`* na poèetku petlje daje logièku istinu; kada rezultat tog izraza postane logièka neistina, programska petlja se prekida.

Kao primjer za `for` petlju, napišimo program za računanje faktorijela (faktorijela od n je umnožak svih brojeva od 1 do n):

$$n! = 1 \cdot 2 \cdot \ldots \cdot (n-2) \cdot (n-1) \cdot n$$

Pogledajmo kôd:

```
#include <iostream.h>

int main() {
    int n;
    cout << "Upiši prirodni broj: ";    // manji od 13!
    cin >> n;

    long int fjel = 1;
    for (int i = 2; i <= n; i++)
        fjel *= i;

    cout << n << "! = " << fjel;
    return 0;
}
```

Prije ulaska u petlju trebamo definirati poèetnu vrijednost varijable `fjel` u koju æemo gomilati umnoške. Na ulasku u petlju deklariramo brojaè petlje, varijablu `i` tipa `int` te joj pridružujemo poèetnu vrijednost 2 (*`poèetni_izraz:`* `int i = 2`). Unutar same petlje množimo `fjel` s brojaèem petlje, a na kraju tijela petlje uveæavamo brojaè (*`izraz_prirasta:`* `i++`). Petlju ponavljamo sve dok je brojaè manji ili jednak unesenom broju (*`uvjet_izvođenja:`* `i <= n`). Pri testiranju programa pazite da uneseni broj ne smije biti veæi od 12, jer æe inaèe doæi do brojèanog preljeva varijable `fjel`.

Zanimljivo je uočiti što će se dogoditi ako za `n` unesemo brojeve manje od 2. Već pri prvom ulasku u petlju neće biti zadovoljen uvjet ponavljanja petlje te će odmah biti preskočeno tijelo petlje i ona se neće izvesti niti jednom! Ovo nam odgovara, jer je 1! = 1 i (po definiciji) 0! = 1. Naravno da smo petlju mogli napisati i na sljedeći način:

```
for (int i = n; i > 1; i--)
    fjel *= i;
```

Rezultat bi bio isti. Iskusni programer bi gornji program mogao napisati još sažetije (vidi poglavlje 5.11), no u ovom trenutku to nam nije bitno.

Može se dogoditi da je uvjet izvođenja uvijek zadovoljen, pa će se petlja izvesti neograničeni broj puta. Program će uletjeti u slijepu ulicu iz koje nema izlaska, osim pomoću tipki *Power* ili *Reset* na kućištu vašeg računala. Na primjer:

```
// ovaj primjer pokrećete na vlastitu odgovornost!
cout << "Beskonačna petlja";
for (int i = 5; i > 1; )
    cout << "a";
```

Varijabla `i` je uvijek veæa od 1, tako da ako se i usudite pokrenuti ovaj program, ekran æe vam vrlo brzo biti preplavljen slovom `a`. Iako naizgled banalne, ovakve pogreške mogu poèetniku zadati velike glavobolje.

Uočimo da smo u gornjem kôdu izostavili *izraz_prirasta*. Općenito, može se izostaviti bilo koji od tri izraza u `for` naredbi – jedino su oba znaka `;` obavezna.

Izostavi li se *uvjet_izvođenja*, podrazumijevana vrijednost æe biti `true` i petlja æe biti beskonaèna!

Štoviše, mogu se izostaviti i sva tri izraza:

```
for ( ; ; )      // opet beskonačna petlja!
```

ali èitatelju prepuštamo da sam zakljuèi koliko je to smisleno.

Koristan savjet (ali ne apsolutno pravilo) je izbjegavati mijenjanje vrijednosti kontrolne varijable unutar bloka naredbi `for` petlje. Sve njene promjene bolje je definirati iskljuèivo u izrazu prirasta.

U protivnom se lako može dogoditi da petlja postane beskonaèna, poput sljedeæeg primjera:

```
for (int i = 0; i < 5; i++)
    i--;    // opet beskonačna petlja!
```

Naredba unutar petlje potpuno potire izraz prirasta, te varijabla `i` alternira između –1 i 0.

Početni izraz i izraz prirasta mogu se sastojati i od više izraza odvojenih operatorom nabrajanja `,` (zarez) . To nam omogućava da program za računanje faktorijela napišemo i (nešto) kraće:

```
#include <iostream.h>

int main() {
    int n;
    cout << "Upiši prirodni broj: ";
    cin >> n;

    long fjel;
    int i;
    for (i = 2, fjel = 1; i <= n; fjel *= i, i++) ;

    cout << n << "! = " << fjel;
    return 0;
}
```

U poèetnom izrazu postavljaju se brojaè i varijabla `fjel` na svoje inicijalne vrijednosti. Naredba za množenje s brojaèem prebaèena je iz bloka naredbi u izraz prirasta, tako da

je od bloka ostala prazna naredba, tj. sam znak `;`. Njega ne smijemo izostaviti, jer bi inaèe prevoditelj prvu sljedeæu naredbu (a to je naredba za ispis rezultata) obuhvatio u petlju. Sada je i deklaracija brojaèa prebaèena ispred `for` naredbe, jer poèetni izraz ne trpi višestruke deklaracije. Da smo `for` naredbu napisali kao:

```
for (int i = 2, long fjel = 1; i <= n; fjel *= i, i++) ;
```

prevoditelj bi javio da je deklaracija varijable `i` okonèana nepravilno, jer bi iza zareza, umjesto imena varijable naišao na kljuènu rijeè `long`.

Ako `for` naredba sadrži deklaraciju varijable, tada se područje te varijable prostire samo do kraja petlje[†] . Na primjer:

```
#include <iostream.h>

int main() {
    int i = -1;
    for (int i = 1; i <= 10; i++)
        cout << i << endl;
    cout << i << endl;          // ispisuje -1
    return 0;
}
```

`for` petlje mogu biti ugniježðene jedna unutar druge. Ponašanje takvih petlji razmotrit æemo na sljedeæem programu:

```
#include <iostream.h>
#include <iomanip.h>

int main() {
    for (int redak = 1; redak <= 10; redak++) {
        for (int stupac = 1; stupac <= 10; stupac++)
            cout << setw(5) << redak * stupac;
        cout << endl;
    }
    return 0;
}
```

Nakon prevoðenja i pokretanja programa, na zaslonu æe se ispisati veæ pomalo zaboravljena, ali generacijama puèkoškolaca omražena tablica množenja do 10:

```
    1    2    3    4    5    6    7    8    9   10
    2    4    6    8   10   12   14   16   18   20
    3    6    9   12   15   18   21   24   27   30
    4    8   12   16   20   24   28   32   36   40
    5   10   15   20   25   30   35   40   45   50
```

[†] Tijekom razvoja standarda to pravilo se mijenjalo. Tako æe neki stariji prevoditelji ostaviti varijablu `i` "živom" i iza `for` petlje.

```
 6   12   18   24   30   36   42   48   54   60
 7   14   21   28   35   42   49   56   63   70
 8   16   24   32   40   48   56   64   72   80
 9   18   27   36   45   54   63   72   81   90
10   20   30   40   50   60   70   80   90  100
```

Kako se gornji program izvodi? Pri ulasku u vanjsku petlju, inicijalizira se brojaè redaka na vrijednost 1 te se s njom ulazi u unutarnju petlju. U unutarnjoj petlji se brojaè stupaca mijenja od 1 do 10 i za svaku pojedinu vrijednost izraèunava se njegov umnožak s brojaèem redaka (potonji je cijelo to vrijeme `redak = 1`). Po završenoj unutarnjoj petlji ispisuje se znak za novi redak, èime završava blok naredbi vanjske petlje. Slijedi prirast brojaèa redaka (`redak = 2`) i povrat na poèetak vanjske petlje. Unutarnja petlja se ponavlja od `stupac = 1` do `stupac = 10` itd. Kao što vidimo, za svaku vrijednost brojaèa vanjske petlje izvodi se cjelokupna unutarnja petlja.

Da bismo dobili ispis brojeva u pravilnim stupcima, u gornjem primjeru smo rabili operator za rukovanje (*manipulator*) `setw()`. Argument tog manipulatora (tj. cijeli broj u zagradi) određuje koliki će se najmanji prostor predvidjeti za ispis podatka koji slijedi u izlaznom toku. Ako je podatak kraći od predviđenog prostora, preostala mjesta bit će popunjena prazninama. Manipulator `setw()` definiran je u datoteci zaglavlja `iomanip.h`.

5.6. Naredba `while`

Druga od tri petlje kojima jezik C++ raspolaže jest `while` petlja. Ona se koristi uglavnom za ponavljanje segmenta kôda kod kojeg broj ponavljanja nije unaprijed poznat. Sintaksa `while` bloka je

```
while ( uvjet_izvođenja )
    // blok_naredbi
```

`uvjet_izvoðenja` je izraz èiji je rezultat tipa `bool`. Tok izvoðenja `for`-bloka je sljedeæi:

1. Izraèunava se logièki izraz *`uvjet_izvoðenja`*.
2. Ako je rezultat jednak logièkoj neistini, preskaèe se *`blok_naredbi`* i program se nastavlja od prve naredbe iz bloka.
3. Ako je *`uvjet_izvoðenja`* jednak logièkoj istini izvodi se *`blok_naredbi`*. Potom se program vraæa na `while` naredbu i izvodi od toèke 1.

Konkretnu primjenu `while` bloka dat æemo programom kojim se ispisuje sadržaj datoteke s brojevima. U donjem kôdu je to datoteka `brojevi.dat`, ali uz promjene odgovarajuæeg imena, to može biti i neka druga datoteka.

```
#include <iostream.h>
#include <fstream.h>
```

```
int main() {
    ifstream ulazniTok("brojevi.dat");
    cout << "Sadržaj datoteke:" << endl << endl;
    float broj;
    while ((ulazniTok >> broj) != 0)
        cout << broj << endl;
    return 0;
}
```

Brojevi u datoteci `brojevi.dat` moraju biti upisani u tekstovnom obliku, razdvojeni prazninama, tabulatorima ili napisani u zasebnim recima (možemo ih upisati pomoæu najjednostavnijeg programa za upis teksta te ih pohraniti na disk).

Na početku programa se stvara objekt `ulTok` tipa (klase) `ifstream`. Iako će klase biti detaljnije objašnjene kasnije, za sada je dovoljno reći da je objekt `ulTok` sličan ulaznom toku `cin` kojeg smo do sada koristili za unos podataka s tipkovnice. Osnovna razlika je u tome što je `cin` povezan na tipkovnicu, dok se `ulTok` veže za datoteku čiji je naziv zadan u dvostrukim navodnicima u zagradama. Želimo li koristiti tokove u našem programu, potrebno je uključiti datoteku zaglavlja `fstream.h` pomoću pretprocesorske naredbe `#include`.

Usredotočimo se na blok `while` naredbe. Na početku `while`-petlje, u uvjetu izvođenja se naredbom

```
ulTok >> broj
```

uèitava podatak. Ako je uèitavanje bilo uspješno, `ulTok` æe biti razlièit od nule (neovisno o vrijednosti uèitanog broja), te se izvodi blok naredbi u `while`-petlji – ispisuje se broj na izlaznom toku `cout`. Nakon što se izvede naredba iz bloka, izvoðenje se vraæa na ispitivanje uvjeta petlje. Ako `ulTok` poprimi vrijednost 0, to znaèi da nema više brojeva u datoteci, petlja se prekida. Blok petlje se preskaèe, te se izvoðenje nastavlja prvom naredbom iza bloka.

Primijetimo zgodno svojstvo petlje `while`: ako je datoteka prazna, ulazni tok će odmah poprimiti vrijednost 0 te će uvjet odmah prilikom prvog testiranja biti neispunjen. Blok petlje se tada neće niti jednom izvesti, što u našem slučaju i trebamo.

Zadatak. *Dopunite gornji program tako da nakon sadržaja datoteke ispiše i broj iščitanih brojeva i njihovu srednju vrijednost.*

Zanimljivo je uočiti da nema suštinske razlike između `for`- i `while`-bloka naredbi – svaki `for`-blok se uz neznatne preinake može napisati kao `while`-blok i obrnuto. Da bismo se u to osvjedočili, napisat ćemo program za računanje faktorijela iz prethodnog poglavlja korištenjem `while` naredbe:

```
#include <iostream.h>

int main() {
    int n;
```

```
        cout << "Upiši prirodni broj: ";
        cin >> n;
        long int fjel = 1;
        int i = 2;
        while (i <= n) {
            fjel *= i;
            i++;
        }
        cout << n << "! = " << fjel;
        return 0;
    }
```

Trebalo je samo poèetni izraz (`int i = 2`) izluèiti ispred `while` naredbe, a izraz prirasta (`i++`) prebaciti iza bloka naredbi.

Zadatak. *Program za ispis datoteke napišite tako da umjesto `while`-bloka upotrijebite `for`-blok naredbi.*

Koji će se pristup koristiti (`for`-blok ili `while`-blok) prvenstveno ovisi o sklonostima programera. Ipak, zbog preglednosti i razumljivosti kôda `for`-blok je preporučljivo koristiti kada se broj ponavljanja petlje kontrolira cjelobrojnim brojačem. U protivnom, kada je uvjet ponavljanja određen nekim logičkim uvjetom, praktičnije je koristiti `while` naredbu.

5.7. Blok `do-while`

Zajednièko `for` i `while` naredbi jest ispitivanje uvjeta izvoðenja prije izvoðenja naredbi bloka. Zbog toga se može dogoditi da se blok naredbi ne izvede niti jednom. Meðutim, èesto je neophodno da se prvo izvede neka operacija te da se, ovisno o njenom ishodu, ta operacija eventualno ponavlja. Za ovakve sluèajeve svrsishodnija je `do-while` petlja:

```
do
    // blok_naredbi
while ( uvjet_ponavljanja );
```

Primjenu `do-while` bloka ilustrirat æemo programom-igricom u kojem treba pogoditi sluèajno generirani `trazeniBroj`. Nakon svakog našeg pokušaja ispisuje se samo poruka da li je broj veæi ili manji od traženog. Petlja se ponavlja sve dok je pokušaj (`mojBroj`) razlièit od traženog broja.

```
#include <iostream.h>
#include <stdlib.h>

int main() {
    int raspon = 100;
    randomize();// inicijalizira generator sluč. br.
    // generira slučajni broj između 1 i raspon
```

```
    int trazeniBroj = (float)rand() / RAND_MAX
                      * (raspon - 1) + 1;
    cout << "Treba pogoditi broj između 1 i "
         << raspon << endl;
    int mojBroj;
    int brojPokusa = 0;

    do {
        cout << ++brojPokusa << ". pokušaj: ";
        cin >> mojBroj;
        if (mojBroj > trazeniBroj)
            cout << "MANJE!" << endl;
        else if (mojBroj < trazeniBroj)
            cout << "VIŠE!" << endl;
    } while(mojBroj != trazeniBroj);

    cout << "BINGO!!!" << endl;
    return 0;
}
```

Za generiranje sluèajnih brojeva upotrijebljena je funkcija `rand()` iz zaglavlja `stdlib.h`. Ona kao rezultat vraæa sluèajni broj između 0 i `RAND_MAX`, pri èemu je `RAND_MAX` broj također definiran u `stdlib` biblioteci. Da bismo generirali broj u rasponu između 1 i `raspon`, broj koji vraæa funkcija `rand()` podijelili smo s `RAND_MAX` (èime smo normirali broj na interval od 0 do 1), pomnožili s `raspon - 1` i uveæali za 1. Prilikom dijeljenja primijenili smo operator dodjele tipa `(float)`, jer bi se u protivnom izgubila decimalna mjesta i rezultat bi bili samo brojevi 0 ili 1.

Uzastopni pozivi funkcije `rand()` uvijek generiraju isti slijed slučajnih brojeva, a prvi poziv te funkcije daje uvijek isti rezultat. Da bi se izbjegle katastrofalne posljedice takvog svojstva na neizvjesnost igre, prije poziva funkcije `rand()` valja pozvati funkciju `randomize()` koja inicijalizira klicu generatora slučajnog broja.

Zadatak. *Napišite program u kojem se računa približna vrijednost funkcije sinus pomoću reda potencija*

$$\sum_{i=0}^{\infty}(-1)^i \frac{x^{2i+1}}{(2i+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \mathrm{K}$$

Petlju za računanje članova reda treba ponavljati sve dok je apsolutna vrijednost zadnjeg izračunatog člana reda veća od nekog zadanog broja (npr. 10^{-7}*).*

5.8. Naredbe `break` i `continue`

Naredba `break` može se koristiti samo u petljama te u `switch` grananjima. Njenu funkciju u `switch` grananjima upoznali smo u poglavlju 0, dok se u petljama njome prekida izvođenje okolne `for`, `while` ili `do-while` petlje. Na primjer, želimo da se

izvođenje našeg program za ispis brojeva iz datoteke `brojevi.dat` (na str. 90) prekine kada se uèita 0. Tada æemo `while`-petlju modificirati na sljedeæi naèin:

```
// ...
while ((ulazniTok >> broj) != 0) {
    if (broj == 0)
        break;              // prekida petlju učitavanja
    cout << broj << endl;
}
// ...
```

Nailaskom na znak broj 0 program iskaèe iz `while`-petlje te se prekida daljnje èitanje datoteke i ispis njena sadržaja.

Naredba `continue` također uzrokuje skok programa na kraj petlje, ali se potom njeno ponavljanje nastavlja. Na primjer, želimo li da naš program za ispis sadržaja datoteke ispisuje samo one znakove koji se mogu prikazati na zaslonu, jedna moguća varijanta bila bi:

```
// ...
while ((znak = fgetc(ulazniTok)) != EOF) {
    if (znak < ' ' && znak != '\r')
        continue;       // preskače naredbe do kraja petlje
    cout << znak;
}
// ...
```

Nailaskom na znak èiji je kôd manji od kôda za prazninu i nije jednak znaku za novi redak, izvodi se naredba `continue` – preskaèu se sve naredbe do kraja petlje (u ovom sluèaju je to naredba za ispis znaka), ali se ponavljanje petlje dalje nastavlja kao da se ništa nije dogodilo.

Ako je više petlji ugniježđeno jedna unutar druge, naredba `break` ili naredba `continue` prouzročit će prekid ponavljanja, odnosno nastavak okolne petlje u kojoj se naredba nalazi.

Valja izbjegavati èesto korištenje `break` i `continue` naredbi u petljama, jer one narušavaju strukturiranost programa.

Koristite ih samo za "izvanredna" stanja, kada ne postoji drugi prikladan naèin da se izvođenje naredbi u petlji prekine. Naredba `continue` redovito se može izbjeæi `if`-blokom.

5.9. Ostale naredbe za skok

Naredba `goto` omoguæava bezuvjetni skok na neku drugu naredbu unutar iste funkcije. Opæi oblik je:

```
goto ime_oznake ;
```

ime_oznake je simbolièki naziv koji se mora nalaziti ispred naredbe na koju se želi prenijeti kontrola, odvojen znakom `:` (dvotoèka). Na primjer

```
if (a < 0)
    goto negativniBroj;
//...
negativniBroj: //naredba
```

Naredba na koju se želi skoèiti može se nalaziti bilo gdje (ispred ili iza naredbe `goto`) unutar iste funkcije. *ime_oznake* mora biti jedinstveno unutar funkcije, ali može biti jednako imenu nekog objekta ili funkcije. Ime oznake jest identifikator, pa vrijede pravila navedena u poglavlju 1.

U pravilno strukturiranom programu naredba `goto` uopæe nije potrebna, te ju velika veæina programera uopæe ne koristi.

Zadnju naredbu za kontrolu toka koju æemo ovdje spomenuti upoznali smo na samom poèetku knjige. To je naredba `return` kojom se prekida izvoðenje funkcija te æemo se njome pozabaviti u poglavlju posveæenom funkcijama.

5.10. O strukturiranju izvornog kôda

Uvlaèenje blokova naredbi i pravilan raspored vitièastih zagrada doprinose preglednosti i èitljivosti izvornog kôda. Programeri koji tek zapoèinju pisati u programskim jezicima C ili C++ èesto se naðu u nedoumici koji stil strukturiranja koristiti. Obièno preuzimaju stil knjige iz koje pretipkavaju svoje prve programe ili od kolege preko èijeg ramena kriomice stjeèu prve programerske vještine, ne sagledavajuæi nedostatke i prednosti tog ili nekog drugog pristupa. Nakon što im taj stil "uðe u krv", teško æe prijeæi na drugi bez obzira koliko je on bolji (u to su se uvjerili i sami autori knjige tijekom njena pisanja!).

Prvi problem jest raspored vitičastih zagrada koje označavaju početak i kraj blokova naredbi. Navedimo nekoliko najčešćih pristupa (redoslijed navođenja je slučajan):

1. vitièaste zagrade uvuèene i meðusobno poravnate:

```
for ( /*...*/ )
    {
    // blok naredbi
    // ...
    }
```

2. poèetna zagrada izvuèena, završna uvuèena:

```
for ( /*...*/ )
{   // blok naredbi
    // ...
    }
```

3. zagrade izvuèene, meðusobno poravnate:

```
for ( /*...*/ )
{   // blok naredbi
    // ...
}
```

4. poèetna zagrada na kraju naredbe za kontrolu, završna izvuèena:

```
for ( /*...*/ ) {
    // blok naredbi
    // ...
}
```

Pristupi 2. i 3. imaju još podvarijante u kojima se redak s poèetnom zagradom ostavlja prazan, bez naredbe. Meðutim, taj prazan redak ne doprinosi bitno preglednosti i nepotrebno zauzima prostor. Ista zamjerka može se uputiti prvom pristupu.

Već letimičnim pogledom na četiri gornja pristupa čitatelj će uočiti da izvučena završna zgrada u 3. odnosno 4. pristupu bolje ističe kraj bloka. U 3. pristupu zagrade u paru otvorena-zatvorena vitičasta zagrada međusobno su poravnate, tako da je svakoj zagradi lako uočiti njenog partnera, što može biti vrlo korisno prilikom ispravljanja programa. Međutim, u gotovo svim knjigama koristi se pristup 4 kod kojeg je početna vitičasta zagrada smještena na kraj naredbe za kontrolu toka. Budući da ona započinje blok te je vezana uz njega, ovaj pristup je logičan. U ostalim pristupima povezanost bloka naredbi s naredbom za kontrolu toka nije vizualno tako očita i može se steći dojam da blok naredbi predstavlja potpuno samostalnu cjelinu. Zbog navedenih razloga (*Kud svi Turci, tuda i mali Mi*), u knjizi koristimo 4. pristup. Uostalom, pronalaženja para, odnosno provjera uparenosti vitičastih zagrada u urednicima teksta (*editorima*) koji se koriste za pisanje i ispravljanje izvornog kôda ne predstavlja problem, jer većina njih ima za to ugrađene funkcije.

Druga nedoumica jest koliko duboko uvlačiti blokove. Za uvlačenje blokova najprikladnije je koristiti tabulatore. Obično editori imaju početno ugrađeni pomak tabulatora od po 8 znakova, međutim za iole složeniji program s više blokova ugniježđenih jedan unutar drugoga, to je previše. Najčešće se za izvorni kôd koristi uvlačenje po 4 ili samo po 2 znaka. Uvlačenje po 4 znaka je dovoljno duboko da bi se i početnik mogao lagano snalaziti u kôdu, pa smo ga zato i mi koristimo u knjizi. Iskusnijem korisniku dovoljno je uvlačenje i po samo 2 znaka, što ostavlja dovoljno prostora za dugačke naredbe. Naravno da kod definiranja tabulatora treba voditi računa o tipu znakova (*fontu*) koje se koristi u editoru. Za pravilnu strukturiranost kôda

neophodno je koristiti neproporcionalno pismo kod kojeg je širina svih znakova jednaka.

Treći “estetski” detalj vezan je uz praznine oko operatora. Početniku u svakom slučaju preporučujemo umetanje praznina oko binarnih operatora, ispred prefiks-operatora i iza postfiks operatora, te umetanje zagrada kada je god u nedoumici oko hijerarhije operatora. U protivnom osim estetskih, možete imati i sintaktičkih problema. Uostalom, neka sam čitatelj procijeni koji je kôd je čitljiviji:

```
a = b * c - d / (2.31 + e) + e / 8.21e-12 * 2.43;
```

ili

Koji æete pristup preuzeti ovisi iskljuèivo o vašoj odluci, ali ga onda svakako koristite dosljedno.

```
a=b*c-d/(2.31+e)+e/8.21e-12*2.43;
```

5.11. Kutak za buduæe C++ “gurue”

Jedna od odlika programskog jezika C++ jest moguænost sažetog pisanja naredbi. Podsjetimo se samo naredbe za inkrementiranje koja umjesto

```
i = i + 1;
```

omoguæava jednostavno napisati

```
i++;
```

Takoðer, moguænost višekratne uporabe operatora pridruživanja u istoj naredbi, dozvoljava da se primjerice umjesto dviju naredbi

```
a = b + 25.6;
c = e * a - 12;
```

napiše samo jedna

```
c = e * (a = b + 25.6) - 12;
```

s potpuno istim efektom. Ovakvo sažimanje kôda ne samo da zauzima manje prostora u editoru i izvornom kôdu, veæ èesto olakšava prevoditelju generiranje kraæeg i bržeg

izvedbenog kôda. Štoviše, mnoge naredbe jezika C++ (poput naredbe za inkrementiranje) vrlo su bliske strojnim instrukcijama mikroprocesora, pa se njihovim prevođenjem dobiva maksimalno efikasan kôd. Pri sažimanju kôda posebno valja paziti na hijerarhiju operatora (vidi tablicu 4.15). Èesto se zaboravlja da logièki operatori imaju niži prioritet od poredbenih operatora, a da operatori pridruživanja imaju najniži prioritet. Zbog toga nakon naredbe

```
c = 4 * (a = b - 5);
```

varijabla `a` neæe imati istu vrijednost kao nakon naredbe

```
c = 4 * ((a = b) - 5);
```

ili nakon naredbi

```
a = b;
c = 4 * (b - 5);
```

U prvom primjeru æe se prvo izraèunati `b - 5` te æe rezultat dodijeliti varijabli `a`, što je oèito razlièito od drugog, odnosno treæeg primjera gdje se prvo varijabli `a` dodijeli vrijednost od `b`, a zatim se provede oduzimanje.

Kod "C-gurua" su uobičajena sažimanja u kojima se umjesto eksplicitne usporedbe s nulom, kao na primjer

```
if (a != 0) {
    // ...
}
```

piše implicitna usporedba

```
if (a) {
    // ...
}
```

Iako æe oba kôda raditi potpuno jednako, suštinski gledano je prvi pristup ispravniji (i èitljiviji) – izraz u `if` ispitivanju po definiciji mora davati logièki rezultat (tipa `bool`). U drugom (sažetom) primjeru se, sukladno ugrađenim pravilima konverzije, aritmetièki tip pretvara u tip `bool` (tako da se brojevi razlièiti od nule pretvaraju u `true`, a nula se pretvara u `false`), pa je konaèni ishod isti.

Slična situacija je prilikom ispitivanja da li je neka varijabla jednaka nuli. U "dosljednom" pisanju ispitivanje bismo pisali kao:

```
if (a == 0) {
    // ...
}
```

dok bi nerijetki “gurui” to kraæe napisali kao

```
if (!a) {
    // ...
}
```

I u ovom sluèaju æe oba ispitivanja poluèiti isti izvedbeni kôd, ali æe neiskusnom programeru išèitavanje drugog kôda biti zasigurno teže. Štoviše, neki prevoditelji æe prilikom prevođenja ovako sažetih ispitivanja ispisati upozorenja.

Mogućnost sažimanja izvornog kôda (s eventualnim popratnim zamkama) ilustrirat ćemo programom u kojem tražimo zbroj svih cijelih brojeva od 1 do 100. Budući da se radi o trivijalnom računu, izvedbeni program će i na najsporijim suvremenim strojevima rezultat izbaciti za manje od sekunde. Stoga nećemo koristiti Gaussov algoritam za rješenje problema, već ćemo (zdravo-seljački) napraviti petlju koja će zbrojiti sve brojeve unutar zadanog intervala. Krenimo s prvom verzijom:

```
// ver. 1.0
#include <iostream.h>

int main() {
    int zbroj = 0;
    for(int i = 1; i <= 100; i++)
        zbroj = zbroj + i;
    cout << zbroj << endl;
    return 0;
}
```

Odmah uoèavamo da naredbu za zbrajanje unutar petlje možemo napisati kraæe:

```
// ver. 2.0
//...
for(int i = 1; i <= 100; i++) {
     zbroj += i;
//...
```

Štoviše, zbrajanje možemo ubaciti u uvjet za poveæavanje kontrolne varijable `for` petlje:

```
// ver. 3.0
//...
for (int i = 1; i <= 100; zbroj += i, i++) ;
//...
```

Blok naredbi u petlji je sada ostao prazan, ali ne smijemo izbaciti znak `;`. U dosadašnjim realizacijama mogli smo brojaè petlje `i` poveæavati i prefiks-operatorom:

```
// ver. 3.1
//...
for (int i = 1; i <= 100; zbroj += i, ++i) ;
//...
```

Efekt je potpuno isti – u izrazu prirasta `for`-naredbe izrazi odvojeni znakom `,` (zarezom) se izraèunavaju postupno, slijeva na desno. Naravno, da su izrazi napisani obrnutim redoslijedom

```
for (int i = 1; i <= 100; i++, zbroj += i)
```

konaèni zbroj bi bio pogrešan – prvo bi se uveæao brojaè, a zatim tako uveæan dodao zbroju – kao rezultat bismo dobili zbroj brojeva od 2 do 101. Meðutim, kada stopimo oba izraza za prirast

```
// ver. 4.0
//...
```

```
for (int i = 1; i <= 100; zbroj += i++) ;
//...
```

više nije svejedno koristi li se postfiks- ili prefiks-operator inkrementiranja, tj. da li se prvo dohvaæa vrijednost brojaèa, dodaje zbroju i zatim poveæava brojaè, ili se prvo poveæava brojaè, a tek potom dohvaæa njegova vrijednost:

```
for (int i = 1; i <= 100; zbroj += ++i)
```

Usporedimo li zadnju inaèicu (`4.0`) s prvom, skraæenje kôda je oèevidno. Ali je isto tako oèito da je kôd postao nerazumljiviji: prebacivanjem naredbe za pribrajanje brojaèa u `for`-naredbu, zakamuflirali smo osnovnu naredbu zbog koje je uopæe petlja napisana. Buduæi da veæina današnjih prevoditelja ima ugraðene postupke optimizacije izvedbenog kôda, pitanje je koliko æe i hoæe li uopæe ovakva sažimanja rezultirati bržim i efikasnijim programom. Stoga vam preporuèujemo:

Ne trošite previše energije na ovakva sažimanja, jer će najvjerojatnije rezultat biti neproporcionalan uloženom trudu, a izvorni kôd postati nečitljiv(iji).

6. Polja, pokazivaèi, reference

Let me take you down,
'cos I'm going to Strawberry Fields.
Nothing is real and nothing to get hungabout.
Strawberry Fields forever†.

John Lennon, Paul McCartney,
"Strawberry Fields forever" (1967)

U ovom poglavlju upoznat æemo se s izvedenim oblicima podataka. Prvo æe biti opisana polja, koja predstavljaju skupove istovjetnih podataka. Ona su u informatiku preuzeta iz matematike i omoguæavaju pristupanje veæem broju podataka preko istom simbolièkog imena i indeksa koji pokazuje redni broj elementa.

U drugom dijelu ovog poglavlja bit će analizirani pokazivači i reference. To su tipovi koji omogućavaju posredno pristupanje podacima pomoću posebnog mehanizma. Takvi tipovi nadovezuju se na adresiranje u strojnom jeziku, gdje oni čine važan mehanizam za pristup podacima u memoriji računala.

† Strawberry Fields je ime psihijatrijske ustanove u Velikoj Britaniji.

6.1. Polja podataka

Èesto se u programima koriste skupovi istovjetnih podataka. Za oznaèavanje i rukovanje takvim istovjetnim podacima bilo bi vrlo nespretno koristiti za svaki pojedini podatak zasebnu varijablu, primjerice `x1`, `x2`, `x3`, itd. Zamislimo samo da trebamo rezultate nekog mjerenja koji sadrže 100 podataka statistièki obraditi, pri èemu svaki podatak treba proæi kroz isti niz raèunskih operacija. Ako bismo koristili zasebna imena za svaki podatak, program bi se morao sastojati od 100 istovjetnih blokova naredbi koji bi se razlikovali samo u imenu varijable koja se u dotiènom bloku obraðuje. Neefikasnost ovakvog pristupa je više nego oèita. Daleko efikasnije je sve podatke smjestiti pod isto nazivlje, a pojedini rezultat mjerenja dohvaæati pomoæu brojèanog indeksa. Ovakvu obradu podataka omoguæavaju *polja podataka* (kraæe *polja*, engl. *arrays*).

Polje podataka je niz konačnog broja istovrsnih podataka – *članova polja*. Ti podaci mogu biti bilo kojeg tipa, ugrađenog (primjerice `float` ili `int`) ili korisnički definiranog. Pojedini članovi polja mogu se dohvaćati pomoću cjelobrojnih *indeksa* te mijenjati neovisno o ostalim članovima polja.

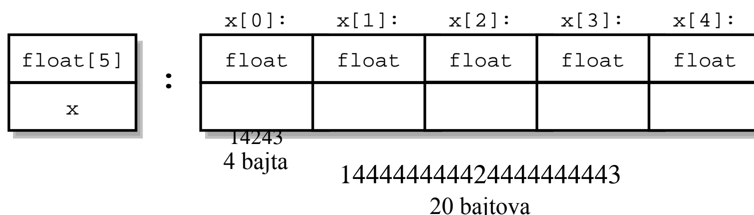
6.1.1. Jednodimenzionalna polja

Najjednostavniji oblik polja su jednodimenzionalna polja kod kojih se èlanovi dohvaæaju preko samo jednog indeksa. Èlanovi polja složeni su u linearnom slijedu, a indeks pojedinog èlana odgovara njegovoj udaljenosti od poèetnog èlana.

Želimo li deklarirati jednodimenzionalno polje `x` koje će sadržavati pet decimalnih brojeva tipa `float`, napisat ćemo

```
float x[5];
```

Prevoditelj æe ovom deklaracijom osigurati kontinuirani prostor u memoriji za pet podataka tipa `float` (slika 6.1). Ovakvom deklaracijom èlanovi polja nisu

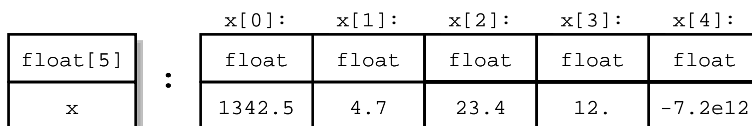


Slika 6.1. Deklaracija polja

inicijalizirani, tako da imaju sluèajne vrijednosti, ovisno o tome što se nalazilo u dijelu memorije koji je dodijeljen (*alociran*) polju. Èlanovi polja se mogu inicijalizirati prilikom deklaracije:

```
float x[] = {1342.5, 4.7, 23.4, 12., -7.2e12};
```

Ovom naredbom se deklarira jednodimenzionalno polje `x`, te se èlanovima tog polja pridjeljuju poèetne vrijednosti. Vrijednosti se navode unutar vitièastih zagrada i odvajaju zarezima. Iako duljina polja nije eksplicitno navedena, prevoditelj æe iz inicijalizacijske liste sam zakljuèiti da je polje duljine 5 i rezervirati odgovarajuæi memorijski prostor (slika 6.2).



Slika 6.2. Jednodimenzionalno polje s inicijaliziranim članovima

Navede li se ipak kod inicijalizacije duljina polja, treba paziti da ona bude veæa ili jednaka broju inicijalizirajuæih èlanova:

```
float x[5] = {1342.5, 4.7, 23.4, 12., -7.2e12};
```

U protivnom æe prevoditelj javiti pogrešku:

```
int a[2] = {10, 20, 30, 40};     // pogreška: previše
                                 //  inicijalizatora!
```

Broj inicijalizatora u listi smije biti manji od duljine polja:

```
int b[10] = {1, 2};
```

Tada èlanovi brojèanog polja kojima nedostaju inicijalizatori postaju jednaki nuli. Nije dozvoljeno pridružiti praznu inicijalizacijsku listu polju koje nema definiranu duljinu:

```
float z[] = {};     // pogreška: kolika je duljina polja?
```

Pri odabiru imena za polje treba voditi raèuna da se ime polja ne smije podudarati s imenom neke varijable u podruèju dosega polja, tj. u podruèju u kojemu je polje vidljivo.

Ako se deklarira polje s imenom koje je veæ iskorišteno za neku drugu varijablu, prevoditelj æe dojaviti pogrešku da je varijabla s dotiènim imenom deklarirana višekratno. Na primjer:

```
int a;
int a[] = {1, 9, 9, 6};     // pogreška: već postoji a!
```

Pojedini èlanovi polja se dalje u kôdu dohvaæaju pomoæu cjelobrojnog *indeksa* koji se navodi u uglatoj zagradi `[]` iza imena polja.

Prvi èlan u polju ima indeks 0, a zadnji èlan ima indeks za 1 manji od duljine polja.

U primjeru sa slike 6.2 to znaèi da su èlanovi polja: `x[0]`, `x[1]`, `x[2]`, `x[3]` i `x[4]`. Prema tome:

```
float a = x[0];      //pridružuje vrijednost prvog člana
x[2] = x[0] + x[1]; //treći član postaje jednak zbroju
                     // prvog i drugog člana
x[3]++;              //uveća četvrti član za 1
```

Kao što je iz ovih primjera oèito, svaki pojedini èlan polja može se dohvaæati i mijenjati neovisno o ostalim èlanovima.

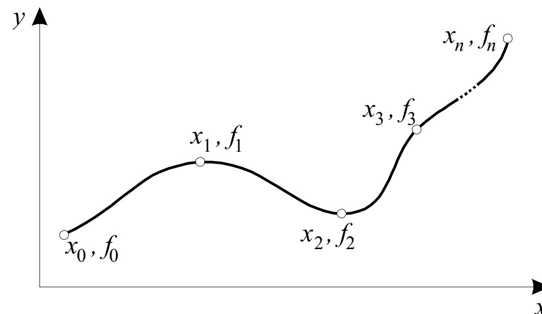
Primjenu polja ilustrirat ćemo programom koji iz datoteke "podaci.dat" učitava (x, y) parove točaka neke funkcije, a zatim za proizvoljni x unesen preko tipkovnice izračunava interpoliranu vrijednost funkcije $f(x)$ u toj točki. Koristit ćemo Lagrangeovu formulu za interpolaciju:

$$f(x) \approx \sum_{i=0}^{n} L_i(x) f_i = L_0(x) f_0 + L_1(x) f_1 + \mathrm{K}\; L_n(x) f_n \,,$$

pri èemu je $L_i(x)$ Lagrangeov koeficijent:

$$L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} = \frac{(x - x_0)(x - x_1)\mathrm{K}\;(x - x_{i-1})(x - x_{i+1})\mathrm{K}\;(x - x_n)}{(x_i - x_0)(x_i - x_1)\mathrm{K}\;(x_i - x_{i-1})(x_i - x_{i+1})\mathrm{K}\;(x_i - x_n)}$$

Tako dobivena funkcija prolazi toèno kroz zadane toèke (x_i, f_i) bez obzira kako su one rasporeðene (slika 6.3). Prvo æemo uèitati parove podataka u polja `x[]`, odnosno `y[]`:



Slika 6.3. Lagrangeova interpolacija

```
#include <iostream.h>
#include <fstream.h>

int main() {
    const int nmax = 100;
    float x[nmax], y[nmax];
    fstream ulazniTok("podaci.dat", ios::in);

    if (!ulazniTok) {
        cerr << "Ne mogu otvoriti traženu datoteku"
             << endl;
        return 1;
    }
    int i = -1;
    while (ulazniTok) {      // prekida se na kraju datoteke
        if (++i >= nmax) {
            cerr << "Previše podataka!" << endl;
            return 2;
        }
        ulazniTok >> x[i] >> y[i];
    }
    if (i < 0) {
        cerr << "Nema podataka!" << endl;
        return 3;
    }
    // nastavak slijedi...
```

Na samom poèetku glavne funkcije deklarira se simbolièka konstanta `nmax` kojom se dimenzioniraju polja `x[]` i `y[]` za pohranu uèitanih parova podataka.

Jedno od ogranièenja pri korištenju polja jest da duljina polja mora biti specificirana i poznata u trenutku prevoðenja kôda. Duljina jednom deklariranog polja se ne može mijenjati tijekom izvoðenja programa.

Zbog toga smo morali definirati vrijednost simbolièke konstante `nmax` prije deklaracija polja. Želimo li promijeniti dimenziju polja, promijenit æemo varijablu `nmax` i kôd ponovno prevesti. `nmax` obavezno treba biti deklarirana kao `const`. Ako to ne bi bio sluèaj, duljina polja bi bila zadana obiènom varijablom èija je vrijednost poznata tek prilikom izvoðenja programa, što je u suprotnosti s gornjim pravilom.

Za učitavanje podataka koristi se ulazni tok tipa `fstream` iz biblioteke `fstream`. Pri otvaranju se provjerava li je datoteka dostupna – ako nije ispisuje se pogreška, završava program, a operacijskom sustavu se kao rezultat vraća broj 1. Učitavanje podataka se obavlja u `while`-petlji. Petlja se ponavlja sve dok je `ulazniTok` različit od nule[†]; nailaskom na kraj datoteke, `ulazniTok` postaje jednak nuli i petlja se prekida.

---

† Valja primijetiti da je `ulazniTok` objekt, a ne cijeli broj te se on ne može direktno usporeðivati s nulom. No postoji operator konverzije koji to omoguæava i koji vraæa nulu ako je datoteka proèitana do kraja. Operatori konverzije æe biti kasnije objašnjeni.

Slijedi unos broja `x0` za koji se traži vrijednost funkcije `y0`, te računanje koeficijenata i tražene vrijednosti:

```
    // nastavak prethodnog kôda:
    int n = i;
    float x0, y0 = 0.;
    cout << "Upiši broj: ";
    cin >> x0;
    for (i = 0; i < n; i++) {
        float Li = 1.;
        for (int j = 0; j < n; j++) {
            if (i != j) {
                Li *= (x0 - x[j]);
                Li /= (x[i] - x[j]);
            }
        }
        y0 += (Li * y[i]);
    }
    cout << "f(" << x0 << ") = " << y0 << endl;
    return 0;
}
```

Petlja za raèunanje koeficijenata sastoji se od dvije `for`-petlje, jedne ugniježðene unutar druge. Vanjska petlja prolazi po indeksu *i* od prvog do zadnjeg uèitanog podatka raèunajuæi sumu u prvoj formuli, a unutarnja petlja za svaki *i* prolazi po indeksu *j* i raèuna produkt iz druge formule. Pri tome varijabla `Li` sadrži vrijednost *i*-tog Lagrangeovog koeficijenta.

Za provjeru, pretpostavimo da datoteka "podaci.dat" sadrži tablicu kvadratnih korijena nekoliko brojeva od 0 do 9:

```
0.00        0.0
0.49        0.7
1.00        1.0
1.44        1.2
2.25        1.5
2.89        1.7
4.00        2.0
6.25        2.5
9.00        3.0
```

Pokretanjem programa, utipkamo li broj 2, dobit æemo ispis približne vrijednosti kvadratnog korijena broja 2:

```
f(2) = 1.41701
```

Uoèimo da se prilikom uèitavanja podataka iz datoteke unutar `while`-petlje provjerava da li je broj uèitanih podataka nadmašio duljinu polja u koje se podaci pohranjuju. To je

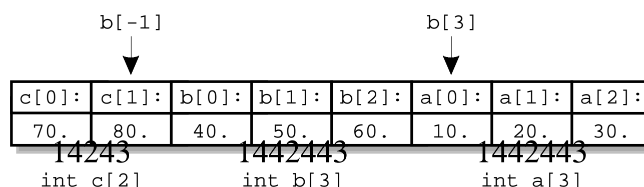
važna mjera zaštite, jer za razliku od mnogih programskih jezika, u jeziku C++ nema provjere granica polja prilikom pristupa elementima.

Navede li se preveliki ili negativni indeks, prevoditelj neæe javiti pogrešku i pristupit æe se memorijskoj adresi koja je izvan podruèja rezerviranog za polje!

Kod dohvaæanja èlana s nedozvoljenim indeksom, uèitana vrijednost bit æe opæenito neki sluèajni broj, što i ne mora biti pogibeljno. Meðutim, kod pridruživanja èlanu s indeksom izvan dozvoljenog opsega, vrijednost æe se pohraniti negdje u memoriju u podruèje predviðeno za neku drugu varijablu ili èak izvršni kôd. U prvom sluèaju dodjela æe uzrokovati promjenu vrijednosti varijable pohranjene na tom mjestu, što æe vjerojatno na kraju dati pogrešku u konaènom rezultatu. U drugom sluèaju, ako vrijednost "odbjeglog" èlana dospije u dio memorije gdje se nalazi izvršni kôd, posljedice su nesagledive i vjerojatno æe rezultirati blokiranjem rada programa ili cijelog raèunala. Da bismo se osvjedoèili u gornje tvrdnje, pogledajmo sljedeæi (bezopasan) primjer:

```
int main() {
    float a[] = {10, 20, 30};
    float b[] = {40, 50, 60};
    float c[] = {70, 80};

    cout << b[-1] << endl;
    cout << b[3] << endl;
    return 0;
}
```

Što æe se ispisati izvršavanjem ovog programa prvenstveno ovisi o tome kako korišteni prevoditelj raspodjeljuje memorijski prostor za polja `a[]`, `b[]` i `c[]`. Prevoditelj koji smo mi koristili složio je polja u memoriju jedno za drugim prema slici 6.4.



Slika 6.4. Dohvaćanje članova polja s indeksom izvan deklariranog opsega

Zbog toga su naredbama za ispis èlanova `b[-1]` i `b[3]` dohvaæeni èlanovi `c[1]`, odnosno `a[0]`.

Kod dohvaćanja članova polja indeks općenito smije biti cjelobrojna konstanta, cjelobrojna varijabla ili cjelobrojni izraz. Stoga su sljedeće naredbe dozvoljene:

```
float a[10];
int i = 2;
int j = 5;
float b = a[i + j];     // a[7]
b = a[i % j];           // a[2]
b = a[2 * i - j];       // a[-1]
```

Međutim, buduæi da indeks ne smije biti ništa osim cjelobrojnog tipa, naredbe poput:

```
b = a[2.3];             // pogreška
float c = 1.23;
b = a[c / 2];           // pogreška
```

uzrokovat æe pogrešku prilikom prevođenja.

6.1.2. Dvodimenzionalna polja

U dosadašnjim primjerima koristili smo jednodimenzionalna polja u kojima su èlanovi složeni u jednom kontinuiranom nizu i dohvaæaju se samo jednim indeksom. Èesto se pak javlja potreba za pohranjivanjem podataka u dvodimenzionalne ili višedimenzionalne strukture. Želimo li pohraniti podatke iz neke tablice s 3 retka i 5 stupaca (slika 6.5), najprikladnije ih je pohraniti u dvodimenzionalno polje

	1. stupac	2. stupac	3. stupac	4. stupac	5. stupac
1. redak	214	...	...	...	...
2. redak	...	...	...	...	...
3. redak	...	101	...	...	...

Slika 6.5. Primjer tablice s tri retka i pet stupaca

```
int Tablica[3][5];
```

Èlanove tog dvodimenzionalnog polja dohvaæamo preko dva indeksa: prvi je određen retkom, a drugi stupcem u kojem se podatak u tablici nalazi. Ne zaboravimo pritom da je poèetni indeks 0:

```
Tablica[0][0] = 214;        // 1.redak, 1.stupac
Tablica[2][1] = 101;        // 3.redak, 2.stupac
```

U suštini se ovo dvodimenzionalno polje može shvatiti kao tri jednaka jednodimenzionalna polja, od kojih je svako duljine 5 (vidi sliku 6.6).

Indeksi za pojedine dimenzije moraju biti odvojeni u zasebne parove uglatih zagrada.

U nekim jezicima se indeksi za sve dimenzije smještaju unutar zajednièkog para uglatih zagrada, odvojeni zarezom:

```
Tablica[2, 1];          // pogrešno
```

C++ prevoditelj na ovakvu naredbu neæe javiti sintaksnu pogrešku! Zarez koji odvaja bojeve 2 i 1 je operator razdvajanja koji æe prouzroèiti odbacivanje prvog broja; stoga æe gornja naredba biti ekvivalentna naredbi

```
Tablica[1];
```

Zapravo se dohvaæa adresa prvog èlana u drugom retku (o adresama æe biti rijeèi kasnije u ovom poglavlju).

Pravila koja vrijede za jednodimenzionalna polja vrijede i za višedimenzionalna polja. Članovi višedimenzionalnog polja mogu se također inicijalizirati prilikom deklaracije:

```
int Tablica[3][5] = { {11, 12, 13, 14, 15},
                      {21, 22, 23, 24, 25} };
```

Ovime se inicijaliziraju èlanovi prvih dvaju redaka. Unutrašnje vitièaste zagrade omeðuju èlanove u pojedinim recima. Buduæi da je inicijalizacijska lista kraæa od duljine polja (nedostaju èlanovi treæeg retka), èlanovi treæeg retka se inicijaliziraju na nulu, kako je prikazano slikom 6.6. Ako su liste unutar zagrada za pojedine retke prekratke, inicijalizirat æe se èlanovi prvih stupaca dotiènih redaka. Na primjer:

int[3][5]
Tablica

Tablica [0][0]:	Tablica [0][1]:	Tablica [0][2]:	Tablica [0][3]:	Tablica [0][4]:
int	int	int	int	int
11	12	13	14	15

Tablica [1][0]:	Tablica [1][1]:	Tablica [1][2]:	Tablica [1][3]:	Tablica [1][4]:
int	int	int	int	int
21	22	23	24	25

Tablica [2][0]:	Tablica [2][1]:	Tablica [2][2]:	Tablica [2][3]:	Tablica [2][4]:
int	int	int	int	int
0	0	0	0	0

Slika 6.6. Prikaz dvodimenzionalnog polja

```
int Tablica[3][5] = { {11}, {21, 22}, {31} };
```

Ovime æe se inicijalizirati samo pojedini èlanovi: `Tablica[0][0]`, `Tablica[1][0]`, `Tablica[1][1]` i `Tablica[2][0]`. Ako se vitièaste zagrade redaka izostave, tada se inicijalizacija provodi postupno od prvog retka, bez preskakanja u sljedeæi redak sve dok se ne "popune" svi èlanovi tog retka. Naredbom

```
int Tablica[3][5] = { 11, 12, 13, 14, 15, 21, 22 };
```

deklarirat æe se polje i inicijalizirati sve èlanove prvog retka (od `Tablica[0][0]` do `Tablica[0][4]`), te samo prva dva èlana drugog retka (`Tablica[1][0]` i `Tablica[1][1]`).

Primjenu dvodimenzionalnih polja prikazat ćemo u programu za rješavanje sustava linearnih jednadžbi Gaussovim postupkom. Gaussov postupak sastoji se iz dva dijela. U prvom se dijelu postupno iz jednadžbi uklanjaju nepoznanice tako da se matrica koeficijenata sustava svodi na gornju trokutastu matricu. U drugom dijelu se iz tako dobivenog trokutastog sustava povratnim supstitucijama izračunavaju nepoznanice. Pretpostavimo da je zadan sustav 4 jednadžbe s 4 nepoznanice:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 = a_{15}$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 = a_{25}$$
$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 = a_{35}$$
$$a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 = a_{45}$$

Dijeljenjem prve jednadžbe s a_{11} ona prelazi u

$$x_1 + b_{12}x_2 + b_{13}x_3 + b_{14}x_4 = b_{15}$$

Od preostale tri jednadžbe treba sada oduzeti prvu jednadžbu pomnoženu s koeficijentima a_{21}, a_{31}, odnosno a_{41}, da bismo iz njih uklonili nepoznanicu x_1. Time dobivamo sustav jednadžbi

$$\begin{aligned} x_1 + b_{12}x_2 + b_{13}x_3 + b_{14}x_4 &= b_{15} \\ a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + a_{24}^{(1)}x_4 &= a_{25}^{(1)} \\ a_{32}^{(1)}x_2 + a_{33}^{(1)}x_3 + a_{34}^{(1)}x_4 &= a_{35}^{(1)} \\ a_{42}^{(1)}x_2 + a_{43}^{(1)}x_3 + a_{44}^{(1)}x_4 &= a_{45}^{(1)} \end{aligned}$$

Podijelimo li sada drugu jednadžbu s $a_{22}^{(1)}$, ona prelazi u

$$x_2 + b_{23}x_3 + b_{24}x_4 = b_{25}$$

Od zadnje dvije jednadžbe treba sada oduzeti gornju jednadžbu pomnoženu koeficijentima $a_{32}^{(1)}$, odnosno $a_{42}^{(1)}$, da bismo iz njih uklonili nepoznanicu x_2. Ponavljanjem postupka do zadnje jednadžbe, na kraju dobivamo sustav jednadžbi

$$\begin{aligned} x_1 + b_{12}x_2 + b_{13}x_3 + b_{14}x_4 &= b_{15} \\ x_2 + b_{23}x_3 + b_{24}x_4 &= b_{25} \\ x_3 + b_{34}x_4 &= b_{35} \\ x_4 &= b_{35} \end{aligned}$$

Nakon što smo matricu sustava sveli na gornju trokutastu, povratnim supstitucijama dobivamo tražene nepoznanice: nepoznanicu x_4 dobivamo izravno iz zadnje jednadžbe, uvrštavanjem x_4 u treæu jednadžbu izraèunava se x_3 itd.

U programu ćemo prvo učitati broj jednadžbi, odnosno nepoznanica, a zatim učitavamo koeficijente jednadžbi:

```
#include <iostream.h>
#include <fstream.h>

int main() {
    const int nmax = 20;
    float a[nmax][nmax+1];
    fstream ulazniTok("koefic.dat", ios::in);

    if (!ulazniTok) {
        cerr << "Ne mogu otvoriti datoteku!" << endl;
        return 1;
    }
    int n;
    ulazniTok >> n;
    if (n >= nmax) {
        cerr << "Sustav jednadžbi prevelik!" << endl;
```

```
        return 2;
    }
    int r, s;
    for (r = 0; r < n; r++)              // po recima
        for (int s = 0; s <= n; s++)         // po stupcima
            ulazniTok >> a[r][s];
    // nastavak slijedi...
```

Koeficijenti se uèitavaju u dvodimenzionalno polje `a[][]`, pri èemu prvi indeks odgovara recima (tj. rednom broju jednadžbe), a drugi indeks polja odgovara stupcima matrice koeficijenata. Koeficijenti s desne strane znaka jednakosti (konstantni èlanovi) dodani su kao zadnji stupac matrice, tako da je broj stupaca za 1 veæi od broja redaka. Za sustav jednadžbi:

$$2x_1 - 7x_2 + 4x_3 = 9$$
$$x_1 + 9x_2 - 6x_3 = 1$$
$$-3x_1 + 8x_2 + 5x_3 = 6$$

datoteka s ulaznim podacima jest:

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

Nakon uèitavanja, podaci æe biti pohranjeni u polje `a[][]` na sljedeæi naèin:

```
a[0][0]=2.      a[0][1]=-7.     a[0][2]=4.      a[0][3]=9.
a[1][0]=1.      a[1][1]=9.      a[1][2]=-6.     a[1][3]=1.
a[2][0]=-3.     a[2][1]=8.      a[2][2]=5.      a[2][3]=6.
```

Ostali èlanovi polja ostaju nedefinirani, ali kako ih neæemo dohvaæati, to nema nikakvog utjecaja na raèun. Slijedi ranije opisani Gaussov postupak:

```
    // nastavak: svođenje na trokutastu matricu...
    for (r = 0; r < n; r++) {
        for (s = r + 1; s <= n; s++)
            a[r][s] /= a[r][r];
        for (int rr = r + 1; rr < n; rr++)
            for (int ss = r + 1; ss <= n; ss++)
                a[rr][ss] -= a[rr][r] * a[r][ss];
    }
    // ...te povratna supstitucija
    for (r = n - 1; r >= 0; r--)
        for (s = n - 1; s > r; s--)
            a[r][n] -= a[r][s] * a[s][n];
    // ispis rezultata
    for (r = 0; r < n; r++)
```

```
        cout << "x" << (r + 1) << " = "
             << a[r][n] << endl;

    return 0;
}
```

Za navedene ulazne podatke, program treba na zaslonu ispisati

```
x1 = 4.
x2 = 1.
x3 = 2.
```

Napomenimo da æe ovaj program "pasti" za sustav jednadžbi koji nema jednoznaèno rješenje, jer æe tijekom svoðenja na trokutastu matricu doæi do dijeljenja s nulom.

Zadatak: *Ubacite u kôd provjeru da li sustav ima jednoznačno rješenje; ako ga nema, neka se ispiše poruka o pogreški i prekine izvođenje programa.*

Ne ulazeći dublje u analizu algoritma i njegove implementacije, uočimo još nekoliko detalja vezanih uz programski kôd. Kod deklaracije polja `a[][]` možemo uočiti da je maksimalna veličina drugog indeksa definirana aritmetičkim izrazom (`nmax + 1`). To je dozvoljeno, uz uvjet da je rezultat tog izraza poznat u trenutku prevođenja kôda.

Uočimo također kako su u gornjem kôdu izostavljene vitičaste zagrade kod `for`-naredbi koje su ugniježđene jedna unutar druge:

```
    for (r = 0; r < n; r++)
        for (int s = 0; s <= n; s++)
            ulazniTok >> a[r][s];
```

Unatoè èinjenici da se unutar vanjske `for`-petlje nalaze dvije naredbe (unutarnja `for`-naredba te naredba za uèitavanje), prevoditelj blok za prvu `for`-naredbu uzima do kraja prve izvršne naredbe, tj. do prvog znaka `;` (toèka-zarez).

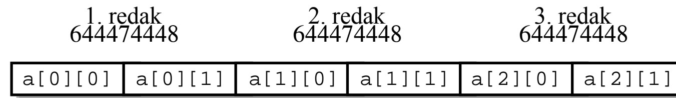
Ponekad se može ukazati potreba i za trodimenzionalnim poljima, pa će primjerice u nekom programu za analizu prostorne raspodjele temperature trebati deklarirati polje

```
float temperatura[x][y][z];
```

kod kojeg æe pojedini indeksi biti odreðeni (*x, y, z*) koordinatama toèaka za koje se raèuna temperatura. Iako broj dimenzija polja nije ogranièen, polja s više od dvije dimenzije se koriste vrlo rijetko. Pritom valja uoèiti da zauzeæe memorije raste s potencijom broja dimenzija: jednodimenzionalno polje `a[10]` broji deset èlanova, dvodimenzionalno polje `b[10][10]` broji $10 \times 10 = 100$ èlanova, a trodimenzionalno polje `c[10][10][10]` sadrži $10 \times 10 \times 10 = 1000$ èlanova.

Poneki čitatelj će se zapitati kako se dvodimenzionalna ili trodimenzionalna polja pohranjuju u memoriju koja se adresira linearno. Prevoditelj pojedine dimenzije polja

slaže uzastopno, jednu za drugom. Tako su članovi dvodimenzionalnog polja deklariranog kao `a[3][2]` raspoređeni u memoriju prema predlošku na slici 6.7.



Slika 6.7. Raspored dvodimenzionalnog polja u memoriji računala

Zbog toga se može dogoditi (slièno kao u primjeru na stranici 107 za tri jednodimenzionalna polja) da indeks izvan dozvoljenog opsega dohvati èlan iz drugog retka.

6.2. Pokazivaèi

Kao što samo ime kaže, *pokazivaèi* (engl. *pointers*) su objekti koji pokazuju na drugi objekt. Sam pokazivaè sadrži memorijsku adresu objekta na kojeg pokazuje. Iako na prvi pogled svrsishodnost pokazivaèa nije oèita, u jeziku C++ njihova primjena je veoma rasprostranjena, jer pružaju praktièki neogranièenu fleksibilnost u pisanju programa. Pokazivaèe smo posredno upoznali kod polja, buduæi da se èlanovi polja dohvaæaju upravo preko pokazivaèa.

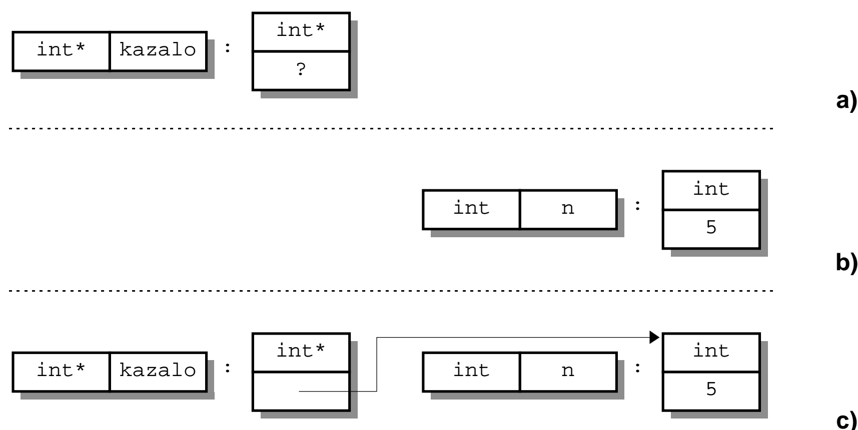
Pokazivač se može deklarirati tako da pokazuje na bilo koji tip podataka. U deklaraciji se navodi tip podatka na koji pokazivač pokazuje, a ispred imena se stavlja `*` (zvjezdica). Naredba

```
int *cajger;
```

deklarira pokazivaè `cajger` na objekt tipa `int`. Smisao pokazivaèa ilustrirat æemo sljedeæim kôdom:

```
int *kazalo;          // pokazivač na int
int n = 5;
kazalo = &n;          // usmjeri pokazivač na n
```

Prvom naredbom se deklarira varijabla `kazalo` kao pokazivaè na (neki) broj tipa `int` (slika 6.8a). Zatim se deklarira varijabla `n` te joj se pridružuje vrijednost 5 (slika 6.8b). Na kraju se varijabli `kazalo`, pomoæu operatora za dohvaæanje adrese `&` pridružuje vrijednost memorijske adrese na kojoj je pohranjena varijabla `n` (slika 6.8c). Sada vrijednost varijable `n` možemo dohvaæati izravno ili posredno, preko pokazivaèa:



Slika 6.8. Deklaracija pokazivača i pridruživanje vrijednosti

```
cout << "n = " << n << endl;              // ispisuje 5...
cout << "*kazalo = " << *kazalo << endl;  // ...i opet 5
```

Uoèimo kako smo pri dohvaæanju sadržaja varijable `n` preko pokazivaèa, morali ispred imena pokazivaèa umetnuti znak za pokazivaè; nas naime zanima sadržaj memorije na koju pokazuje pokazivaè, a ne sadržaj memorije u kojoj se nalazi pokazivaè.

Operator `*` naziva se *operatorom dereferenciranja* ili *indirekcije* (*dereferencing, indirection operator*) dok se operator `&` naziva *operatorom adrese* (*address-of operator*).

Izostavimo li znak `*` ispred imena pokazivaèa, dohvatit æemo sadržaj pokazivaèa, tj. memorijsku adresu na kojoj je pohranjena varijabla `n`:

```
cout << "kazalo = " << kazalo << endl;
                // ispisuje memorijsku adresu varijable n
```

Što æe se ispisati izvoðenjem ove naredbe nije jednoznaèno odreðeno, jer to ovisi o organizaciji memorije u pojedinom raèunalu te o memorijskoj lokaciji u koju æe se program smjestiti prilikom izvršavanja. Stoga je vrlo vjerojatno da æe višekratno izvoðenje programa na istom raèunalu dati ispis razlièitih memorijskih adresa, jer se program svaki puta smješta u drugi dio memorije. Ova nejednoznaènost ne treba zabrinjavati, jer se vrlo rijetko barata izravno s memorijskim adresama – veæinu tih operacija obavlja prevoditelj. U svakom sluèaju æe gornji ispis dati neki heksadecimalni broj, na primjer 0x2190.

Sve operacije koje su dozvoljene izravno na cjelobrojnoj varijabli, mogu se provesti i preko pokazivača. Tako gornjem kôdu možemo pridodati sljedeće naredbe:

```
*kazalo += 5;           // isto kao: n += 5
                        //  sada je n = 10
n = *kazalo - 2;        // isto kao: n = n - 2;
                        //  sada je n = 8
```

Kod operacija preko pokazivaèa naroèito treba paziti da se ne izostavi operator `*`, jer u protivnom može doæi do neugodnih efekata. Izostavimo li u prvoj gornjoj naredbi operator `*`

```
kazalo += 5;                    // preusmjerava pokazivač!
cout << "kazalo = " << kazalo << endl;
cout << "*kazalo = " << *kazalo << endl;
```

prevoditelj neæe imati uputu da želimo pristupiti vrijednosti na koju on pokazuje. Naprotiv, on æe baratati adresom pohranjenom u pokazivaèu. Tako æe prva naredba poveæati tu adresu za 5 memorijskih blokova, pri èemu je velièina tih memorijskih blokova definirana prostorom koji zauzima `int` varijabla (da je pokazivaè bio definiran kao `float *`, prirast bi bio jednak peterostrukoj duljini `float` varijable u memoriji). Prvi æe ispis dati adresu koja je za 5 duljina cjelobrojne varijable veæa od prvobitno ispisane adrese. Drugi ispis æe dati nepredvidljivi rezultat, jer ne znamo što se može naæi na toj adresi – to ovisi o ostalim varijablama, te o naèinu na koji prevoditelj organizira pohranjivanje varijabli.

Definiramo li novu cjelobrojnu varijablu, pokazivač možemo preusmjeriti na nju (slika 6.9) :

```
int m = 3;
kazalo = &m;
```

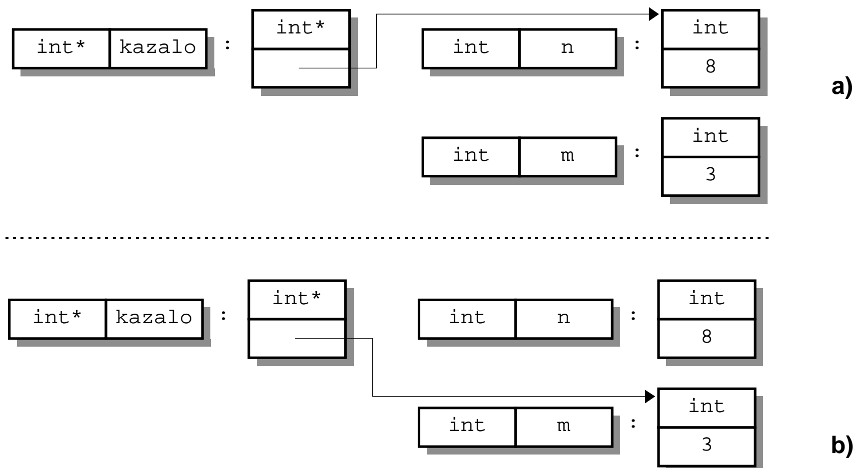
Ovime se oèito raskida veza izmeðu pokazivaèa `kazalo` i varijable `n`, a uspostavlja identièna veza izmeðu `kazalo` i `m`.

Valja uočiti da se pokazivač prilikom deklaracije može inicijalizirati, ali samo na objekt koji već postoji u memoriji. Tako smo početne deklaracije iz gornjeg primjera mogli pisati kao:

```
int n = 5;
int *kazalo = &n;
```

Postoji još jedan moguæi oblik inicijalizacije pokazivaèa prilikom deklaracije pomoæu operatora `new`, o èemu æemo govoriti kasnije u odjeljku 6.2.5.
Buduæi da brojèane konstante nemaju svoj prostor u memoriji, sljedeæa naredba nema smisla:

```
float *pi = &3.14;      // pogreška: pokazivač na što?
```



Slika 6.9. Preusmjeravanje pokazivača

Zadatak. *Odredite što će se ispisati izvršavanjem sljedećih naredbi:*

```
int i = 1;
int j = 10;
int *p = &j;
*p *= *p;
i = i + j;
p = &i;
cout << i << endl << j << endl << *p << endl;
```

Međusobne operacije s pokazivaèima na razlièite tipove podataka nisu preporuèljive i redovito su nedozvoljene. Razmotrimo to na primjeru:

```
int n = 5;
int *pokn = &n;
float x = 10.27;
float *pokx = &x;
*pokn = *pokx;       // n = 10
pokx = pokn;         // pogreška
pokx = &n;           // pogreška
```

Poèetnim deklaracijama i inicijalizacijama stvorene su cjelobrojna varijabla `n = 5`, te pokazivaè na nju i decimalna varijabla `x = 10.27` s pripadajuæim pokazivaèem. Naredbom

```
*pokn = *pokx;
```

se varijabli na koju je pokazuje `pokn` pridružuje vrijednost varijable na koju pokazuje `pokx`. Kako su te varijable razlièitih aritmetièkih tipova, prilikom pridruživanja primjenjuju se uobièajena pravila konverzije, pa æe nakon obavljene naredbe varijabla `n` poprimiti vrijednost 10.

Slijede dvije međusobno jednake naredbe kojima pokušavamo pokazivač na decimalnu varijablu preusmjeriti na adresu na kojoj se nalazi cjelobrojna varijabla. Iako adrese na različite tipove podataka u nekom računalu uvijek (manje-više) imaju isti oblik, načini na koje su te varijable pohranjene na dotičnim lokacijama su različiti, te bi pridruživanja ovakvog tipa davala nepredvidive rezultate. Zato prevoditelj ne dozvoljava ovakva pridruživanja i javlja pogrešku.

Zadatak: *Prije nego što pročitate objašnjenje koje slijedi, razmislite zašto će izvođenje sljedećeg kôda prouzročiti pogrešku:*

```
int *pokn;
float x = 10.27;
float *pokx = &x;

*pokn = *pokx;        // pogreška pri izvođenju
```

Kôd se razlikuje od prethodnoga po tome što nije inicijalizirana vrijednost pokazivaèa `pokn`. To znaèi da æe biti alociran prostor za taj pokazivaè, no njegova æe vrijednost biti sluèajna (ovisno o sadržaju memorije prije alokacije). Pokušaj pridruživanja u zadnjem retku vrlo æe vjerojatno prouzroèiti prekid programa ako `pokn` pokazuje na neku nedozvoljenu memorijsku lokaciju, jer æe se vrijednost 10.27 pokušati prepisati u zabranjeno memorijsko podruèje[†].

Izuzetak od gornjih pravila o međusobnom pridruživanju pokazivača čine pokazivači tipa `void *` (engl. *void* - prazan, ispražnjen). Oni pokazuju na neodređeni tip podataka, tj. na općenitu memorijsku lokaciju. Stoga pokazivač na `void` možemo preusmjeriti na objekt bilo kojeg tipa:

```
int n = 5;
int *pokn = &n;
float x = 10.27;
float *pokx = &x;
void *nesvrstan;         // pokazivač na void

nesvrstan = &n;          // preusmjeri na n
nesvrstan = pokx;        // preusmjeri na x
```

Međutim, pokazivaèu na `void` ne možemo izravno pridružiti vrijednost:

[†] Veæina današnjih procesora posjeduje mehanizme kojima se pojedini segmenti memorije mogu zaštititi od neovlaštene promjene. Zbog toga æe se gornji program prekinuti samo ako `pokx` pokazuje izvan dozvoljenog podruèja.

```
*nesvrstan = x;          // pogreška: void se ne može
                         //   dereferencirati
```

Buduæi da pokazivaè na `void` ne sadrži informaciju o tipu podatka na koji pokazuje, prilikom dohvaæanja vrijednosti moramo mu priložiti odgovarajuæe natuknice. Na primjer, želimo li u prethodnom primjeru ispisati sadržaj varijable `n` preko pokazivaèa `nesvrstan`, tada æemo morati napisati:

```
nesvrstan = &n;
cout << *(int*)nesvrstan << endl;
```

Prije ispisa, pokazivaèu na `void` treba dodijeliti tip pokazivaèa na `int`, da bi prevoditelj znao pravilno proèitati sadržaj memorije na koju `nesvrstan` pokazuje. Stoga na njega primjenjujemo operator dodjele tipa `(int*)`.

Zanimljivo je uočiti da pokazivači, unatoč tome što pokazuju na različite tipove podataka, na istom računalu uvijek zauzimaju jednake memorijske prostore[†]. U to se možemo osvjedočiti pomoću operatora `sizeof`: kako svi pokazivači zauzimaju jednaki memorijski prostor, naredbe za ispis

```
                                // duljine pokazivača na:
cout << sizeof(int*) << endl;   //   int,
cout << sizeof(float*) << endl; //   float,
double *xyz;
cout << sizeof(xyz) << endl;    //   double
```

æe dati isti ishod. Prepuštamo èitatelju da sam provjeri ovu tvrdnju.

6.2.1. Nul-pokazivaèi

Posebna vrijednost koji pokazivaè može imati jest *nul-pokazivaè* (engl. *null-pointer*). Takav pokazivaè ne pokazuje nikuda pa pokušaj pristupa sadržaju na koji takav pokazivaè pokazuje može završiti vrlo pogubno. Nema provjere tijekom izvoðenja programa je li vrijednost pokazivaèa nul-pokazivaè ili ne; na programeru je sva odgovornost da do takve situacije ne doðe. Pokazivaè se može inicijalizirati kao nul-pokazivaè tako da mu se jednostavno dodijeli nula:

```
int *x = 0;
```

U gornjem primjeru se pokazivaè na cijeli broj ne inicijalizira tako da ga se odmah usmjeri na neki broj, veæ se eksplicitno naznaèava da vrijednost pokazivaèa nije postavljena. Umjesto nule, možemo koristiti konstantu `NULL` koja je definirana u

---

[†] Izuzetak od ovog pravila postoji ako se koriste razlièiti modovi adresiranja memorije, kao što je sluèaj s DOS aplikacijama kod kojih postoje *bliski* (engl. *near*) i *daleki* (engl. *far*) pokazivaèi koji su razlièitih duljina.

standardnim bibliotekama. Tako se eksplicitnije naznaèava da se radi baš o nul-pokazivaèu. Na primjer:

```
int *x = NULL;
```

Postavlja se pitanje èemu uopæe služe nul-pokazivaèi, kada njihova primjena može imati poguban uèinak po izvoðenje programa. Ako se pokazivaè ne inicijalizira odmah nakon deklaracije, može mu se dodijeliti vrijednost nul-pokazivaèa. Prije upotrebe, provjerom vrijednosti pokazivaèa moguæe je ustanoviti je li on inicijaliziran ili ne.

6.2.2. Kamo sa zvijezdom (petokrakom)

Osvrnimo se na trenutak na naèin deklaracije pokazivaèa. U gornjim deklaracijama smo simbol za pokazivaè `*` pisali uz ime pokazivaèa. Opæenito su praznine oko znaka `*` proizvoljne: smiju se staviti na jednu i/ili drugu stranu ili se ne moraju uopæe staviti. Tako je dozvoljeno pisati:

```
float *ZvjezdicaUzIme;
float* ZvjezdicaUzTip;
float * ZvjezdicaOdmaknutaOdImenaITipa;
float*ZvjezdicaUzImeITip;
```

Više no oèito je zadnji naèin nepregledan, pa se ne koristi. Autori mnogih knjiga preferiraju pisanje zvjezdice uz tip:

```
float* kazalo;
```

jer je znak za pokazivaè `*` (zvjezdica) pridružen tipu, èime se tip podatka jaèe istièe kao "pokazivaè na `float`". Meðutim, ovaj naèin je nespretan pri višestrukim deklaracijama, na primjer:

```
int* kazaljka, nekazaljka;
```

Znak za pokazivaè u gornjoj deklaraciji je pridružen samo varijabli `kazaljka`, dok je `nekazaljka` obièni `int` (lat. *int vulgaris domesticus*). Želimo li ostaviti operator za pokazivaè uz tip na koji varijabla pokazuje, zbog dosljednosti bismo morali gornju deklaraciju razbiti na dvije:

```
int* kazaljka;
int* kazaljka2;
```

Pristupom koji æemo i mi koristiti u knjizi, izbjegnute su takve zamke:

```
int *PeterPointer, *WhereAreYou;
```

Osim toga, takav naèin pisanja dosljednije odražava prikaz pokazivaèa u izrazima. To je naroèito izraženo u izrazima s množenjem. Na primjer, u sljedeæem primjeru želimo pomnožiti dvije varijable na koje pokazuju pokazivaèi `poka` i `pokb`. Preglednije je pisati operatore dereferenciranja neposredno uz imena pokazivaèa:

```
int a = 5;
int b = 10;
int *poka = &a;
int *pokb = &b;

int c = *poka * *pokb;
```

Druga rješenja su manje pregledna:

```
int c = *poka * * pokb;
```

ili, još gore:

```
int c = *poka **pokb;
```

Jednaka razmatranja vrijede iz za `&` (operator adrese): kao i simbol za pokazivaè, on se može pisati uz oznaku tipa ili uz identifikator.

Koji od pristupa će čitatelj koristiti prepuštamo njegovom izboru. Mi smo se samo osjetili dužnima navesti razloge za ili protiv nekog pristupa, da bi se početnik lakše opredijelio. Jer, teže je mijenjati navike kada vam nešto "uđe u krv". U svakom slučaju, kada jednom odaberete neki pristup, držite ga se dosljedno do kraja.

6.2.3. Tajna veza između pokazivaèa i polja

U programskom jeziku C++ pokazivaèi i polja su međusobno èvrsto povezani. Iako to kod dohvaæanja preko indeksa nije oèito, èlanovi polja dohvaæaju se u biti preko pokazivaèa. Naredbom

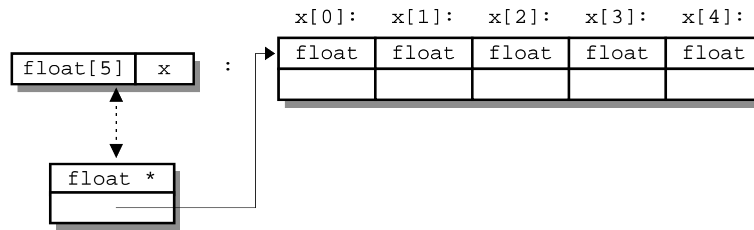
```
float x[5];
```

deklarira se jednodimenzionalno polje objekata koje se sastoji od pet èlanova tipa `float`. Pri tome samo ime `x` ima smisao pokazivaèa na prvi èlan polja `x[0]` (slika 6.10). Prilikom dohvaæanja èlanova polja, prevoditelj æe vrijednost indeksa pribrojiti pokazivaèu na prvi èlan; tako æe naredba

```
float a = x[2];
```

biti zapravo prevedena kao

```
float a = *(x + 2);
```



Slika 6.10. Veza između polja i pokazivača

Ova zadnja naredba može se interpretirati na sljedeæi naèin: uzmi adresu prvog èlana polja, poveæaj ju za dva, pogledaj što se nalazi na toj adresi, te pridruži vrijednost na toj lokaciji varijabli `a`. Valja uoèiti da se pritom adresa ne poveæava za dva bajta, veæ za dva segmenta u koje stanu podaci tipa `float`. Na primjer, ako se poèetak polja `x` nalazi na adresi 0x2000, a varijable tipa `float` zauzimaju 4 bajta, tada æe pokazivaè `x + 2` pokazivati na adresu 0x2008. U ovom primjeru upoznali smo se s aritmetièkim operacijama s pokazivaèima, o èemu æe ipak opširnije biti govora u sljedeæem potpoglavlju.

Zbog navedene veze između pokazivača i polja, ako se navede naziv polja bez indeksa, on ima značenje pokazivača na prvi član, te će sljedeći kôd ispisati iste vrijednosti:

```
int b[] = {10, 20, 30};
cout << b << endl;          // adresa početnog člana
cout << *b << endl;         // njegova vrijednost
```

Isto tako æe obje naredbe

```
cout << &(b[1]) << endl;    // adresa člana b[1]
cout << (b + 1) << endl;    // ista stvar, druga forma
```

ispisati memorijsku adresu u kojoj je pohranjen èlan polja `b[1]`. Oèito je da su pristupi èlanovima polja preko indeksa ili preko pokazivaèa potpuno ekvivalentni – koji pristup æe se koristiti ovisi iskljuèivo o ukusu programera.

Dovitljivi čitatelj će na osnovi ove povezanosti pokazivača i polja sada sam zaključiti zašto je početni indeks polja u jeziku C++ upravo nula, a ne 1 kao u nekim drugim programskim jezicima.

No važno je ipak razumjeti osnovnu razliku između pokazivača i polja. Deklaracijom

```
int x[5];
```

se ne stvara pokazivaè `x` koji pokazuje na polje; `x` je jednostavno samo sinonim za pokazivaè na prvi èlan polja. Njega možemo koristiti isti naèin na koji možemo koristiti

i ostale pokazivaèe, ali mu ne možemo promijeniti vrijednost. Sljedeæi kôd æe ispisati iste vrijednosti:

```
int x[5];
cout << "x: " << x << endl;
cout << "&x: " << &x << endl;
```

To nam svjedoèi da se u memoriji raèunala nigdje ne alocira prostor za pokazivaè `x`, pa `x` i `&x` imaju isti smisao. Poneki prevoditelji èak imaju upozorenje da se znak `&` ispred imena polja zanemaruje.

Polja i pokazivaèi su slièni, no valja biti svjestan njihove razlike. Ime polja je jednostavno sinonim za pokazivaè na poèetnu vrijednost, no sam pokazivaè nije nigdje alociran u memoriji.

Neposredna posljedica ovakvog rukovanja poljima jest nemoguænost izravnog pridruživanja sadržaja cijelog polja. Želimo li sve èlanove polja `a` preslikati u polje `b`, to neæemo moæi ostvariti jednostavnom operacijom pridruživanja:

```
float a[] = {10, 20, 30, 40};
float b[4];
b = a;                  // pogreška prilikom prevođenja
```

Potrebno je napisati petlju koja æe pojedinaèno dohvaæati èlanove izvornog polja i preslikavati ih u odredišno polje. Pokušaj pridruživanja pokazivaèa:

```
*b = *a;        // b[0] = a[0]
```

rezultirat æe preslikavanjem samo prvog èlana `a[0]` u prvi èlan `b[0]`. Ovakvo djelovanje operatora pridruživanja na polja je logièno, ako se razumije kako se polja prikazuju na razini strojnog kôda.

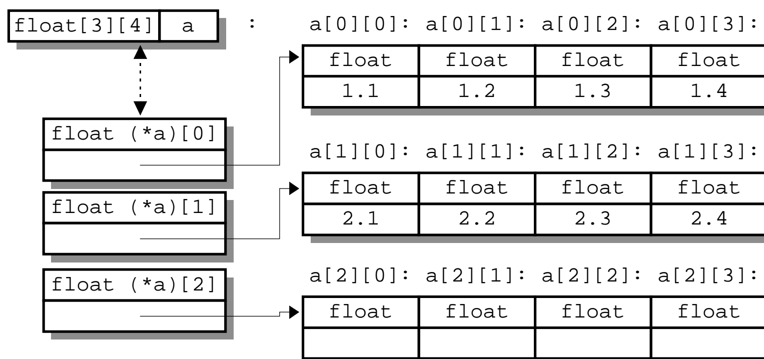
Promotrimo još vezu između pokazivača i višedimenzionalnih polja. Višedimenzionalna polja se pohranjuju u memoriju linearno (vidi sliku 6.7): polje deklarirano kao

```
float a[3][4];
```

može se stoga shvatiti kao niz od tri jednodimenzionalna polja, od kojih svako sadrži èetiri elementa, složenih jedno za drugim. Pritom su `a[0]`, `a[1]` i `a[2]` pokazivaèi na poèetne èlanove svakog od tih jednodimenzionalnih polja. U to se možemo osvjedoèiti sljedeæim primjerom:

```
float a[3][4] = {{1.1, 1.2, 1.3, 1.4},
                 {2.1, 2.2, 2.3, 2.4}};
cout << *a[0] << "\t" << *a[1] << endl;
```

Naredbom za ispis dobit æemo brojeve 1.1 i 2.1, tj. poèetne èlanove prva dva jednodimenzionalna broja. Slijedi da se dvodimenzionalno polje može interpretirati kao polje pokazivaèa, od kojih svaki pokazuje na poèetak jednodimenzionalnog polja (slika 6.11).



Slika 6.11. Prikaz dvodimenzionalnog polja preko polja pokazivača

Vrijedi ista ograda kao u sluèaju pokazivaèa na jednodimenzionalno polje: `a[1]` je samo sinonim za pokazivaè na èlan `a[1][0]`, no on nije nigdje u memoriji posebno alociran. Zbog toga mu se ne može uzeti adresa (`a[1]` i `&a[1]` su iste vrijednosti), te mu se ne može mijenjati vrijednost.

6.2.4. Aritmetièke operacije s pokazivaèima

Na pokazivaèima su definirane neke aritmetièke operacije, kao i usporedbe pokazivaèa. Na primjer, moguæe je pokazivaèu pribrojiti ili oduzeti cijeli broj, raèunati razliku izmeðu dva pokazivaèa, te usporeðivati pokazivaèe.

Ako se pokazivaču pribroji ili oduzme cijeli broj (konstanta, varijabla ili izraz), rezultat takve operacije bit će pokazivač koji pokazuje na memorijsku lokaciju udaljenu od početne za navedeni broj objekata dotičnog tipa. Na primjer, ako se od pokazivača na objekt tipa `int` oduzme broj 3, dobit će se pokazivač koji pokazuje na memorijsku lokaciju koja je za `3 * sizeof(int)` manja od početne:

```
int a;
int  *poka = &a;
cout << poka << endl << poka - 3 << endl;
```

Ako na raèunalu na kojem radimo tip `int` zauzima dva bajta, gornji programski odsjeèak æe ispisati dvije memorijske adrese od kojih je druga za 2 * 3 = 6 manja od prve. Opæenito, svaki izraz tipa

```
T *pok1, *pok2;
int i;
// ...
pok2 = pok1 + i;
```

može se interpretirati kao

```
pok2 = (T *)(((char *)pok1) + i * sizeof(T));
```

Pri tome je `T` neki proizvoljan tip, a `i` neki cijeli broj. Gornja naredba se treba èitati ovako: "Pretvori `pok1` u pokazivaè na char, zato jer je `char` tip duljine jednog bajta. Dobivenu adresu uveæaj za broj koji je jednak `i`-strukoj velièini tipa `T`. Tako dobivenu adresu pretvori natrag u pokazivaè na tip `T`."

Za pokazivače su dozvoljeni i skraćeni aritmetički operatori, kao *inkrement*, *dekrement* te operatori *obnavljajućeg pridruživanja*.

Često se aritmetika s pokazivačima koristi u vezi s poljima. Naime, kako su članovi polja složeni linearno u memoriji, a ime polja predstavlja pokazivač na njegov prvi član, pomoću pokazivačke aritmetike je jednostavno moguće pristupati susjednim članovima:

```
float x[10];
float *px = &x[3];
float x2 = *(px - 1);       // x[2] – prethodni član
float x5 = *(px + 1);       // x[4] – sljedeći član
```

Dozvoljeno je meðusobno oduzimanje pokazivaèa samo na objekte istog tipa. Rezultat takvog izraza broj objekata tog tipa koji bi se mogli smjestiti izmeðu ta dva pokazivaèa. Oduzimanje pokazivaèa se èesto koristi kada pokazivaèi koje oduzimamo pokazuju na èlanove istog polja: rezultat je tada jednak razlici indeksa èlanova na koje ti pokazivaèi pokazuju:

```
float x[10];
float *prviClanUNizu = x;
float *zadnjiClanUNizu = &x[9];
int razmak = zadnjiClanUNizu - prviClanUNizu;       // = 9
razmak = prviClanUNizu - zadnjiClanUNizu;           // = -9
```

Pokazivaèe je moguæe usporeðivati. Pri tome se usporeðuju memorijske adrese na koje pokazivaèi pokazuju. U nastavku gornjeg primjera

```
if (prviClanUNizu >= zadnjiClanUNizu)
    cout << "Frka!";
```

Poseban smisao ima operator jednakosti, kojim se utvrðuje da li dva pokazivaèa pokazuju na istu memorijsku lokaciju. Takoðer, èesto se pokazivaè usporeðuje s nul-vrijednosti:

```
#include <stdio.h>

int main() {
    FILE *pok;
    pok = fopen("datoteka.dat", "r");
    if (pok == 0)
        cout << "Nema tražene datoteke.";
}
```

Gornji primjer poziva funkciju `fopen()` iz standardne biblioteke `stdio.h` koja otvara neku datoteku. U sluèaju da ta datoteka postoji, vratit æe se pokazivaè na objekt tipa `FILE` (definiranog također u datoteci `stdio.h`), pomoæu kojeg se kasnije može pristupati sadržaju datoteke. Ako operacija ne uspije, vratit æe se nul-pokazivaè. Gornju provjeru možemo skraæeno napisati i ovako:

```
if (!pok)   // ...
```

Ilustrirajmo primjenu pokazivaèke aritmetike programom kojim se u polje brojeva sortiranih po velièini umeæe novi èlan. Tada je potrebno sve èlanove polja veæe od novoga èlana pomaknuti prema kraju polja, tako da se napravi prazno mjesto. Radi preglednosti, kôd je maksimalno pojednostavljen, te nema provjere da li je broj èlanova u polju nadmašio njegovu duljinu (èitatelju prepuštamo da nadopuni kôd odgovarajuæom provjerom).

```
// inicijalizirajmo varijable
int brClanova = 9;
int polje[20] = {10, 20, 30, 40, 50, 60, 70, 80, 90};
// pokazivač na lokaciju iza zadnjeg broja:
int *odrediste = polje + brClanova;
int noviBroj = 15;
// kreće od zadnjeg (najvećeg) broja
while (*(--odrediste) > noviBroj) {
    if (odrediste < polje)      // prošao je početak polja,
        break;                  // pa prekida petlju
    *(odrediste + 1) = *odrediste;
                                // pomiče veće brojeve
}
*(odrediste + 1) = noviBroj;    // umeće novi broj
brClanova++;
```

Srž gornjeg programa èini `while` petlja koja brojeve koji su veæ smješteni u polju uspoređuje s `noviBroj` kojeg želimo umetnuti. Pretraživanje kreæe od zadnjeg (najveæeg) broja u nizu, a istovremeno s pretraživanjem pomièu se èlanovi niza veæi od novopridošlog. Može se vidjeti kako je za pomicanje po elementima polja korišten pokazivaè `odrediste` koji se dekrementira u svakom prolasku kroz petlju. Tako se na jednostavan naèin omoguæava prolazak kroz sve èlanove polja.

> Aritmetika s pokazivaèima ima prednosti kod dohvaæanja uzastopnih èlanova polja.

Varijable i èlanovi polja inicijalizirani su proizvoljnim vrijednostima na poèetku kôda da bi se lakše shvatio njegov smisao. Ako bi polje inicijalno bilo prazno, tada bismo uzastopnim pozivanjem gornjeg kôda dobili program koji uèitava podatke te ih odmah slaže po velièini.

Usredotočimo se na operacije s pokazivačima unutar `while` petlje. Osim usporedbe u uvjetu izvođenja petlje, zanimljivo je uočiti usporedbu pokazivača `odrediste` i `polje` u `if` naredbi, gdje se koristi usporedba pokazivača za određivanje uvjeta prekida petlje. Ako `odrediste` pokazuje na adresu manju od one na koju pokazuje pokazivač `polje`, petlja se prekida, jer je program tijekom pretraživanja stigao na početak polja, ne našavši član u nizu manji od novog broja te `noviBroj` treba umetnuti na početak polja. Također uočimo pridruživanje:

```
*(odrediste + 1) = *odrediste;
```

Na prvi pogled se èini da bi prevoditelj trebao javiti pogrešku da se s lijeve strane operatora `=` ne nalazi *lvrijednost*. Meðutim, razmotrimo li stvarno znaèenje gornje naredbe, nedoumice æe nestati: vrijednost pohranjenu u memoriji na adresi na koju pokazuje `odrediste` ne pridružujemo varijabli `odrediste + 1`, veæ pohranjujemo u memoriju na adresu koja je za jedan memorijski blok veæa od adrese `odrediste`.

Gornju petlju mogli smo napisati i kraće, tako da pomake brojeva ubacimo u uvjet izvođenja `while` petlje:

```
while ((*(odrediste--) = *(odrediste - 1)) > noviClan)
    if (odrediste < polje)
        break;
```

Zadatak. *U gornji kôd ubacite provjeru broja unesenih članova niza tako njegova duljina ne nadmaši duljinu polja. Također ubacite provjeru da li broj već postoji u nizu, te onemogućite višekratno pohranjivanje jednakih brojeva.*

6.2.5. Dinamièko alociranje memorije operatorom `new`

Do sada smo pokazivaè uvijek usmjeravali na prethodno deklariranu i (eventualno) inicijaliziranu varijablu:

```
float nekiBroj;
float *pokazivacNaTajBroj = &nekiBroj;
*pokazivacNaTajBroj = 34.234;
```

Deklaracija varijable `nekiBroj` u ovom primjeru treba nam samo da bi se osigurao memorijski prostor na koji æe `pokazivacNaTajBroj` biti usmjeren – dalje sve možemo napraviti preko pokazivaèa. Je li deklaracija varijable `nekiBroj` baš neophodna ili ju možemo nekako izbjeæi? Drugim rijeèima, je li moguæe istodobno s deklaracijom pokazivaèa rezervirati memorijski prostor za objekt èija æe vrijednost biti tamo pohranjena? Odgovor na to pitanje pruža sljedeæi kod:

```
float *pokaZivac;
pokaZivac = new float();
*pokaZivac = 10.5;
```

Operatorom `new` rezervira se potreban memorijski prostor za objekt tipa `float`. Iza kljuène rijeèi `new` navodi se tip podatka za koji se osigurava memorijski prostor, a zatim (neobavezno) par okruglih zagrada. Ako je operacija alociranja uspješno izvedena, operator vraæa adresu alociranog prostora koju pridružujemo našem pokazivaèu `pokaZivac`. Ako postupak nije bio uspješan (na primjer, nema dovoljno memorije za zahtijevani objekt), operator vraæa nulu, odnosno nul-pokazivaè. Zato bismo, radi vlastite sigurnosti, trebali provjeriti rezultat operatora `new` te eventualno javiti pogrešku ako alokacija memorije ne uspije:

```
float *pokaZivac = new float();
if (pokaZivac == 0)
    cout << "Pun sam kao šipak!" << endl;
//...
```

Unutar okruglih zagrada iza oznake tipa može se definirati poèetna vrijednost objekta, što znaèi da smo prethodni primjer mogli pisati još kraæe:

```
float *pokaZivac = new float(10.5);
if (!pokaZivac)
    cout << "Pun sam kao šipak!" << endl;
//...
```

Alocirani prostor zauzet operatorom `new` se ne oslobaða automatski prilikom izlaska iz bloka naredbi. Do sada smo koristili iskljuèivo *automatske objekte* (engl. *automatic objects*) kojima je prevoditelj sam alocirao memorijski prostor pa se sam i brinuo o oslobaðanju tog prostora pri izlasku iz bloka unutar kojeg je objekt bio deklariran. Za pohranjivanje takvih objekata koristi se *stog* (engl. *stack*), zasebni dio memorije u koji se pohranjuju privremeni podaci. Izlaskom iz bloka naredbi objekti koji su bili deklarirani u bloku proglašavaju se nepotrebnima i preko njih se prepisuju novi podaci. Istina, ako nema novih podataka, ti podaci još mogu postojati zapisani u memoriji, ali prevoditelj više ne garantira njihov integritet, pa ih stoga proglašava nedohvatljivima.

Za razliku od automatskih objekata, prostor za *dinamičke objekte* (engl. *dynamic objects*) alocira programer operatorom `new`. Dinamički objekti se smještaju u javni dio memorije, tzv. *hrpu* (engl. *heap*). Budući da prevoditelj ne kontrolira "čišćenje" tog dijela memorije, sam programer mora uništiti dinamičke objekte kada mu više ne

trebaju, ali svakako prije završetka programa – u protivnom će se izvođenjem programa bespovratno smanjivati raspoloživa memorija[†].

Uništavanje dinamičkih objekata obavlja se operatorom `delete`; kada nam više prostor na koji `pokaZivac` pokazuje nije potreban, napisat ćemo naredbu:

```
delete pokaZivac;
```

Iza kljuène rijeèi `delete` navodi se ime pokazivaèa na lokaciju koju treba osloboditi. Naravno da pritom taj pokazivaè mora biti dohvatljiv – iako je prostor na koji pokazivaè pokazuje alociran dinamièki, sam `pokaZivac` je deklariran kao automatski objekt, te mu se po izlasku iz bloka gubi svaki trag.

Valja uočiti da primjena operatora `delete` ne podrazumijeva i brisanje sadržaja objekta – njime samo prostor koji je zauzimao objekt postaje slobodnim za odstrel, tj. za druge dinamičke objekte. Stoga je gotovo sigurno da će izvođenje naredbi:

```
int *varijablaKojaNestaje = new int(123);
delete varijablaKojaNestaje;
cout << *varijablaKojaNestaje << endl;
```

ispisati broj 123, unatoè tome da je prethodno osloboðen prostor koji je `varijablaKojaNestaje` zauzimala. Sadržaj objekta nakon primjene operatora `delete` je neodreðen.

6.2.6. Dinamièka alokacija polja

Operatorom `new` se može alocirati prostor i za polje objekata. To omoguæava da se duljina polja definira tijekom izvoðenja programa, umjesto da ju moramo definirati prije prevoðenja kao što je to sluèaj kod automatskih polja.

Kod alokacije polja operatorom `new` treba u uglatim zagradama nakon oznake tipa navesti duljinu polja. Tako će naredba

```
int *sati = new int[12];
```

alocirati prostor za polje od 12 cjelobrojnih èlanova. Operator `new` æe u sluèaju uspješnog osiguranja memorijskog prostora kao rezultat vratiti pokazivaè na poèetak polja. U sluèaju neuspjeha, vraæa se nul-pokazivaè.

Kao što se za oslobađanje prostora dinamičkih varijabli koristi operator `delete`, za oslobađanje alociranih polja se koristi operator `delete []`, tako da se iza uglate zagrade navede ime polja:

```
delete [] sati;
```

[†] To nije toèno za sve prevoditelje jer mnogi prevoditelji automatski generiraju kôd koji uništava sve dinamièke objekte prilikom okonèanja programa. No na to svojstvo se ne treba oslanjati, veæ valja uredno osloboditi svu zauzetu memoriju ruèno.

Vrlo je lako zaboraviti umetnuti par uglatih zagrada prilikom oslobaðanja polja. Prevoditelj neæe javiti nikakvu pogrešku, pa èak niti upozorenje, no takav kôd može prouzroèiti probleme prilikom izvoðenja. Operator `delete` oslobaða memoriju samo jednog èlana, dok operator `delete []` prvo provjerava duljinu polja, te uništava svaki element polja.

Operatori `delete` i `delete[]` nisu isti operatori. `delete` služi za oslobaðanje jednog objekta, dok `delete []` oslobaða polja.

Kao primjer za alokaciju i dealokaciju polja poslužit æe nam program kojemu je svrha za zadane toèke izraèunati koeficijente pravca koji se može najbolje provuæi izmeðu njih. Zamislimo trkaèa koji trèi stalnom brzinom. Mjerenjem meðuvremena na svakih 100 metara želimo odrediti prosjeènu brzinu kojom on trèi, te koliko æe mu vremena trebati da pretrèi stazu dugaèku 800 metara. Rezultati mjerenja dani su u tablici 6.1.

Tablica 6.1. Ulazni podaci za primjer dinamièkog alociranja polja.

s	0	100 m	200 m	300 m	400 m	500 m
t	0	13 s	31 s	46 s	63 s	76 s

Prikažemo li te podatke grafièki (slika 6.12), uoèit æemo da su mjereni podaci grupirani oko pravca. Nagib tog pravca odgovara prosjeènoj brzini trkaèa, a ekstrapolacijom pravca na 800 m moæi æemo procijeniti vrijeme potrebno za cijelu stazu.

Jednadžbu pravca odredit ćemo postupkom najmanjih kvadrata. Koeficijenti pravca

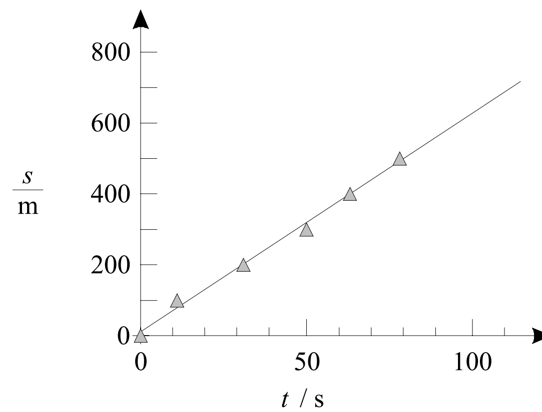
$y = a \cdot x + b$

odreðeni su izrazima

$$a = \frac{1}{\sum_{i=0}^{n-1} (x_i - \bar{x})^2} \cdot \sum_{i=0}^{n-1} (x_i - \bar{x}) \cdot y_i \, ,$$

$$b = \bar{y} - a \cdot \bar{x} \, ,$$

gdje su



Slika 6.12. Linearna interpolacija mjerenih točaka.

$$\bar{x} = \frac{1}{n}\sum_{i=0}^{n-1} x_i\,, \qquad \bar{y} = \frac{1}{n}\sum_{i=0}^{n-1} y_i$$

srednje vrijednosti koordinata toèaka. Pokušajmo napisati program koji koristi gore opisanu metodu najmanjih kvadrata. Prvo treba uèitati broj mjernih toèaka i pripadajuæe koordinate:

```
#include <iostream.h>

int main() {
    cout << "Koliko točaka mogu očekivati? ";
    int n;                      // broj točaka
    cin >> n;

    float *x = new float[n];    // alocira se prostor za
    float *y = new float[n];    // polja s koordinatama

    double srednjiX = 0.0;      // srednja vrijednost od x
    double srednjiY = 0.0;      // srednja vrijednost od y
    cout << "Upiši koordinate!" << endl;
    for (int i = 0; i < n; i++) {
        cout << "t(" << i+1 << ") = ";
        cin >> x[i];
        cout << "s(" << i+1 << ") = ";
        cin >> y[i];
        srednjiX += x[i];
        srednjiY += y[i];
    }
    srednjiX /= n;
```

```
    srednjiY /= n;
    // nastavak slijedi...
```

Koordinate toèaka pohranjuju se u dva jednodimenzionalna polja (`x` i `y`) za koja je alociran prostor operatorom `new`:

```
float *x = new float[n];
float *y = new float[n];
```

Iza oznake tipa podatka, u uglatim zagradama naveden je broj èlanova polja. Kao i kod alokacije pojedinaènih varijabli, operator `new` vraæa kao rezultat pokazivaè na poèetak polja ili nul-pokazivaè, ovisno o uspjehu alokacije (*Sa štitom ili na štitu!*[†]). Ovo je jedini naèin za alokaciju polja èija je duljina poznata prilikom izvoðenja – ako bismo koristili automatska polja, bilo bi neophodno predvidjeti najveæi moguæi broj elemenata koji æe se smještati u polje, te deklarirati takvo polje. U veæini sluèajeva bi to znaèilo uludo trošenje memorije, a s druge strane nemamo jamstva da æe predviðena duljina polja uvijek biti dostatna. Ovako se alocira upravo potreban broj èlanova, èime se racionalizira potrošnja memorije.

Pri učitavanju koordinata u petlji, u varijable `srednjiX` i `srednjiY` se akumuliraju njihove sume, a po izlasku iz petlje se iz njih računaju srednje vrijednosti. Slijedi petlja u kojoj se računaju sume u brojniku i nazivniku izraza za koeficijent smjera `a`:

```
    // ... nastavak
    double a = 0.0;              // koeficijent smjera
    double nazivnik = 0.0;       // nazivnik izraza za k.s.
    for (i = 0; i < n; i++) {
        double deltaX = x[i] - srednjiX;
        a += deltaX * y[i];
        nazivnik += deltaX * deltaX;
    }
    a /= nazivnik;
    double b = srednjiY - a * srednjiX;
    // ima još...
```

Buduæi da nam polja `x` i `y` više ne trebaju, operatorom `delete []`æemo osloboditi prostor kojeg oni zauzimaju, a potom ispisati rezultate (za podatke iz tablice dobiva se prosjeèna brzina trkaèa od 6,41 m/s, te 124 sekunde potrebne za stazu od 800 m):

```
    // ...konac djelo krasi:
    delete [] x;             // oslobađamo prostor
    delete [] y;

    cout << "Prosječna brzina trkača je "
```

[†] Pozdrav kojim su spartanski vojnici otpravljani u ratove. Znaèi: “Vratite se kao pobjednici ili nam se ne vraæajte živi pred oèi!”

```
        << a << " m/s" << endl;
    cout << "Ako bude jednako uporan, za 800 metara "
         << "trebat će mu oko " << (800 - b) / a
         << " sekundi." << endl;
    return 0;
}
```

Dealokacija polja `x` i `y` provedena je tako da je između operatora `delete` i imena polja umetnut prazan par uglatih zagrada:

```
delete [] x;
delete [] y;
```

èime se naznaèava da se oslobaða polje. Unutar zagrada ne smije biti upisano ništa – prevoditelj zna kolika je duljina polja, te æe na osnovi toga osloboditi odgovarajuæi prostor. U starijim inaèicama C++ jezika, unutar uglatih zagrada trebalo je navesti duljinu polja, meðutim standard sada to ne zahtijeva i ne dozvoljava.

Spomenimo još nekoliko važnih činjenica vezanih uz alokaciju polja: za razliku od alokacije prostora za pojedinačne varijable, prilikom alokacije polja članovi se ne mogu inicijalizirati. Također, unatoč činjenici da je polje alocirano tijekom izvođenja programa, duljina tog polja se ne može naknadno dinamički mijenjati. Na primjer, što napraviti ako broj članova polja postane veći od duljine alociranog polja? Poneki dovitljivi "haker" bi mogao doći na ideju za sljedeći kôd:

```
float *prvoPolje = new float[n];
// ...
delete [] prvoPolje;
float *novoPolje = new float[m];
```

u vjeri da æe se `novoPolje` smjestiti na mjesto nastalo dealokacijom polja `prvoPolje`, tako da æe se postojeæi èlanovi iz `prvoPolje` jednostavno pojaviti u novom polju. Meðutim, ne postoji nikakva garancija da æe se takvo što i dogoditi (štoviše, to je gotovo nevjerojatno). Prema tome, kada alocirano polje postane pretijesno, jedini pouzdan naèin da se osigura dodatni prostor jest alociranje potpuno novog polja veæe duljine, kopiranje svih postojeæih èlanova u novo polje, te uništenje starog polja. Daleko fleksibilnije promjene duljine niza podataka omoguæavaju *vezane liste* (engl. *linked lists*) s kojima æemo se upoznati kasnije.

6.2.7. Dinamièka alokacija višedimenzionalnih polja

Nakon što smo savladali dinamièku alokaciju jednodimenzionalnih polja, upoznat æemo se i s alokacijom višedimenzionalnih polja. Kao što smo u odsjeèku 6.2.3 vidjeli, dvodimenzionalno se polje sastoji od niza jednodimenzionalnih polja jednake duljine. Stoga prilikom alokacije primjerice dvodimenzionalnog polja 3×4, u suštini treba alocirati prostor za 3 jednodimenzionalna polja duljine 4:

```
float (*a)[4] = new float[3][4];
```

Operator `new` vraæa pokazivaèe na poèetke tih jednodimenzionalnih polja, tj. adresu polja od 3 pokazivaèa, od kojih svaki pokazuje na poèetak jednodimenzionalnog polja s 4 èlana (vidi sliku 6.11 na strani 124). Okrugla zagrada oko pokazivaèa `*a` je neophodna jer uglata zagrada ima viši prioritet od znaka za pokazivaè. Izostavljanjem zagrada:

```
float *a[4] = new float[3][4];      // pogreška
```

u stvari se `a` deklarira kao pokazivaè na polje od èetiri èlana, te prevoditelj prijavljuje pogrešku. Oslobađanje prostora se obavlja identièno kao i kod jednodimenzionalnih polja:

```
delete [] a;
```

Treba naglasiti da kod alokacije prostora za višedimenzionalna polja duljine svih dimenzija osim prve moraju poznate prilikom prevođenja. Zbog toga æe naredba

```
double (*b)[n] = new double[10][n];     // pogreška
```

prouzroèiti pogrešku pri prevođenju, dok je naredba

```
double (*c)[5] = new double[n][5];      // OK
```

dozvoljena. Razlog tome leži u naèinu na koji generirani strojni kôd pristupa poljima. Dvodimenizonalno polje se slaže u memoriju tako da se reci upisuju uzastopno jedan iza drugoga. Da bi mogao odrediti poèetak svakog pojedinog retka, prevoditelju je potreban podatak o duljini redaka, a to je broj stupaca.

6.2.8. Pokazivaèi na pokazivaèe

Kao što je moguæe definirati pokazivaèe na objekte bilo kojeg tipa, moguæe je definirati i pokazivaèe na pokazivaèe. Netko æe se priupitati èemu je to potrebno; zar nije dovoljna fleksibilnost postignuta veæ samim pokazivaèem, koji može pokazivati na bilo koje mjesto u memoriji? Da bismo odagnali sve nedoumice, èitatelju æemo podastrijeti vrlo praktièan primjer dinamièkog alociranja prostora za dvodimenzionalno polje. Za razliku od gornjeg primjera u kojem sve dimenzije polja moraju biti poznate u trenutku alociranja memorijskog prostora, ovaj primjer (preuzet iz [Borland94]) pruža daleko veæu fleksibilnost, jer broj èlanova ne mora biti jednak u svakom retku.

Pogledajmo prvi dio kôda u kojem se alocira neophodan prostor za polje, te se članovima pridružuju vrijednosti. Budući da reci mogu imati proizvoljan broj članova, valja pohraniti podatke o broju članova za svaki redak. Taj broj ćemo pohraniti kao prvi podatak u svakom retku, prije vrijednosti samih članova.

```
#include <iostream.h>

int main() {
    float **dvaDpolje;
    int i, redaka, stupaca;

    cout << "Broj redaka: ";
    cin >> redaka;
    dvaDpolje = new float*[redaka];
    for (i = 0; i < redaka; i++) {
        cout << "Broj članova u " << (i+1) << ". retku:";
        cin >> stupaca;
        dvaDpolje[i] = new float[stupaca + 1];
        dvaDpolje[i][0] = stupaca;
        for (int j = 1; j <= stupaca; j++)
            dvaDpolje[i][j] = i + 1 + j / 100.;
    }

    cout << "Ispis članova polja:" << endl;
    for (i = 0; i < redaka; i++)
        for (int j = 1; j <= dvaDpolje[i][0]; j++)
            cout << "[" << i << "][" << j << "] = "
                 << dvaDpolje[i][j] << endl;

// nastavak slijedi nakon raščlanbe...
```

Deklaracijom

```
float **dvaDpolje;
```

varijablu `dvaDpolje` smo proglasili pokazivaèem na pokazivaè na varijablu tipa `float`. Naredbom

```
dvaDpolje = new float*[redaka];
```

alociran je prvo prostor za polje pokazivaèa na `float`. Ti æe pokazivaèi biti usmjereni na poèetke pojedinih redaka. Slijedi petlja po recima unutar koje se alociraju prostori za èlanove svakog pojedinog retka (slika 6.13):

```
dvaDpolje[i] = new float[stupaca + 1];
```

Duljine polja za podatke po recima su veæe od broja èlanova za podatak o duljini svakog retka.

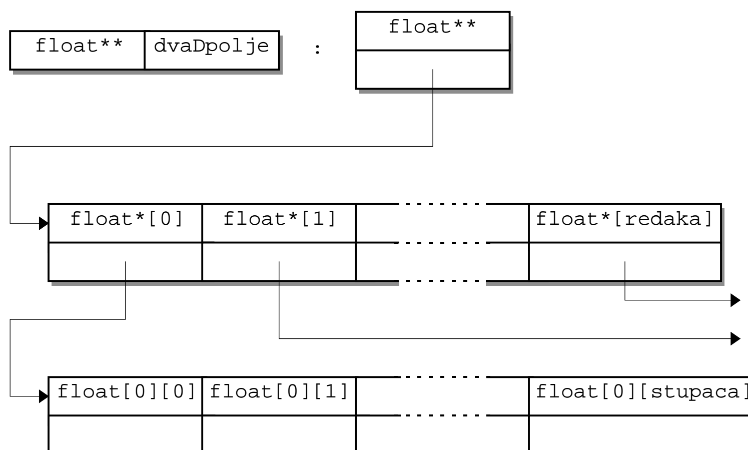
Nakon što polje postane nepotrebno, prostor treba osloboditi:

```
    // nastavak: oslobađanje prostora
    for (i = 0; i < redaka;  i++)
        delete [] dvaDpolje[i];
    delete [] dvaDpolje;
    return 0;
}
```

Prvo se u petlji oslobaðaju prostori za polja pojedinih redaka, a tek potom se uklanja pokazivaè (na pokazivaè) `dvaDpolje`. Redoslijed oslobaðanje je vrlo važan – mora se obavljati obrnutim redoslijedom od onoga pri pridruživanju. Naime, ako bi se prvo uklonio poèetni pokazivaè na pokazivaè `dvaDpolje`, izgubila bi se veza prema pokazivaèima na poèetke pojedinih redaka, pa ih ne bi bilo moguæe više dohvatiti. Time bi bila onemoguæena dealokacija prostora za pojedine retke.

6.3. Nepromjenjivi pokazivaèi i pokazivaèi na nepromjenjive objekte

Kljuènom rijeèi `const` možemo deklarirati nepromjenjivi objekt (simbolièku



Slika 6.13. Pokazivač na pokazivač

konstantu). Vidjeli smo da æe na pokušaj promjene takvog objekta prevoditelj javiti pogrešku:

```
const float pi = 3.14;
pi = 3.1415926;             // pogreška
```

Pokušajmo nadmudriti prevoditelja tako da na `pi` usmjerimo pokazivaè, te `pi` promijenimo posredno preko pokazivaèa:

```
float *pipok = &pi;         // pogreška
*pipok = 3.1415926;
```

Ni prevoditelj nije veslo sisao! Prepoznat æe naš podmukli pokušaj da na simbolièku konstantu usmjerimo pokazivaè i preduhitriti nas javljajuæi pogrešku pri pridruživanju adrese nepromjenjivog objekta. Da bi se sprijeèile nehajne promjene simbolièkih konstanti, na njih se smiju usmjeravati samo pokazivaèi koji su eksplicitno deklarirani tako da pokazuju na nepromjenjive objekte:

```
float const *pipok;   // pokazivač na nepromjenjivi objekt
pipok = &pi;
```

Kao što smo vidjeli, u protivnom prevoditelj prijavljuje pogrešku. Tako deklarirani pokazivaè smije se preusmjerivati:

```
const float e = 2.71828;
pipok = &e;                 // OK
```

Štoviše, može se usmjeriti i na promjenjivi objekt:

```
float r = 23.4;
pipok = &r;                 // OK
```

Promjena varijable u tom sluèaju moguæa je samo izravno, ali ne i preko pokazivaèa na konstantu:

```
r = 45.2;                   // OK
*pipok = 9.23               // pogreška
```

Prema tome, usmjeravanjem pokazivaèa, koji je deklariran kao pokazivaè na konstantu, ne osigurava se nepromjenjivost tog objekta – deklaracijom pokazivaèa kao pokazivaèa na konstantni objekt dozvoljava se samo njegovo preusmjeravanje na nepromjenjive objekte te se spreèavaju promjene takvih objekata posredno preko pokazivaèa.

Također je moguće deklarirati nepromjenjivi pokazivač koji može pokazivati samo na jedan objekt. Takav pokazivač se mora inicijalizirati, te se kasnije ne može preusmjerivati:

```
double nekiBroj = 1989.1005;
double* const neMrdaj = &nekiBroj;  // nepromjenjivi
                                    // pokazivač
```

Objekt na koji pokazivaè pokazuje može se po volji mijenjati, izravno ili preko pokazivaèa:

```
nekiBroj = 1993.1009;           // OK
*neMrdaj = 1992.3001;           // OK
```

Meðutim, pokušaj preusmjeravanja nepromjenjivog pokazivaèa na neki drugi objekt sprijeèit æe prevoditelj, javljajuæi pogrešku:

```
double nekiDrugiBroj = 1205;
neMrdaj = &nekiDrugiBroj;       // pogreška
```

Konaèno, moguæe je deklarirati i nepromjenjivi pokazivaè na nepromjenjivi objekt:

```
const float q = 1.602e-19;
const float* const nabojElektrona = &q;
```

Ovako složena sintaksa èak i iskusnim programerima zadaje glavobolje. Evo vrlo jednostavnog naputka za prepoznavanje tipa podataka [Barton94]: èitajte deklaraciju tipa s desna na lijevo, tj. od identifikatora prema naprijed.

Na svaki pokušaj preusmjeravanja pokazivaèa `nabojElektrona`, te na svaki pokušaj promjene vrijednosti varijable na koju on pokazuje, prevoditelj æe javiti pogrešku.
Na primjer, deklaracija `int *` se, prema gornjem naputku, može èitati kao *pokazivaè na `int`*, deklaracija `const double *` kao *pokazivaè na `double` koji je nepromjenjiv*, deklaracija `double * const` kao *nepromjenjivi pokazivaè na podatak tipa `double`*, a `const float * const` se može èitati kao *nepromjenjivi pokazivaè na `float` koji je nepromjenjiv*.

Promjenjivost ili nepromjenjivost pokazivača je dio tipa: tipovi `const char *` i `char *` su različiti tipovi te ih nije moguće koristiti jedan umjesto drugoga. No postoji standardna konverzija pokazivača koja automatski pretvara pokazivač na promjenjivi objekt u pokazivač na konstantan objekt. Da bismo to ilustrirali, deklarirat ćemo sljedeće pokazivače:

```
const char *pokKonst;
char *pokPromj;
```

Vrijednost pokazivaèa `pokPromj` dozvoljeno je dodijeliti pokazivaèi `pokKonst`:

```
pokKonst = pokPromj;
```

Gornja dodjela je dozvoljena jer time integritet objekta na koji `pokPromj` pokazuje neæe biti narušen: jedino se vrijednost objekta neæe moæi promijeniti preko pokazivaèa `pokKonst`. No obrnuta dodjela neæe biti dozvoljena:

```
pokPromj = pokKonst;            // pogreška
```

U gornjem sluèaju se pokazivaè na konstantni objekt pokušava dodijeliti pokazivaèu na promjenjivi objekt. Ako bi to bilo dozvoljeno, tada bi postojala opasnost da se preko pokazivaèa `pokPromj` promijeni vrijednost konstantnog objekta, što bi narušilo njegov integritet. U sluèaju da ipak bezuvjetno (glavom kroz zid) želimo provesti pridruživanje, to moramo uèiniti pomoæu eksplicitne dodjele tipa kojom se *odbacuje* (engl. *cast away*) konstantnost pokazivaèa `pokKonst`:

```
pokPromj = (char *)pokKonst;    // OK
```

Gornja naredba sada prevoditelju daje na znanje da programer na sebe preuzima svu odgovornost za eventualnu promjenu konstantnog objekta.

Sve što je rečeno za nepromjenjive pokazivače i pokazivače na nepromjenjive objekte vrijedi i za `volatile` pokazivače i pokazivače na `volatile` objekte. Tako je moguće deklarirati različite pokazivače:

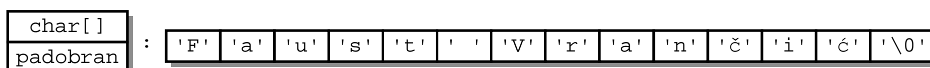
```
volatile int *pok1;     // volatile pokazivač na int
int volatile *pok2;     // pokazivač na volatile int
```

Također, prilikom dodjele pokazivaèa, osim po tipu pokazivaèi se moraju slagati i po `volatile` kvalifikatoru.

6.4. Znakovni nizovi

Za pohranjivanje tekstova koriste se *znakovni nizovi* (engl. *character strings*, kraæe *strings*). U suštini su to jednodimenzionalna polja èije èlanove èine znakovi (`char`). Prilikom inicijalizacije znakovnog niza, njegov sadržaj se navodi unutar para znakova `"` (dvostruki navodnici):

```
char minijature[] = "Julije Klović";
char padobran[] = "Faust Vrančić";
```



Slika 6.14. Pohranjivanje znakovnog niza u memoriji

Na slici 6.14 prikazan je predložak po kojem se pohranjuje znakovni niz u memoriji raèunala. Kao što se vidi, znakovni niz se sastoji od èlanova tipa `char` iza kojih slijedi znak `'\0'`.

Svaki pravilno inicijalizirani znakovni niz sadrži i zaključni znak `'\0'` (*nul-znak*, engl. *null-character*). Njega kod inicijalizacije nije potrebno eksplicitno navesti, ali treba voditi raèuna da on zauzima jedno znakovno mjesto.

Za provjeru, napišimo petlju koja æe ispisati pojedine èlanove niza `padobran`:

```
for (int i = 0; i < 20; i++)
    cout << padobran[i] << "\t"
         << (int)(unsigned char)padobran[i] << endl;
```

Dodjela tipa `(int)` neophodna je da bi se ispisao ASCII kôd pojedinog znaka, dok je dodjela tipa `(unsigned char)` dodana radi pravilnog ispisa kôdova hrvatskih dijakritièkih znakova. Izvoðenjem, dobit æemo sljedeæi ispis (kôdovi za hrvatska slova mogu se razlikovati, ovisno o korištenom skupu znakova):

```
F   70
a   97
u   117
s   115
t   116
    32
V   86
r   114
a   97
n   110
č   232
i   105
ć   230
    0
J   74
u   117
l   108
i   105
j   106
e   101
```

Pažljiv èitatelj je zasigurno primijetio da iako smo ispisivali znakovni niz `padobran`, dobili smo i dio niza `minijature`. Naime, prevoditelj kojeg smo koristili za izvoðenje gornjeg primjera poredao je u memoriju znakovne nizove u obrnutom redoslijedu od redoslijeda deklaracije. Znakovni niz `minijature` smješten je neposredno iza niza `padobran`. Kako smo u `for`-petlji ispisivali dvadeset znakova, što je više nego što ih ima `padobran`, "zagazili" smo i u niz `minijature`. Ovakvo ponašanje je vrlo ovisno o

prevoditelju: neki drugi prevoditelj æe možda smjestiti podatke na drukèiji naèin te æete možda dobiti drukèiji ispis.

Nul-znak između dva niza umetnuo je prevoditelj da bi funkcijama koje će dohvaćati i obrađivati znakovni niz dao do znanja gdje on završava. Tako će izlazni tok za ispis:

```
cout << padobran << endl;
```

toèno znati da je slovo 'æ' ispred nul-znaka zadnje u nizu, te æe pravilno ispisati sadržaj niza `padobran` samo do tog slova:

```
Faust Vrančić
```

Prepišemo li nul-znak nekim drugim znakom, na primjer naredbom:

```
padobran[13] = '&';
```

gornja naredba za ispis niza `padobran` dat æe:

```
Faust Vrančić&Julije Klović
```

tj. ispisuju se svi znakovi do prvog nul-znaka. Ako bismo prepisali i nul-znak koji zakljuèuje drugi niz, rezultat ispisa bi bio nepredvidiv.

Na osnovi ovog razmatranja slijedi da smo znakovne nizove mogli deklarirati i inicijalizirati identično kako smo to radili i kod jednodimenzionalnih polja, ali uz obaveznu eksplicitnu inicijalizaciju zaključnog nul-znaka:

```
char padobran[] = { 'F', 'a', 'u', 's', 't', ' ',
                    'V', 'r', 'a', 'n', 'č', 'i', 'ć',
                    '\0'
                    };
```

Vjerujemo da èitatelja ne treba posebno upuæivati u nespretnost ovakvog naèina inicijalizacije.

Činjenica da znakovnim nizovima pripada i zaključni nul-znak nalaže dodatni oprez pri inicijalizaciji i rukovanju. Primjerice, rizično je eksplicitno definirati duljinu znakovnog niza:

```
char Stroustrup[3] = "C++";     // nezaključen niz!
```

Ovakvom inicijalizacijom rezerviran je prostor od 3 bajta za znakove. Buduæi da sam niz sadrži 3 znaka, nul-znak æe biti pohranjen u memoriju na prvu lokaciju iza polja `Stroustrup`, te neæe biti zaštiæen od moguæeg prepisivanja pri inicijalizaciji varijable ili polja koje æe prevoditelj smjestiti iza polja `Stroustrup`.

Prilikom alociranja prostora za znakovni niz, neophodno je predvidjeti i prostor za zaključni nul-znak.

Također, prilikom preslikavanja znakovnih nizova, neophodno je preslikati i nul-znak, jer æe u protivnom preslikani niz ostati nezaključen. Sreæom, za veæinu operacija s nizovima postoje gotove funkcije u datoteci `string.h` koje vode raèuna o nul-znaku, pa je najsigurnije njih koristiti.

Budući da se dohvaćanje članova polja u suštini obavlja preko pokazivača, zašto ne bismo deklarirali i inicijalizirali pokazivač:

```
char *Stroustrup = "C++";
```

Naizgled, ova inicijalizacija neæe proæi, jer kada smo sliènu stvar pokušali s brojevima:

```
int *n = 10;                // pogreška
```

prevoditelj je javljao pogrešku. Međutim, kod pokazivaèa na niz situacija je drukèija. Prevoditelj æe osigurati neophodan memorijski prostor za pohranu cijelog znakovnog niza zajedno sa zakljuènim nul-znakom, te æe usmjeriti pokazivaè `Stroustrup` na njegovu poèetnu adresu. Sljedeæa naredba æe prouzroèiti pogrešku prilikom prevođenja:

```
char *Kernighan = 'C';      // pogreška
```

Naime, ovom naredbom deklarira se pokazivaè na znak (a ne znakovni niz) i pokušava ga se usmjeriti na znakovnu konstantu `'C'` (za koju nije predviđen stalan memorijski prostor).

Vjerujemo da je pažljiviji čitatelj već uočio suštinsku razliku između znaka i znakovnog niza, ali ipak nije na odmet još jednom ponoviti:

Znak (`char`) je samostalna cjelina, a znakovne konstante se pišu omeđene znakom ' (jednostruki navodnik). Znakovni nizovi sastoje se od niza znakova i zakljuènog nul-znaka, a nizovne konstante pišu se omeđene znakom " (dvostruki navodnik).

Koliko je važno paziti da li se znak ili znakovni niz omeđuju jednostrukim ili dvostrukim navodnicima najbolje ilustrira sljedeæa (vrlo tipièna) pogreška:

```
char Slovo;
cin >> Slovo;
if (Slovo == "a")     // pogreška: usporedba znaka i niza
    // ...
```

Oèito je da znakovne nizove nije moguæe preslikavati jednostavnim operatorom pridruživanja – kao i kod polja brojeva, preslikavanje se mora obavljati znak po znak. Uz prevoditelje se isporuèuje biblioteka `string.h` koja sadrži osnovne funkcije neophodne za rukovanje znakovnim nizovima. Buduæi da su te funkcije standardizirane i vrlo korisne, opisat æemo ih u poglavlju posveæenom funkcijama.

Kao što se mogu definirati znakovni nizovi složeni od znakova tipa `char`, mogu se definirati i nizovi složeni od znakova tipa `wchar_t` (vidi odjeljak 2.4.10)

```
wchar_t *duplerica = L"abc";
```

6.4.1. Polja znakovnih nizova

Èesto se u programu javljaju srodni znakovni nizovi koje je praktièno pohraniti u jedno polje. Buduæi da je znakovni niz sam po sebi jednodimenzionalno polje znakova, polje znakovnih nizova u stvari æe biti dvodimenzionalno polje. Evo primjera:

```
#include <iostream.h>
int main() {
    char dani[7][12] = {"nedjelja",
                        "ponedjeljak",
                        "utorak",
                        "srijeda",
                        "četvrtak",
                        "petak",
                        "subota"    };
    cout << "Kad će taj " << dani[5] << endl;
    return 0;
}
```

Pažljiviji èitatelj æe odmah uoèiti nedostatke ovakvog pristupa. Prilikom deklaracije dvodimenzionalnog polja potrebno je zadati obje dimenzije: broj dana u tjednu i duljinu naziva dana (uveæanu za zakljuèni nul-znak). Buduæi da imena dana nisu jednake duljine, trebamo se ravnati prema najduljem imenu, dok æe kraæa imena ostavljati neiskorišteni prostor u dvodimenzionalnom polju znakova. (Koliko bi samo život programera bio jednostavniji da su svi nazivi jednakih duljina!)

Čitatelj koji je već dobro upućen u vezu između pokazivača i polja će sam naslutiti pravo rješenje za gornji problem:

```
char *dani[] = {
                "nedjelja",
                "ponedjeljak",
                "utorak",
                "srijeda",
                "četvrtak",
                "petak",
                "subota" };
```

Umjesto dvodimenzionalnog polja, deklarira se polje pokazivaèa, koji se usmjeravaju na poèetke znakovnih nizova s imenima dana. Prevoditelj æe naslagati te nizove jedan za drugim tako da ne ostaju neiskorištene "rupe".

Poteškoće nastaju žele li se neki od članova ili svi članovi niza promijeniti. Najjednostavnije je deklarirati i definirati drugo polje znakovnih nizova, te preusmjeriti pokazivače:

```
char* days[] = {
                "sunday",
                "monday",
                "tuesday",
                "wednesday",
                "thursday",
                "friday",
                "saturday" };
for (int i = 0; i < 7; i++)
    dani[i] = days[i];
```

6.5. Reference

Reference su poseban tip podataka. One djeluju slièno pokazivaèima, te su u suštini drugo ime za objekt odreðenog tipa. Razmotrimo sljedeæi primjer:

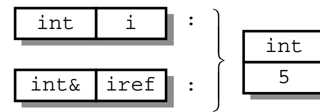
```
#include <iostream.h>

int main() {
    int i = 5;                  // varijabla
    int &iref = i;          // referenca na varijablu i
    cout << "i = " << i << "  iref = " << iref << endl;

    // promjenom reference mijenja se izvorna varijabla
    iref = 75;
    cout << "i = " << i << "  iref = " << iref << endl;
    return 0;
}
```

Na poèetku se deklarira i inicijalizira cjelobrojna varijabla `i`, a potom se deklarira cjelobrojna referenca `iref` na varijablu `i`:

```
int &iref = i;
```

Time se zapravo uvodi novo ime za veæ postojeæu varijablu `i` (vidi sliku 6.15). Promjenom reference mijenja se u stvari izvorna varijabla, u što nas uvjerava ponovni ispis varijabli u gornjem programu.

Nema bitne razlike između pokazivača i reference: oboje omogućavaju posredan pristup nekom objektu. Reference se i implementiraju kao pokazivači. Međutim, pokazivač se može preusmjeriti na neki drugi objekt. Pri dohvaćanju preko pokazivača neophodan je operator `*`, a sam pokazivač iziskuje određeni memorijski prostor u koji će biti pohranjena adresa na koju on pokazuje. Naprotiv, referenca se inicijalizira prilikom deklaracije i pokazuje na dani objekt dok ona postoji, te nije potreban nikakav operator za dohvaćanje sadržaja.



Slika 6.15. Referenca

Očito je da referenca prilikom deklaracije mora biti inicijalizirana – mora već postojati objekt na koji se referenca poziva. Stoga će deklaracija reference bez inicijalizacije izazvati pogrešku prilikom prevođenja:

```
int &nref;       // pogreška
```

Buduæi da postoje konstantni objekti, postoje i reference na nepromjenjive objekte:

```
const float korijen2 = 1.41;
const float &refKor2 = korijen2;
```

`refKor2` je referenca na nepromjenjivi `korijen2`. Zanimljivo je primijetiti da se referenca na `const` objekt može vezati i na promjenjivi objekt:

```
double x;
const double& xref = x;
```

Pokušaj promjene konstantne reference `xref` prevoditelj æe prijaviti kao pogrešku, ali se varijabla `x` može mijenjati bez ikakvih ogranièenja.

Reference najveću primjenu imaju kao argumenti i povratne vrijednosti funkcija, pa ćemo ih u poglavlju o funkcijama pomnije upoznati.

6.5.1. Reference za hakere

Reference se u principu implementiraju kao pokazivaèi koji ne traže operator `*` za pristup sadržaju. Mogli bismo pomisliti da æemo pomoæu sljedeæe naredbe doseæi adresu reference, te direktnim upisivanjem nove adrese promijeniti objekt na koji ona pokazuje:

```
int i, j;
int &ref = j;
*(&ref) = &j;         // bezuspješan pokušaj preusmjeravanja
                      // reference
```

Pri tome haker-hlebinac može pomisliti da æe `&ref` dati adresu na kojoj se zapravo nalazi pokazivaè te `*(&ref)` omoguæava upisivanje adrese varijable `j` na to mjesto. Gornji kôd se neæe uopæe niti prevesti, jer æe prevoditelj dojaviti da ne može pretvoriti `int *` u `int`. O èemu se zapravo radi?

Iako su reference u biti pokazivači, jezik C++ ne dozvoljava izravan pristup adresi reference; što se njega tiče, referenca je samo sinonim za neki drugi objekt. Zbog toga će `&ref` vratiti adresu objekta na koji referenca pokazuje, a ne adresu reference. U to se možemo osvjedočiti sljedećim primjerom:

```
int main() {
    char a;
    char &ref = a;
    cout << "Adresa od a: " << &a << endl;
    cout << "Adresa reference: " << &ref << endl;
    return 0;
}
```

Nakon izvoðenja, dobit æemo ispis u kojemu su obje adrese identiène. Takoðer, `sizeof` operator neæe vratiti velièinu reference (koja je jednaka velièini pokazivaèa), nego æe vratiti velièinu referiranog objekta:

```
int main() {
    char a;
    char &ref = a;
    cout << "Veličina od a: " << sizeof(a) << endl;
    cout << "Veličina reference: " << sizeof(ref) << endl;
    return 0;
}
```

Nakon izvoðena, dobit æemo da je velièina i varijable `a` i reference `ref` jednaka jedan. No referencu se ipak može preusmjeriti prljavim trikovima. Donji program uspješno je preveden i izveden pomoæu Borland C++ 4.5 prevoditelja na Pentium raèunalu te ne garantiramo uspješno izvoðenje na nekoj drugoj kombinaciji prevoditelja i/ili raèunala.

```
int main() {
    char zn1 = 'a',zn2 = 'b', &ref = zn1;
    // Join us on the following page as Dirty Harry strikes...
    char **pok = (char **)(&zn2 - sizeof(char*));
    *pok = &zn2;
    cout << ref << endl;
    return 0;
}
```

Nakon izvoðenja ispisao se znak `'b'`, èime smo se osvjedoèili da `ref` više ne referira varijablu `zn1`, nego `zn2`. Ideja je sljedeæa: prevoditelj sve lokalne varijable smješta u kontinuirano podruèje memorije. Usporedbom adresa varijabli `zn1` i `zn2` ustanovili smo da æe naš prevoditelj prvo smjestiti varijablu `ref`, zatim `zn2`, pa onda `zn1`, (dakle, u

obrnutom redoslijedu od slijeda deklaracije). Zbog toga smo s `&zn2` dobili adresu koju smo umanjili za velièinu pokazivaèa. Rezultat je adresa reference koju smo dodijelili pokazivaèu `pok`. On je nakon dodjele tipa pokazivao na referencu (vidimo da je to pokazivaè na pokazivaè, što je uèinjeno i u deklaraciji). Zatim je sadržaj tog pokazivaèa promijenjen, što je rezultiralo i promjenom sadržaja reference. Na kraju još slijedi slavodobitan ispis. (*Imamo referencu!*)

Gornji program se koristi prljavim trikovima koji izlaze izvan definicija jezika C++ te ovise o implementaciji prevoditelja. Zbog toga pisanje i izvođenje takvih programa provodite na vlastitu odgovornost. (Mi smo u ovom slučaju Poncije Pilati.) Ako takav program i radi na jednoj vrsti računala, vjerojatno neće raditi ako se promijeni prevoditelj ili se program prenese na drugi tip računala.

6.6. Nevolje s pokazivaèima

Pokazivaè se može usmjeriti na bilo koji objekt u memoriji. Međutim, mnogi objekti u programu ima svoj ogranièeni životni vijek, od trenutka kada su deklarirani, do trenutka kada se oni uništavaju. Lokalni objekti se automatski uništavaju na kraju bloka naredbi u kojem su deklarirani. Uništenjem objekta, na mjesto u memoriji koje je on zauzimao prevoditelj æe pohraniti neki drugi objekt. Poteškoæe nastaju ako je pokazivaè ostao usmjeren na tu memorijsku lokaciju, unatoè èinjenici da dotiènog objekta tamo više nema. Ilustrirajmo to sljedeæim primjerom u kojem u ugniježđenom bloku definiramo lokalnu varijablu `j` na koju usmjeravamo pokazivaè `pok`:

```
int main() {
    int *pok;               // pokazivač
    int i = 10;

    {
        int j = 100;        // lokalna varijabla u bloku
        pok = &j;           // pokazivač usmjeravamo na nju
    }

    // ...
    // varijabla j više ne postoji!
    cout << *pok << endl;   // upitan ispis
    return 0;
}
```

Izlaskom iz bloka varijabla `j` prestaje postojati, ali i dalje postoji pokazivaè `pok` koji pokazuje na mjesto gdje je varijabla `j` bila pohranjena. Prevoditelj æe to oslobođeno podruèje vrlo vjerojatno upotrijebiti za pohranu neke druge varijable, pa æe ispis sadržaja preko pokazivaèa davati nepredvidivi rezultat. Još je gora moguænost da preko pokazivaèa `pok` pohranimo neku vrijednost i prepišemo sadržaj novostvorene varijable. Ovakve “mrtve duše” mogu zadavati velike glavobolje programerima buduæi da je takvim pogreškama dosta teško naknadno uæi u trag. Pokazivaèi koji pokazuju na

uništene ili nepostojeæe objekte èesto se nazivaju *viseæim pokazivaèima* (engl. *dangling pointers*) .

Što učiniti s pokazivačem nakon što je objekt na koji je on pokazivao uništen? Ne želimo li uništiti pokazivač (jer nam možda treba kasnije za neki drugi objekt), najjednostavnije je pridružiti mu vrijednost nul-pokazivača:

```
pok = 0;
```

Prilikom dohvaæanja preko pokazivaèa u tom sluèaju æemo prethodno provjeriti je li pokazivaè razlièit od nul-pokazivaèa:

```
//...
if (pok)
    cout << *pok << endl;
else
    cout << "nul-pokazivač!" << endl;
//...
```

Slika 6.16. Viseći pokazivač

6.7. Skraæeno oznaèavanje izvedenih tipova

Kako je u poglavlju o tipovima veæ reèeno, pomoæu kljuène rijeèi `typedef` može se uvesti sinonim za èesto korišteni tip kako bi se programski kôd uèinio èitljivijim. To posebno dolazi do izražaja prilikom definiranja pokazivaèa i referenci na objekte, te prilikom deklaracije polja.

Ako često koristimo tip “pokazivač na znak” - `char *`, možemo uvesti sinonim `pokZnak` koji će označavati gornji tip:

```
typedef char *pokZnak;
```

Znakovne nizove sada možemo deklarirati na sljedeæi naèin:

```
pokZnak uPotrazi = "gDJE mI jE tA pROKLETA cAPSlOCK tIPKA?"
pokZnak indeks;
```

Deklaracija `typedef` se obavlja tako da se iza kljuène rijeèi `typedef` navede obièna deklaracija. Naziv sinonima se umeæe na mjesto gdje bi se inaèe nalazio naziv objekta koji se deklarira: u gornjem primjeru, ako bismo kljuènu rijeè `typedef` izostavili, dobili bismo deklaraciju varijable `pokZnak`. Kako je ispred varijable stavljena kljuèna rijeè `typedef`, deklarira se sinonim `pokZnak` koji oznaèava tip jednak tipu objekta koji bi se deklarirao u sluèaju bez kljuène rijeèi `typedef`. Deklaracija `typedef` se može nalaziti samo u globalnom podruèju imena i, za razliku od deklaracija klasa, ne smije se nalaziti unutar definicije funkcije.

Jednom deklarirani sinonim može se kasnije iskoristiti u drugim deklaracijama sinonima. Recimo, ako radimo s poljem od sto znakovnih nizova, pogodno je uvesti novi tip `poljeZnakovnihNizova` koji će označavati taj tip:

```
typedef pokZnak poljeZnakovnihNizova[100];
```

Ako želimo deklarirati varijablu `telefonskiImenik`, umjesto deklaracije polja od sto pokazivaèa na znak

```
char *telefonskiImenik[100];
```

razumljivije æemo napisati

```
poljeZnakovnihNizova telefonskiImenik;
```

Sada ne moramo razbijati glavu time ima li zvjezdica viši ili niži prioritet od uglatih zagrada, te jesmo li možda zapravo deklarirali pokazivaè na polje od sto znakova.

7. Funkcije

Test prvoklasne inteligencije sastoji se u sposobnosti istovremenog držanja dviju suprotnih ideja u glavi, a da se pritom zadrži sposobnost funkcioniranja.

Scott Fitzgerald (1896-1940)

U ovom poglavlju upoznat æemo se s funkcijama: njihovim znaèenjem i namjenom, razlikom između deklaracije i definicije, tipom funkcija i argumentima. Prouèit æemo što se može funkciji prenijeti kao argument, a što ona može vratiti kao rezultat te kako funkcije djeluju na objekte koji su deklarirani izvan njena tijela. Upoznat æemo znaèenje kljuène rijeèi `inline`, preoptereæenja funkcije i predloška funkcije. Na kraju æemo posebnu pažnju posvetiti funkciji `main()` i standardnim funkcijama.

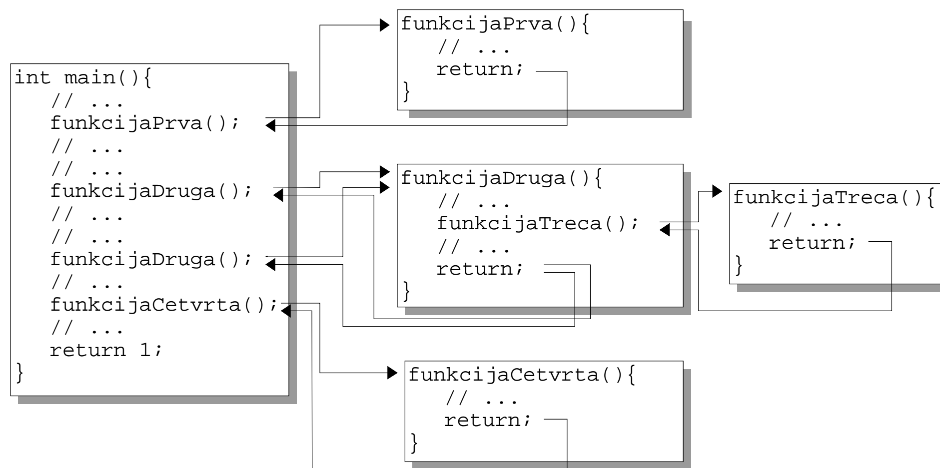
7.1. Što su i zašto koristiti funkcije

U svakom složenijem programu postoje nizovi naredbi koji opisuju neki postupak zajednièki nizu ulaznih vrijednosti. Do sada, ako smo htjeli ponoviti isti postupak s drugim parametrima, morali smo preslikati sve naredbe na sva mjesta gdje se taj postupak koristi. Mane takvog pristupa su oèite: osim što programski kôd postaje glomazan, otežava se i ispravak pogrešaka: primijeti li se pogreška u postupku, ispravak je potrebno unijeti na sva mjesta gdje smo postupak koristili.

Kako bi se takvo ponavljanje izbjeglo, C++ nudi rješenje tako da se ponavljani postupak smjesti u zasebnu cjelinu – funkciju, te se pozove na mjestima gdje je to potrebno. Funkcije se pišu kao zasebni blokovi naredbi prije ili poslije glavne funkcije `main()`. Poziv funkcije se obavlja tako da se izvođenje glavnog kôda trenutno prekine te se nit izvođenja prenese u funkciju. Nakon što se kôda u funkciji izvede, glavni program se nastavlja od sljedeće naredbe iza poziva funkcije (slika 7.1). Gledano izvana, funkcije se ponašaju poput zasebnih cjelina čije unutarnje ustrojstvo ne mora biti poznato korisniku funkcije. Pozivatelj mora znati što neka funkcija čini, no nije mu bitno kako ona to čini. Funkcije imaju *parametre* ili *argumente* (engl. *parameters*, *arguments*) koje pozivatelj mora zadati prilikom poziva, te vraćaju *povratnu vrijednost* (engl. *return value*) pozivatelju čime ga informiraju o rezultatu svoga rada.

Funkcije je moguće pozvati i iz neke druge funkcije (primjerice poziv funkcije `funkcijaTreca()` iz funkcije `funkcijaDruga()` na slici 7.1), pa čak i iz sebe same. Svaka se funkcija može pozivati neograničeni broj puta.

Razbijanjem kôda u funkcije, doprinosi se modularnosti programa – program se cijepa na manje cjeline koje je lakše koristiti. Kôd ujedno postaje čitljiviji i razumljiviji.



Slika 7.1. Izvođenje programa s funkcijama

Ilustrirajmo to vrlo jednostavnim primjerom – programom za računanje binomnih koeficijenata po formuli

$$\binom{p}{r} = \frac{p!}{r!(p-r)!}\,.$$

Bez korištenja funkcija program bi izgledao ovako:

```
#include <iostream.h>

int main() {
    int i, p, r;
    cout << "p = ";
    cin >> p;
    cout << "r = ";
    cin >> r;

    long brojnik = 1;
    for (i = p; i > 1; i--)          // računa p!
        brojnik *= i;
    long nazivnik = 1;
    for (i = r; i > 1; i--)          // računa r!
        nazivnik *= i;
    for (i = p - r; i > 1; i--)      // računa (p-r)!
        nazivnik *= i;

    cout << brojnik / nazivnik << endl;
    return 0;
}
```

Odmah su uoèljive tri vrlo sliène petlje u kojima se raèunaju faktorijele u brojniku, odnosno nazivniku binomnog koeficijenta[†].

Pretpostavimo da imamo na raspolaganju gotovu funkciju `faktorijel()` koja računa faktorijelu broja. U tom slučaju gornji kôd bismo mogli napisati ovako:

```
#include <iostream.h>

int main() {
    int p, r;
    cout << "p = ";
    cin >> p;
    cout << "r = ";
    cin >> r;

    cout << faktorijel(p) / faktorijel(r) /
            faktorijel(p - r) << endl;
    return 0;
}
```

Ovaj kôd je više nego oèito pregledniji i razumljiviji od prethodnoga. Štoviše, funkcije koje napišemo mogu se pohraniti u biblioteke te se zatim ukljuèivati i u programe koje kasnije pišemo, bez potrebe da se kôd tih funkcija ponovo piše ili kopira. Našu funkciju `faktorijel()` možemo tako spremiti u biblioteku `matem`. Poželimo li kasnije napisati program koji treba izraèunavati faktorijele (na primjer za rješavanje zadataka iz vjerojatnosti), jednostavno æemo ukljuèiti biblioteku `matem` u naš projekt. Uz prevoditelje se redovito isporuèuju biblioteke s nizom gotovih funkcija koje obavljaju tipiène zadatke kao što su raèunanje kvadratnog korijena, pristup datotekama ili manipulacija znakovnim nizovima. Veæina tih funkcija je standardizirana, te æe neke od njih biti opisane kasnije u ovom poglavlju.

7.2. Deklaracija i definicija funkcije

Poput varijabli, i funkcije treba prije prvog poziva deklarirati. *Deklaracija* funkcije obznanjuje naziv funkcije, broj i tip parametara te tip njene povratne vrijednosti. Deklaracija ne sadržava opis što i kako funkcija radi, ona daje tzv. *prototip funkcije* (engl. *function prototype*). Deklaracija funkcije ima oblik:

```
<povratni_tip> ime_funkcije ( <tip> arg1, <tip> arg2
);
```

Tip ispred imena funkcije odreðuje kakvog æe tipa biti podatak kojeg æe funkcija vraæati pozivatelju kao rezultat svoga izvoðenja. Argumenti unutar zagrada iza imena

[†] Istina, kôd bi se, kraæenjem dijela umnoška u brojniku i nazivniku, mogao napisati efikasnije, sa samo dvije `for` petlje, ali bi to umanjilo efekt primjera!

funkcije su podaci koji se predaju funkciji prilikom njena poziva. Broj i tip tih argumenata može biti proizvoljan, s time da funkcija može biti i bez argumenata, a mogu se deklarirati i funkcije s neodređenim brojem argumenata. Broj argumenata, njihov redoslijed i tip nazivaju se *potpisom funkcije* (engl. *function signature*).

Kao jednostavan primjer uzmimo funkciju koja računa kvadrat broja. Nazovimo tu funkciju `kvadrat`. Argument te funkcije bit će broj `x` koji treba kvadrirati; uzmimo da je taj argument tipa `float`. Funkcija će kao rezultat vraćati kvadrat broja; pretpostavimo da je to broj tipa `double`. Deklaracija funkcije `kvadrat()` izgledat će u tom slučaju ovako:

```
double kvadrat(float x);
```

Ovo je samo *prototip* funkcije. Za sada nismo definirali funkciju – nismo napisali kôd same funkcije, odnosno nismo odredili što i kako funkcija radi. *Definicija funkcije* (engl. *function definition*) `kvadrat()` izgledala bi:

```
double kvadrat(float x) {
    return x * x;
}
```

Naredbe koje se izvode prilikom poziva funkcije èine *tijelo funkcije* (engl. *function body*). Tijelo funkcije uvijek poèinje prvom naredbom iza lijeve vitièaste zagrade `{`, a završava pripadajuæom desnom zagradom `}`. U gornjem primjeru tijelo funkcije sastoji se samo od jedne jedine naredbe koja kvadrira `x`. Kljuèna rijeè `return` upuæuje na to da se umnožak `x * x` prenosi kao povratna vrijednost u dio programa koji je pozvao funkciju.

Deklaracija i definicija funkcije mogu biti razdvojene i smještene u potpuno različitim dijelovima izvornog kôda. Deklaracija se mora navesti u svakom programskom odsječku gdje se funkcija poziva, prije prvog poziva. To je potrebno kako bi prevoditelj znao generirati strojni kôd za poziv funkcije (taj kôd ovisi o potpisu funkcije). Definicija funkcije, se naprotiv, smješta u samo jedan dio kôda. Svi pozivi te funkcije će se prilikom povezivanja programa usmjeriti na tu definiciju.

Funkcija mora biti deklarirana u izvornom kôdu prije nego što se prvi puta pozove. Definicija funkcije oblikom mora u potpunosti odgovarati deklaraciji.

Ako se definicija i deklaracija razlikuju, prevoditelj æe javiti pogrešku. Moraju se poklapati tip funkcije te broj, redoslijed i tip argumenata.

Imena argumenata u deklaraciji i definiciji funkcije mogu se razlikovati. Štoviše, u deklaraciji funkcije imena argumenata mogu se izostaviti, što znači da smo gornju deklaraciju mogli napisati i kao:

```
double kvadrat(float);
```

Nakon što je funkcija deklarirana (i eventualno definirana), možemo ju pozivati iz bilo kojeg dijela programa tako da navedemo naziv funkcije te unutar zagrada `()` specificiramo parametre – možemo reæi da na naziv funkcije primjenjujemo operator `()`. Pogledajmo kako bi izgledao program za ispis tablice kvadrata brojeva u èijem se glavnom dijelu poziva funkcija `kvadrat()`:

```
#include <iostream.h>
#include <iomanip.h>

double kvadrat(float);      // deklaracija funkcije

int main() {
    for(int i = 1; i <= 10; i++)
        cout << setw(5) << i
             << setw(10) << kvadrat(i) << endl;
    return 0;
}

double kvadrat(float x) {   // definicija funkcije
    return x * x;
}
```

Uoèimo u gornjem kôdu razliku između argumenta u pozivu funkcije i argumenta u definiciji funkcije. Argument `x` u definiciji je *formalni argument* (engl. *formal argument*), simbolièko ime kojim prevoditelj barata tijekom prevođenja tijela funkcije. Taj identifikator je dohvatljiv samo unutar funkcije, dok za kôd izvan nje nije vidljiv. Kada se program izvodi, pri pozivu funkcije se formalni argument inicijalizira *stvarnim argumentom* (engl. *actual argument*), tj. konkretnom vrijednošæu – u gornjem primjeru je to vrijednost varijable `i`. Imena formalnog i stvarnog argumenta ne moraju biti jednaka. Èak, kao što je i sluèaj u primjeru, ne moraju biti jednakog tipa – uobièajenim pravilima konverzije stvarni argument æe biti sveden na tip formalnog argumenta. Naravno da u pozivu funkcije stvarni argument može biti i brojèana konstanta.

Broj argumenata u pozivu funkcije mora biti jednak broju argumenata u definiciji funkcije, a tipovi stvarnih argumenata u pozivu moraju se podudarati s tipovima odgovarajuæih formalnih argumenata u definiciji, ili se moraju (ugrađenim ili korisnièki definiranim pravilima konverzije) dati svesti na tipove formalnih argumenata.

U protivnom æe prevoditelj javiti pogrešku. Na primjer, na pokušaj poziva gore definirane funkcije `kvadrat()` naredbom:

```
kvadrat(4, 3);          // pogreška: argument viška
```

prevoditelj æe javiti pogrešku: “Suvišni parametar u pozivu funkcije kvadrat(float) …”. Ovakva stroga sintaksna provjera poziva funkcija može zasmetati poèetnika. Međutim, ona osigurava ispravno prevođenje i rad programa: prevoditelj æe uoèiti pogrešku u

kôdu, ispisati poruku o pogrešnom pozivu funkcije i prekinuti postupak generiranja izvršnog programa. Ako bi prevoditelj, unatoè pogreški dozvolio generiranje izvršnog kôda, nepravilan poziv funkcije bi izazvao nepredvidiv prekid programa tijekom njegovog izvođenja.

Stroga sintaksna provjera izvornog kôda prilikom prevođenja svodi broj *pogrešaka pri izvođenju* (engl. *run-time errors*) na najmanju moguæu mjeru.

Upravo da bi se omoguæila potpuna sintaksna provjera, deklaracija funkcije mora prethoditi prvom pozivu funkcije[†].

Obratimo stoga na trenutak pažnju strukturi prethodnog programa: nakon pretprocesorske naredbe `#include` slijedi deklaracija funkcije `kvadrat()`. Zatim slijedi glavna (`main()`) funkcija unutar koje se poziva funkcija `kvadrat()`, a na kraju se nalazi definicija funkcije `kvadrat()`. Ako bi deklaracija funkcije bila izostavljena, prevoditelj bi tijekom prevođenja izvornog kôda u funkciji `main()` naletio na njemu nepoznatu funkciju `kvadrat()`, te bi prijavio pogrešku.

Druga je mogućnost da se definicija funkcije `kvadrat()` prebaci na početak koda. U definiciji funkcije je implicitno sadržana i njena deklaracija, te će ona tada biti deklarirana prije poziva u funkciji `main()`. Shodno tome, u jednostavnijim programima je dovoljno sve definicije funkcija staviti ispred funkcije `main()`, tj. funkciju `main()` staviti kao zadnju funkciju u kôdu. Međutim, situacija postaje zamršena ako se funkcije međusobno pozivaju, jedna iz druge.

Svaka definicija je ujedno i deklaracija. Nakon definicije nije dozvoljeno ponavljati deklaracije.

7.2.1. Deklaracije funkcija u datotekama zaglavlja

U složenijim programima se izvorni kôd obièno razbija u više razlièitih datoteka – *modula*. Osnovni motiv za to je brže prevođenje i povezivanje programa: promijenimo li kôd neke funkcije, bit æe potrebno ponovno prevesti samo modul u kojem je dotièna funkcija definirana. Prevođenjem se stvara novi objektni kôd promijenjenog modula, koji se potom povezuje s veæ postojeæim objektnim kôdovima ostalih modula.

Budući da se u tako raščlanjenim programima lako može dogoditi da neka funkcija bude pozvana iz različitih modula, neophodno je deklaracije pozivanih funkcija učiniti dostupnima iz bilo kojeg modula. Stoga se deklaracije funkcija stavljaju u zasebne *datoteke zaglavlja* (engl. *header files*), koje se zatim pretprocesorskom naredbom

---

[†] Zanimljivo je spomenuti da je ovaj princip prvo uveden u jezik C++, da bi tek potom bio prihvaæen u jeziku C i ukljuèen u ANSI C standard [Stroustrup94].

`#include` uključuju u sve datoteke izvornog kôda u kojima se neka od deklariranih funkcija poziva.

Prikažimo ovakvu organizaciju kôda na primjeru apstraktnog programa s dvije funkcije (nazvat ćemo ih `funkcijaPrva()` i `funkcijaDruga()`) koje se pozivaju iz glavne (`main()`) funkcije, a uz to se jedna funkcija poziva iz druge funkcije. Prvo ćemo napisati kôd kako bi izgledao smješten u jednu datoteku:

```
void funkcijaPrva();        // deklaracije funkcija
void funkcijaDruga();

int main() {
    // ...
    funkcijaPrva();
    // ...
    funkcijaDruga();
}

funkcijaPrva(){             // definicija funkcije
    // ...
}

funkcijaDruga(){            // definicija funkcije
    // ...
    funkcijaPrva();
    // ...
}
```

Sada æemo taj kôd razmjestiti u tri odvojena modula, koje æemo nazvati `poglavni.cpp`, `funk1.cpp` i `funk2.cpp`; u prvi modul æemo smjestiti glavnu (`main()`) funkciju, a u potonje module definicije funkcija. Pogledajmo sadržaje pojedinih modula:

poglavni.cpp

```
#include "funk1.h"      // uključuje deklaracije funkcija
#include "funk2.h"

int main() {
    // ...
    funkcijaPrva();
    // ...
    funkcijaDruga();
}
```

`funk1.cpp`

```
void funkcijaPrva(){            // definicija funkcije
    // ...
}
```

`funk2.cpp`

```
#include "funk1.h"

void funkcijaDruga(){           // definicija funkcije
    // ...
    funkcijaPrva();
    // ...
}
```

U gornjim kôdovima važno je uoèiti pretprocesorske naredbe za ukljuèivanje datoteka zaglavlja `funk1.h`, odnosno `funk2.h`, u kojima æemo deklarirati funkcije definirane u `funk1.cpp`, odnosno `funk2.cpp`:

`funk1.h`

```
extern void funkcijaPrva();     // deklaracija funkcije
```

`funk2.h`

```
extern void funkcijaDruga();    // deklaracija funkcije
```

Ova ukljuèenja datoteka zaglavlja su neophodna zbog zahtjeva da deklaracija funkcije mora prevoditelju biti poznata prije njenog prvog pozivanja. Buduæi da se u glavnoj funkciji prozivaju obje funkcije, na poèetku datoteke `poglavni.cpp` morali smo ukljuèiti obje deklaracije. Takoðer, kako se `funkcijaPrva()` poziva iz `funkcijaDruga()`, na poèetku datoteke `funk2.cpp` morali smo ukljuèiti deklaraciju funkcije `funkcijaPrva()`. Ako se u neku datoteku i ukljuèi neka suvišna datoteka zaglavlja, ništa bitno se neæe dogoditi (osim što æe se program prevoditi nešto dulje). Kako datoteke zaglavlja u principu ne sadrže definicije, nego samo deklaracije, neæe doæi do generiranja nepotrebnog kôda.

Uočimo da su imena datoteka zaglavlja umjesto unutar znakova `< >` (manje od - veće od), navedena unutar dvostrukih navodnika `" "`. To je naputak procesoru da te datoteke prvo treba tražiti u tekućem imeniku (direktoriju). Ako ih tamo ne pronađe, pretražuju se imenici koji su definirani u prevoditelju, prilikom njegova instaliranja.

Ključna riječ `extern` ispred deklaracija označava da su funkcije definirane u nekoj drugoj datoteci – ona nije neophodna za pravilno prevođenje zaglavlja.

Vrlo je važno dobro organizirati datoteke prilikom razvoja složenih programa. Pravilnim pristupom problemu organizacije kôda može se uštediti velika količina vremena (te živaca, kave i neprospavanih noći) prilikom razvoja. Organizacija se dodatno komplicira uvođenjem klasa te predložaka funkcija i klasa. Zbog tih razloga, organizaciji kôda bit će posvećeno zasebno poglavlje 14.

7.3. Tip funkcije

Tip funkcije odreðuje kakvog æe tipa biti podatak koji funkcija vraæa pozivajuæem kôdu. Tip se funkciji pridjeljuje prilikom deklaracije, tako da se ispred imena funkcije navede identifikator tipa, na primjer:

```
double kvadrat(float x);
float kvadratniKorijen(float x);
char *gdjeSiMojaNevidjenaLjubavi();
```

Funkcija `kvadrat()` je tipa `double`, funkcija `kvadratniKorijen()` je tipa `float`, a `gdjeSiMojaNevidjenaLjubavi()` je tipa `char *` (ona vraæa pokazivaè na znak). Funkcija može opæenito biti i korisnièki definiranog tipa. Konkretnu vrijednost koju funkcija vraæa odreðuje se pomoæu naredbe `return` u definiciji funkcije, što možemo ilustrirati jednostavnim primjerom, funkcijom `apsolutno()`:

```
float apsolutno(float x) {
    return (x >= 0) ? x : -x;
}
```

Uvjetni operator radi na sljedeæi naèin: ako je argument `x` veæi ili jednak nuli, rezultat operatora je jednak vrijednosti argumenta, a u protivnom sluèaju rezultat je argument s promijenjenim predznakom (odnosno apsolutna vrijednost). Naredbom `return` se rezultat operatora proglašava za povratnu vrijednost, izvoðenje funkcije se prekida te se vrijednost vraæa pozivajuæem programu. Parametar naredbe `return` opæenito može biti bilo kakav broj, varijabla ili izraz koji se izraèunava prije završetka funkcije. Pri tome tip rezultata izraza mora odgovarati tipu funkcije.

Ako je rezultat izraza naredbe `return` različitog tipa od tipa funkcije, rezultat se (ugrađenim ili korisnički definiranim) pravilima konverzije svodi na tip funkcije. Stoga će funkcija `apsolutno()` deklarirana kao:

```
double apsolutno(float x) {
    return (x >= 0) ? x : -x;
}
```

vraæati rezultat tipa `double`, unatoè tome što je argument, odnosno rezultat `return` naredbe tipa `float` Naravno da nema previše smisla ovako definirati funkciju tipa

`double` buduæi da se pretvorbom rezultata u naredbi `return` ne dobiva na toènosti povratne vrijednosti. Meðutim, ima smisla definirati funkciju:

```
long apsolutnoCijelo(float x) {
    return (x >= 0) ? x : -x;
}
```

koja æe (ako argument nije prevelik) vraæati cjelobrojni dio argumenta. Rezultat uvjetnog pridruživanja je opet `float`, ali kako je funkcija deklarirana kao `long`, pozivajuæem kôdu æe biti vraæen samo cjelobrojni dio argumenta.

Funkcija kao rezultat može vraćati podatke bilo kojeg tipa, izuzev polja i funkcija (iako može vraćati pokazivače i reference na takve tipove).

U pozivajuæem kôdu rezultat funkcije može biti dio proizvoljnog izraza. Konkretno, prije definiranu funkciju `kvadrat()` možemo koristiti u aritmetièkim izrazima s desne strane operatora pridruživanja, tretirajuæi ju kao svaki drugi `double` podatak. Na primjer:

```
float xNaPetu = kvadrat(x) * kvadrat(x) * x;
double cNaKvadrat = kvadrat(a) + kvadrat(b);
```

Moguæe je èak poziv funkcije smjestiti kao argument poziva funkcije:

```
float xNa4 = kvadrat(kvadrat(x));
```

Rezultat funkcije se može i ignorirati. Funkciju `kvadrat()` smo mogli pozvati i ovako:

```
kvadrat(5);
```

Iako je smislenost gornje naredbe upitna, poziv je sasvim dozvoljen. Ako funkcija ne treba vratiti vrijednost, to se može eksplicitno naznaèiti tako da se deklarira tipom `void`. Obièno su to funkcije za ispis poruka ili rezultata ovisnih o vrijednostima argumenata. Na primjer, zatreba li nam funkcija koja æe samo ispisivati kvadrat broja, a sam kvadrat neæe vraæati pozivnom kôdu, deklarirat æemo i definirati funkciju:

```
void ispisiKvadrat(float x) {
    cout << (x * x) << endl;
    return;
}
```

Zadatak. *Napišite program za određivanja dana u tjednu korištenjem dviju funkcija čije su deklaracije:*

```
int nadnevakUbroj(int dan, int mj, long LjetoGospodnje);
void danUtjednu(int);
```

Prva funkcija neka pretvara zadani datum u cijeli broj (između 0 i 6), a druga funkcija shodno tom broju neka ispisuje tekst naziva dana u tjednu.

Buduæi da funkcija tipa `void` ništa ne vraæa pozivnom kôdu, naredba `return` ne smije sadržavati nikakav podatak. Štoviše, u ovakvim sluèajevima se naredba `return` može i izostaviti.

> Naredba `return` je obavezna za izlazak iz funkcija koje vraæaju neki rezultat. Za funkcije tipa `void` naredba `return` se može izostaviti – u tom sluèaju prevoditelj æe shvatiti da je izlaz iz funkcije na kraju njene definicije.

Funkcija tipa `void` ne može se koristiti u aritmetièkim operacijama, pa æemo gore definiranu funkciju `ispisiKvadrat()` stoga uvijek pozivati na sljedeæi naèin:

```
ispisiKvadrat(10);
```

Niže navedeni pokušaj pridruživanja prouzroèit æe pogrešku prilikom prevođenja:

```
float a = ispisiKvadrat(10);    // pogreška
```

Povratak iz funkcije je moguæ s bilo kojeg mjesta unutar tijela funkcije. Stoga se naredba `return` može pojavljivati i na više mjesta. Takav primjer imamo u sljedeæoj funkciji:

```
void lakonski(int odgovor) {
    switch (odgovor) {
      case 0:
        cout << "Ne";
        return;                         // 1. return
      case 1:
        cout << "Da";
        return;                         // 2. return
    }
    cout << "Nekada sam bio tako neodlučan,"
            " a možda i nisam?!";
}                                       // 3. return
```

U gornjoj funkciji postoje dvije eksplicitne `return` naredbe, te podrazumijevani `return` na kraju tijela funkcije.

Na kraju, uočimo jednu bitnu razliku kod deklaracija funkcija u programskom jeziku C++ u odnosu na deklaracije u jeziku C: u programskom jeziku C, tip funkcije u deklaraciji/definiciji se smije izostaviti – u tom slučaju prevoditelj pridružuje toj funkciji podrazumijevani tip `int`. Zato sve funkcije u C kôdu ne moraju biti deklarirane prije prvog poziva. Naleti li C-prevoditelj na poziv nedeklarirane funkcije, on će

pretpostaviti da je ta funkcija tipa `int`. Naprotiv, u programskom jeziku C++ tip funkcije je obavezan i izostanak tipa u deklaraciji, odnosno nailazak na nedeklariranu funkciju rezultira pogreškom tijekom prevođenja.

7.4. Lista argumenata

Argumenti funkcije su podaci koji se predaju funkciji da ih ona obradi na odgovarajuæi naèin postupkom odreðenim u definiciji funkcije. Argument funkcije može biti bilo koji ugraðeni ili korisnièki definirani tip podatka (objekta), odnosno pokazivaè ili referenca na neki takav objekt. Tip `void` se ne može pojaviti u listi argumenata funkcije (buduæi da taj tip ne definira nikakvu konkretnu vrijednost). Dozvoljeno je prosljeðivanje pokazivaèa na tip `void`.

7.4.1. Funkcije bez argumenata

Neke funkcije za svoj rad ne iziskuju argumente – takve funkcije imaju praznu listu argumenata, tj. unutar zagrada se u deklaraciji ne navode argumenti. Tako smo u dosadašnjim primjerima funkciju `main()` deklarirali bez argumenata:

```
int main() {
    // ...
    return 0;
}
```

ANSI standard za programski jezik C zahtijeva da se unutar zagrada kod deklaracije funkcije bez argumenata navede kljuèna rijeè `void`[†]:

```
int main(void) {
    // ...
}
```

Zbog kompatibilnosti, u jeziku C++ je dozvoljen i ovakav zapis. Meðutim, dosljedno gledano kljuèna rijeè `void` je potpuno suvišna, jer bi upuæivala da se kao argument pojavljuje nekakav podatak tipa `void`.

Funkcije bez argumenata se obično koriste za ispis poruka ili za ispis podataka čiji rezultat ovisi isključivo o kôdu unutar same funkcije.

7.4.2. Prijenos argumenata po vrijednosti

Uobièajeni naèin prijenosa argumenta u funkcije jest prijenos vrijednosti podatka, kao što smo to veæ radili s funkcijama `faktorijel()`, odnosno `kvadrat()` u ovom

---

[†] Ovo je posljedica nedosljednosti nastalih tijekom razvoja programskog jezika C. Naime, u izvornoj varijanti jezika C, argumenti funkcije nisu se deklarirali unutar zagrada, veæ iza liste parametara, a ispred tijela funkcije.

poglavlju. Međutim, vrlo je važno uoèiti da prilikom takvog poziva funkcije formalni argument i vrijednost koja se prenosi nisu međusobno povezani. Formalni argument "živi" samo unutar funkcije te je njegova vrijednost po izlasku iz funkcije izgubljena. Ako u funkciji mijenjamo vrijednost argumenta, promjena se neæe odraziti na objekt koji smo naveli u listi prilikom poziva. Pogledajmo na jednom banalnom primjeru kakve to ima praktiène posljedice:

```
#include <iostream.h>

int DodajSto(float i) {
    i += 100;
    return i;
}

int main() {
    int n = 1;
    DodajSto(n);
    cout << "Radio " << n << endl;      // ispisuje 1
    n = DodajSto(n);
    cout << "Radio " << n << endl;      // ispisuje 101
    return 0;
}
```

Iako funkcija `DodajSto()` u svojoj definiciji uveæava vrijednost argumenta, ona barata samo s lokalnom varijablom koja se prilikom poziva inicijalizira na vrijednost stvarnog argumenta. Pojednostavljeno reèeno, funkcija je napravila kopiju argumenta te cijelo vrijeme radi s njom. Prilikom izlaska se kopija uništava, jer je ona definirana samo unutar funkcijskog bloka. Zbog toga æe nakon prvog poziva funkcije varijabla `n` i dalje imati istu vrijednost kao i prije poziva. Ovakav prijenos vrijednosti funkciji naziva se *prijenos po vrijednosti* (engl. *pass by value*).

Kako se uvećana vrijednost vraća kao rezultat funkcije, tek nakon drugog poziva funkcije, tj. pridruživanja povratne vrijednosti varijabli `n`, varijabla `n` će doista biti uvećana.

Čitatelju naviknutom na osobine nekih drugih programskih jezika (npr. BASIC, FORTRAN) činit će se ovakvo ponašanje argumenata funkcije vrlo nespretnim i neshvatljivim. Međutim, ovakav pristup ima jednu veliku odliku: zaštitu podataka u pozivajućem kôdu. Radi ilustracije, pretpostavimo da se prilikom izvođenja funkcije zaista mijenja i vrijednost stvarnog argumenta (nazovimo to *BASIC-pristup*). Uz takvu pretpostavku, gornji program bi ispisao brojeve 101 i 201. Razmotrimo sada kakve bi posljedice u takvom slučaju imala naizgled banalna promjena definicije funkcije `DodajSto()`. Na primjer, definiciju funkcije će netko napisati kraće kao:

```
int DodajSto(int i) {
    return i + 100;
}
```

Ovime se smisao funkcije nije promijenio – ona i nadalje kao rezultat vraæa broj jednak argumentu uveæanom za 100. Međutim, u tijelu funkcije se vrijednost formalnog argumenta više ne mijenja, tako da bi nakon ovakve preinake uz *BASIC-pristup* konaèni ishod programa bio drukèiji. Naprotiv, ispis uz *ne-BASIC-pristup* æe biti uvijek isti, štogod mi radili s argumentom unutar funkcije, uz pretpostavku da je povratna vrijednost u naredbi `return` pravilno definirana.

Ovakvim pristupom u jeziku C++ u velikoj su mjeri izbjegnute *popratne pojave* (engl. *side-effects*) koje mogu dovesti do neželjenih rezultata, budući da je količina podataka koji se izmjenjuju između pozivnog kôda i funkcije svedena na najmanju moguću mjeru: na argumente i povratnu vrijednost.

Kada se funkciji argumenti prenose po vrijednosti, tada se njihova vrijednost u pozivajuæem kôdu ne mijenja.

Ako su stvarni argument i formalni argument u deklaraciji razlièitih tipova, tada se prilikom inicijalizacije formalnog argumenta na stvarni argument primjenjuju uobièajena pravila konverzije, navedena u poglavlju 2.

Kao stvarni argument funkcije može se upotrijebiti i neki izraz. U takvim slučajevima se izraz izračunava prije poziva same funkcije. Tako bismo tablicu kvadrata brojeva mogli ispisati i pomoću sljedećeg kôda:

```
int i = 0;
while (i < 10) cout << kvadrat(++i) << endl;
```

Međutim, pri pisanju takvih izraza valja biti oprezan:

Kod funkcija s dva ili više argumenata, redoslijed izraèunavanja argumenata nije definiran standardom.

Ilustrirajmo to pomoæu funkcije `pow()` za raèunanje potencije x^y, koja je deklarirana u `math.h` kao:

```
double pow(double x, double y);
```

Gornja èinjenica prouzroèit æe da vrlo vjerojatno kôd:

```
int n = 2;
cout << pow(++n, n) << endl;    // promjenjivi rezultat!
```

preveden na nekom prevoditelju ispisuje kao rezultat 27 (rezultat potenciranja: 3^3), a na nekom drugom prevoditelju 8 (2^3), ovisno o tome da li se prvo izraèunava vrijednost prvog ili drugog parametra.

Pri korištenju izraza u pozivima funkcija s više argumenata treba biti umjeren. Daleko je sigurnije te izraze izvuæi ispred poziva funkcije.

Ako prethodni primjer drukèije napišemo, dobit æemo kôd koji æe imati isti rezultat neovisan o implementaciji pojedinog prevoditelja:

```
int n = 2;
++n;
cout << pow(n, n) << endl;      // ispisuje 27
```

7.4.3. Pokazivaè i referenca kao argument

Ponekad se funkcija ne može implementirati korištenjem prijenosa po vrijednosti. Primjerice, pokušajmo napisati funkciju koja æe zamijeniti vrijednosti dviju varijabli. Nepromišljeni programospisatelj bi mogao pokušati taj problem riješiti na sljedeæi naèin:

```
void zamijeni(int prvi, int drugi){         // pogrešno
    int segrt;          // pomoćna varijabla

    segrt = prvi;
    prvi = drugi;
    drugi = segrt;
}
```

Prilikom poziva funkcije `zamijeni(a, b)`, unutar funkcije vrijednosti varijabli `prvi` i `drugi` æe biti zamijenjene. Dakle, algoritam je suštinski korektan. Međutim, po izlasku iz funkcije ta zamjena nema nikakvog efekta, jer je funkcija baratala s preslikama vrijednosti varijabli `a` i `b`, a ne s izvornicima.

Jedno moguće rješenje je upotreba pokazivača prilikom poziva funkcije. Umjesto vrijednosti, funkciji ćemo proslijediti pokazivače na objekte. Tada ćemo funkciju definirati ovako:

```
void zamijeniPok(int *prvi, int *drugi) {
    int segrt = *prvi;
    *prvi = *drugi;
    *drugi = segrt;
}
```

Prilikom poziva, umjesto stvarnih vrijednosti `a` i `b` proslijedit æemo adrese tih objekata:

```
zamijeniPok(&a, &b);
```

Pokazivaèi se, doduše, prenose po vrijednosti, što znaèi da æe formalni argumenti `prvi` i `drugi` sadržavati kopije pokazivaèa. Meðutim, te kopije i dalje pokazuju na iste lokacije na koje su pokazivali stvarni argumenti. Funkcija je napisana tako da obraðuje vrijednosti preko pokazivaèa (zbog toga imamo operator `*` ispred argumenata `prvi` i `drugi`) – to zapravo znaèi da æe se zamjena provesti na lokacijama na koje `prvi` i `drugi` pokazuju. Nakon završetka funkcije `prvi` i `drugi` “umiru”, no to nam nije bitno, jer se promjena odvijala (i odrazila) na objektima `a` i `b` u pozivajuæem kôdu.

Ovakav način prijenosa argumenata postoji u jeziku C, te je to i jedini način promjene vrijednosti u pozivajućem kôdu. Jezik C++ nudi još elegantnije rješenje: *prijenos po referenci* (engl. *pass by reference*) . Umjesto prijenosa pokazivača, u funkciju `zamijeni()` prenijet ćemo reference na vanjske objekte:

```
void zamijeniRef(int &prvi, int &drugi) {
    int segrt = prvi;
    prvi = drugi;
    drugi = segrt;
}
```

Poziv funkcije sada nije potrebno komplicirati operatorom `&`, veæ je dovoljno napisati:

```
zamijeniRef(a, b);
```

Suštinski gledano, nema razlike izmeðu pristupa preko pokazivaèa ili referenci: rezultat je isti. Reference su u biti pokazivaèi koje nije potrebno dereferencirati prilikom korištenja pa se mehanizam prenošenja se u jednom i u drugom sluèaju na razini generiranog strojnog kôda odvija preko pokazivaèa. Meðutim, veæ je na prvi pogled uoèljiva veæa jednostavnost kôda ako koristimo reference. To se odnosi na definiciju funkcije gdje nije potrebno koristiti operator dereferenciranja `*`, a naroèito na poziv funkcije, jer ne treba navoditi operator adrese `&`. Zato èitatelju najtoplije preporuèujemo korištenje referenci. Posebno se to odnosi na *C-gurue* naviknute iskljuèivo na pokazivaèe, buduæi da u programskom jeziku C reference ne postoje kao tip podataka.

Podsjetimo se da se pokazivači i reference na različite tipove podataka ne mogu pridruživati. Pokušamo li funkciju `zamijeniPok()` pozvati tako da joj prenesemo kao argument pokazivač na nešto što nije `int`, prevoditelj će javiti pogrešku prilikom prevođenja:

```
float a = 10.;
int b = 3;
zamijeniPok(*a, *b);        // pogreška: a je float
```

Meðutim, ako funkciji `zamijeniRef()` umjesto reference na `int` prenesemo referencu na neki drugi tip podataka, prevoditelj æe samo uputiti upozorenje:

```
float a = 10.;
int b = 3;
zamijeniRef(a, b);        // oprez!
```

Kako se proslijeðeni tip razlikuje od tipa formalnog argumenta (potrebno je obaviti konverziju `float` u `int`), prevoditelj æe generirati privremenu varijablu tipa `int` u koju æe smjestiti osakaæeni `float`. U funkciju æe se prenijeti adresa te privremene varijable. U funkciji æe biti provedena zamjena vrijednosti izmeðu privremenog objekta i varijable `b`. Pri izlasku iz funkcije privremeni objekt se uništava pa æe nakon gornjeg poziva funkcije obje varijable imati vrijednost 10.

U pozivima funkcija koje imaju pokazivaèe ili reference kao argumente, tipovi pokazivaèa, odnosno referenci koji se prenose moraju toèno odgovarati tipovima u deklaraciji funkcije, jer se za njih ne provode nikakve konverzije.

Vjerojatno je svakom pažljivijem èitatelju jasno da se u pozivima funkcije `zamijeniPok()` i `zamijeniRef()` ne mogu kao argumenti navesti brojèane konstante. To je i logièno ako znamo da nije moguæe odrediti adresu konstante.

Pokazivači i reference se koriste kao argumenti funkcija kada se želi promijeniti vrijednost objekta u pozivajućem kôdu. To je pomalo u suprotnosti s osnovnom idejom funkcije da su argumenti podaci koji ulaze u nju, a povratna vrijednost rezultat funkcije. Mijenjaju li se vrijednosti argumenata, kôd će postati nečitljiviji, jer dolaze do izražaja popratne pojave. Neupućeni čitatelj kôda bit će prisiljen analizirati kôd same funkcije da bi "pohvatao" sve promjene koja funkcija provodi na argumentima.

Uvijek kada je to moguæe, treba izbjegavati reference i pokazivaèe kao argumente funkcija te argumente prenositi po vrijednosti. Time se izbjegavaju popratne pojave.

Upotreba referenci i pokazivaèa je ipak neizbježna kada funkcija treba istovremeno promijeniti dva ili više objekata, ili kada se funkciji moraju prenijeti velike strukture podataka, poput polja (vidi sljedeæi odjeljak 7.4.5). U takvim prilikama, ako funkcija ne mijenja vrijednosti argumenata, korisna je navika deklarirati te argumente nepromjenjivima pomoæu kljuène rijeèi `const`.

7.4.4. Promjena pokazivaèa unutar funkcije

Poèetnici su èesto vrlo zbunjeni kad naiðu na potrebu da unutar funkcije promijene vrijednost nekog pokazivaèa proslijeđenog kao parametar. Na primjer, zamislimo da želimo napisati funkciju `UnesiIme()` koja treba rezervirati memorijski prostor te uèitati ime korisnika. Pri tome je sasvim logièno ime proslijediti kao parametar. No postavlja se pitanje kojeg tipa mora biti taj parametar. Proslježivanje pokazivaèa na znak neæe biti dovoljno:

```
void UnesiIme(char *ime) {       // ovo neće raditi
    ime = new char[100];
    cin >> ime;
}

int main() {
    char *korisnik;
    UnesiIme(korisnik);          // loš poziv
    cout << korisnik << endl;
    delete [] korisnik;
}
```

Na veliko razoèaranje C++ žutokljunaca, gornji primjer neæe ispisati uneseno ime, veæ æe po svoj prilici izbaciti neki nesuvisli niz znakova. U èemu je problem?

Do ključa za razumijevanja gornjeg problema doći ćemo ako se podsjetimo da se parametri u sve funkcije prenose po vrijednosti. To znači da se parametar naveden u pozivu funkcije kopira u privremenu varijablu koja živi isključivo za vrijeme izvođenja funkcije. Parametar možemo slobodno mijenjati unutar funkcije, a promjene se neće odraziti na stvarni parametar naveden u pozivajućem kôdu. Upravo se to događa s našim parametrom `korisnik` – prilikom poziva se vrijednost pokazivača `korisnik` kopira u privremenu varijablu `ime`. Funkcija `UnesiIme()` barata s tom privremenom vrijednosti, a ne sa sadržajem varijable `korisnik`. Adresa memorijskog bloka alociranog operatorom `new` se pridružuje lokalnoj varijabli, te se na to mjesto učitava znakovni niz. Nakon što funkcija završi, vrijednost varijable `ime` se gubi, a varijabla `korisnik` ostaje nepromijenjena.

Rješenje ovog problema nije tako složeno: umjesto prosljeđivanja vrijednosti varijable `korisnik`, potrebno je funkciji proslijediti njenu adresu. Parametar funkcije će sada postati pokazivač na pokazivač na znak: sama varijabla `korisnik` je tipa pokazivač na znak, a njena adresa je pokazivač na pokazivač. Unutar funkcije `UnesiIme()` također treba voditi o tome računa, tako da se vrijednosti pokazivača pristupi pomoću `*ime`. Evo ispravnog programa:

```
void UnesiIme(char **ime) {      // parametar je pokazivač
                                 // na pokazivač
    *ime = new char[100];        // pristup preko pokazivača
    cin >> *ime;
}

int main() {
    char *korisnik;
    UnesiIme(&korisnik);         // prosljeđuje se adresa
    cout << korisnik << endl;
    delete [] korisnik;          // uvijek počistite za sobom!
}
```

Gornji primjer æe sada funkcionirati: prosljeðuje se adresa varijable `korisnik`, te se na taj naèin može promijeniti njen sadržaj. No sam program time postaje neèitljiviji: u

pozivnom kôdu je potrebno uzimati adresu, a u funkciji petljati s operatorom `*`. No reference spašavaju stvar. Naime, pokazivaè na znak je neki tip koji se može proslijediti po vrijednosti, a to znaèi da treba proslijediti referencu na njega. Za one koji su na "Vi" sa sintaksom za deklaraciju tipova u jeziku C++, referenca na pokazivaè na znak se bilježi `char * &` deklaracijom – kljuèno je proèitati deklaraciju s desna na lijevo. Evo kôda koji koristi referencu na pokazivaè:

```
void UnesiIme(char * & ime) {   // parametar je referenca
                                // na pokazivač
    ime = new char[100];        // pristup preko reference
    cin >> ime;
}

int main() {
    char *korisnik;
    UnesiIme(korisnik);         // prosljeđuje se adresa,
                                // ali to nije potrebno
                                // eksplicitno navesti
    cout << korisnik << endl;
    delete [] korisnik;
}
```

7.4.5. Polja kao argumenti

Vidjeli smo da kod prijenosa po vrijednosti promjene argumenata unutar funkcije nemaju odraza na stvarne argumente u pozivajuæem kôdu. Na prvi pogled, izuzetak od tog pravila èine polja. Pogledajmo sljedeæi primjer:

```
void Pocisti(int polje[], int duljina) {
    while (duljina--)
        polje[duljina] = 0;
}

int main() {
    int b[] = {5, 10, 15};
    Pocisti(b, 3);
    cout << b[0] << endl
         << b[1] << endl
         << b[2] << endl;
    return 0;
}
```

Neupuæeni èitatelj bi iz gornjeg kôda mogao zakljuèiti da se polje prenosi po vrijednosti te da æe nakon poziva funkcije `Pocisti()`, èlanovi polja `b[]` ostati nepromijenjeni. Međutim, izvoðenjem programa uvjerit æe se da poziv funkcije `Pocisti()` uzrokuje trajno brisanje svih èlanova polja. Pažljivijem èitatelju æe odmah biti jasno o èemu se radi: polje se u biti ne prenosi po vrijednosti, veæ se prenosi pokazivaè na njegov prvi

èlan. Stoga funkcija preko tog pokazivaèa rukuje s izvornikom, a ne preslikom tog polja. Shodno tome, gornju funkciju smo mogli potpuno ravnopravno deklarirati i na sljedeæi naèin:

```
void Pocisti(int *polje, int duljina);
```

Zadatak. *Napišite definiciju ovako deklarirane funkcije* `Pocisti()` *koristeći aritmetiku s pokazivačima.*

Iako ovakvo ponašanje polja kao argumenta unosi određenu nedosljednost u jezik, ono je posve praktiène naravi. Zamislimo da treba funkciji kao argument prenijeti polje od nekoliko tisuæa èlanova. Kada bi pri pozivu funkcije èlanovi polja inicijalizirali èlanove novog, lokalnog polja u funkciji, proces pozivanja funkcije bi trajao vrlo dugo, ne samo zbog operacija pridruživanja pojedinih èlanova, veæ i zbog vremena potrebnog za dodjeljivanje memorijskog prostora za novo polje. Osim toga, lokalne varijable unutar funkcije se smještaju na stog, posebni dio memorije èija je duljina ogranièena. Smještaj dugaèkih polja na stog vrlo bi ga brzo popunio i onemoguæio daljnji rad programa.

Prilikom deklaracije je moguće, ali nije potrebno navesti duljinu polja (ograničimo se za sada na jednodimenzionalna polja) – prenosi se samo pokazivač na prvi član. Zbog toga, sljedeće tri deklaracije gornje funkcije su iste:

```
void Pocisti(int polje[], int duljina);
void Pocisti(int polje[5], int duljina);
void Pocisti(int *polje, int duljina);
```

Duljina polja je navedena kao dodatni argument funkciji `Pocisti()`, što omoguæava da se funkcija može pozvati za èišæenje polja proizvoljne duljine.

Znakovni nizovi su također polja, tako da za njih vrijede istovjetna razmatranja. To znači da funkciji treba prenijeti pokazivač na prvi član. Kako su znakovni nizovi zaključeni nul-znakom, podatak o duljini niza nije neophodno prenijeti. Na primjer:

```
int DuljinaNiza(char *niz) {
    int i = 0;
    while (*(niz + i))
        i++;
    return i;
}
```

Iako postoji standardna funkcija `strlen()` koja raèuna duljinu znakovnog niza, napisali smo svoju inaèicu takve funkcije. Uoèimo u uvjetu `while` petlje kako je zbog višeg prioriteta operatora `*` bilo neophodno staviti zagrade oko operacije pribrajanja pokazivaèu. Da te zagrade nema, uvjet petlje bi dohvaæao prvi èlan polja, te ono što se tamo nalazi (kôd znaka) uveæao za brojaè `i`.

Problem prijenosa polja funkciji postaje složeniji želi li se prenijeti višedimenzionalno polje. Ako su dimenzije polja zadane u izvornom kôdu, prijenos je trivijalan:

```
#include <iostream.h>
#include <iomanip.h>

const int redaka = 2;
const int stupaca = 3;

void ispisiMatricu(float m[redaka][stupaca]) {
    for (int r = 0; r < redaka; r++) {
        for (int s = 0; s < stupaca; s++)
            cout << setw(10) << m[r][s];
        cout << endl;
    }
}

int main() {
    float matrica[redaka][stupaca] =
                        {   { 1.1, 1.2, 1.3 },
                            { 2.1, 2.2, 2.3 }  };
    ispisiMatricu(matrica);
    return 0;
}
```

Iz poziva funkcije je jasno da se matrica prenosi preko pokazivaèa na prvi èlan. Stoga je navedeni zapis u deklaraciji funkcije ovakav samo radi bolje razumljivosti. Varijable `redaka` i `stupaca` deklarirane su ispred tijela funkcija `ispisiMatricu()` i `main()`, tako da su one dohvatljive iz obiju funkcija (o podruèju dosega imena govorit æemo opširnije u zasebnom poglavlju).

Zadatak. *Napišite funkciju* `ispisiMatricu()` *koristeći pokazivače i aritmetiku s pokazivačima. U funkciju prenesite samo pokazivač na prvi član.*

Vidjeli smo da se višedimenzionalna polja pohranjuju u memoriju kao nizovi jednodimenzionalnih polja. Stoga je prva dimenzija nebitna za pronalaženje odreðenog èlana u polju, pa ju nije obavezno ukljuèiti u deklaraciju parametra:

```
void ispisiMatricu(float m[][stupaca], int redaka) {
    for (int r = 0; r < redaka; r++) {
        for (int s = 0; s < stupaca; s++)
            cout << setw(10) << m[r][s];
        cout << endl;
    }
}
```

Gornja funkcija æe raditi za proizvoljan broj redaka, pa smo podatak o broju redaka prenijeli kao poseban argument. Poteškoæe iskrsavaju ako dimenzije polja nisu zadane u

izvornom kôdu, veæ se odabiru tijekom izvršavanja programa. Sljedeæi pokušaj završit æe debaklom veæ kod prevođenja:

```
void ispisiMatricu(float m[][], int redaka, int stupaca) {
                                              // pogreška
    for (int r = 0; r < redaka; r++) {
        for (int s = 0; s < stupaca; s++)
            cout << setw(10) << m[r][s];
        cout << endl;
    }
}
```

Argument `m[][]` u deklaraciji funkcije nije dozvoljen, jer u trenutku prevođenja mora biti poznata druga dimenzija (za višedimenzionalna polja: sve dimenzije osim prve).

U prethodnom poglavlju smo pokazali kako se članovi dvodimenzionalnog polja mogu dohvaćati preko pokazivača na pokazivače. Iskoristimo tu činjenicu pri pozivu funkcije za ispis članova matrice:

```
#include <iostream.h>
#include <iomanip.h>

void ispisiMatricu(float **m, int redaka, int stupaca) {
    for (int r = 0; r < redaka; r++) {
        for (int s = 0; s < stupaca; s++)
            cout << setw(10) << ((float*)m)[r*stupaca+s];
        cout << endl;
    }
}

int main() {
    int redaka = 2;
    const int stupaca = 3;

    float (*matrica)[stupaca] = new float[redaka][stupaca];
    matrica[0][0] = 1.1;
    matrica[0][1] = 1.2;
    matrica[0][2] = 1.3;
    matrica[1][0] = 2.1;
    matrica[1][1] = 2.2;
    matrica[1][2] = 2.3;
    ispisiMatricu((float**)matrica, redaka, stupaca);
    return 0;
}
```

Oèito je ovakva notacija teško èitljiva. Daleko elegantnije rješenje pružaju korisnièki definirani tipovi – klase, koje se mogu tako definirati da sadrže podatke o pojedinim dimenzijama te da sadrže funkcije za ispis èlanova. Klase æe biti opisane u narednim poglavljima, a za sada uoèimo još samo da je u gornjem kôdu `stupaca` bilo neophodno

deklarirati kao `const`, buduæi da druga dimenzija polja mora biti poznata u trenutku prevoðenja koda.

7.4.6. Konstantni argumenti

Kada se argumenti funkciji prenose po vrijednosti, podaci u pozivajuæem kôdu ostaju zaštiæeni od promjena u funkciji. Meðutim, neki podaci (poput polja) ne mogu se prenijeti po vrijednosti, veæ ih treba prenijeti preko pokazivaèa ili referenci, èime se izlažu opasnosti možebitnih promjena prilikom poziva funkcije. Programer koji æe pisati funkciju æe sasvim sigurno voditi raèuna o tome da ne mijenja vrijednosti argumenata ako to nije potrebno. No lako se može dogoditi da kasnije, prilikom prepravke funkcije smetne tu èinjenicu s uma, ili da netko drugi krene u radikalne zahvate na tijelu funkcije.

Da bi se izbjegle neprilike koje naknadne nepažljive promjene kôda mogu izazvati, preporučljivo je argumente koji se prenose preko pokazivača ili referenci, a koji se ne žele mijenjati, učiniti konstantnima.

Ilustrirajmo to na primjeru funkcije `DuljinaNiza()` koja nam je poslužila za izraèunavanje duljine znakovnog niza. Argument `niz` æe postati pokazivaè na nepromjenivi znakovni niz dodamo li u deklaraciji funkcije modifikator `const` ispred identifikatora tipa:

```
int DuljinaNiza(const char *niz);
```

Svaki pokušaj promjene sadržaja tog niza unutar funkcije prouzroèit æe pogrešku prilikom prevoðenja:

```
int DuljinaNiza(const char *niz) {
    // ...
    *niz = 0;           // pogreška: pokušaj promjene
    return i;           // nepromjenjivog objekta
}
```

Naravno da treba paziti da se kvalifikator `const` navede i u deklaraciji i u definiciji funkcije.

Drugi važan razlog zašto je dobro koristiti kvalifikator `const` kod deklaracije argumenata jest taj da se funkciji mogu uputiti kao argumenti podaci koji su deklarirani kao nepromjenjivi. Da bismo to ilustrirali, deklarirat ćemo gornju funkciju bez kvalifikatora `const` ispred parametra:

```
int DuljinaNiza(char *niz) {
    // ....
}
```

Sada više nije moguæe prosljeðivati kao parametar pokazivaèe na konstantne objekte, jer postoji opasnost da æe se vrijednost konstantnog objekta promijeniti u funkciji:

```
int main() {
    const char* TkoPjeva = "Franjo Šafranek";
    cout << DuljinaNiza(TkoPjeva);          // pogreška
    return 0;
}
```

Èak ako i ne mijenjamo vrijednosti na koje pokazivaè pokazuje unutar funkcije `DuljinaNiza()`, prevoditelj to ne može znati. Iz svoje paranoje on æe sprijeèiti gornji pokušaj pridruživanja nepromjenivog objekta promjenjivom parametru kako bi se osiguralo da objekt u svakom sluèaju ostane nepromijenjen (lat. *object intacta*). Zbog toga ako funkcija ne mijenja svoje parametre, poželjno je to obznaniti javno tako da se umetne modifikator `const` u listu parametara.

Naravno da je obrnuto prosljeđivanje dozvoljeno: promjenjivi objekt slobodno se može pridružiti nepromjenjivom parametru. Time se ne narušava integritet objekta koji se prosljeđuje.

Gornje ponašanje je posljedica standarnih konverzija pokazivača: pokazivač na promjenjivi objekt se uvijek može svesti na pokazivač na nepromjenjivi objekt, dok obrnuto ne ide bez eksplicitne dodjele tipa.

7.4.7. Podrazumijevani argumenti

Postoje funkcije kojima se u veæini poziva prenosi jedna te ista vrijednost argumenta ili argumenata. Na primjer, u funkciji za raèunanje odreðenog integrala trapeznom formulom tipièna relativna pogreška rezultata može biti vrijednost 10^{-4}. Kako bi se izbjeglo suvišno navoðenje željene pogreške u sluèaju da tipièna pogreška zadovoljava, prilikom poziva se pogreška može izostaviti, a u deklaraciji funkcije navesti podrazumijevana vrijednost pogreške.

Na početku ovog poglavlja, u programu za računanje binomnih koeficijenata (vidi kôd na str. 152) pretpostavili smo da imamo na raspolaganju funkciju `faktorijel()`. Bez ikakve optimizacije, binomne koeficijente smo računali doslovno preko formule

$$\binom{p}{r} = \frac{p!}{r!(p-r)!}$$

Svakom imalo bistrijem matematièaru jasno je da se dio umnoška u faktorijeli brojnika može pokratiti s $(p-r)!$ u nazivniku, tako da se ukupni broj množenja u raèunu smanjuje. Na primjer:

$$\binom{5}{2} = \frac{5!}{2!\,3!} = \frac{5\cdot 4\cdot 3\cdot 2\cdot 1}{(2\cdot 1)(3\cdot 2\cdot 1)} = \frac{5\cdot 4}{2\cdot 1}$$

Kako iskoristiti tu èinjenicu u programu za raèunanje binomnih koeficijenata? Mogli bismo napisati dvije funkcije: jednu za raèunanje faktorijele (u nazivniku), drugu za

raèunanje umnoška svih cijelih brojeva između dva zadana (u brojniku). Time æe se ukupni broj množenja u raèunu doista smanjiti, ali æe se poveæati broj funkcija u izvornom kôdu.

Obje tako definirane funkcije u suštini bi provodile isti postupak: množenje uzastopnih cijelih brojeva. Umjesto da definiramo dvije odvojene funkcije, možemo poopćiti funkciju `faktorijel()` tako da proširimo listu argumenata dodatnim parametrom `stojBroj` koji će ograničavati broj množenja. Za uobičajeni faktorijel taj argument bi imao vrijednost 1. Da bismo otklonili potrebu za eksplicitnim navođenjem te vrijednosti kod svakog poziva funkcije, definirat ćemo *podrazumijevanu vrijednost* (engl. *default argument value*) tom dodatnom argumentu:

```
#include <iostream.h>

long faktorijel(int n, int stojBroj = 1) {
    long umnozak = stojBroj;
    while (++stojBroj <= n)
        umnozak *= stojBroj;
    return umnozak;
}
int main() {
    int p, r;
    cout << "p = ";
    cin >> p;
    cout << "r = ";
    cin >> r;
    cout << faktorijel(p, p-r) / faktorijel(r) << endl;
    return 0;
}
```

U pozivu funkcije s oba argumenta, svaki od formalnih argumenata (`n`, odnosno `stojBroj`) æe biti inicijaliziran odgovarajuæim vrijednostima (`p`, odnosno `p - r`). Izostavi li se drugi argument u pozivu, `stojBroj` æe biti inicijaliziran na vrijednost 1, kako je zapisano u deklaraciji funkcije.

Napomenimo da će gornja funkcija `faktorijel()` korektno raditi samo za pozitivne brojeve (uključivo i 0, jer je po definiciji 0!=1), te za slučajeve kada su oba argumenta veća od 0 i kada je prvi argument veći ili jednak drugome.

Zadatak: *Napišite funkciju* `faktorijel()` *koja će provjeravati da li argumenti ispunjavaju ove uvjete, te u protivnom ispisivati poruku o pogreški i vraćati vrijednost −1.*

Oèito je da funkcije s podrazumijevanim argumentom (ili više njih) znaèe moguæe odstupanje od pravila da broj argumenata u pozivu funkcije mora biti jednak broju argumenata u definiciji funkcije. Podrazumijevani argumenti kompliciraju postupak sintaksne provjere koju provodi prevoditelj. Da bi se otklonile sve eventualne nedoumice zbog neslaganja u broju argumenata, prevoditelj mora veæ prilikom nailaska

na poziv funkcije znati da ona može imati podrazumijevane argumente. Zbog toga se podrazumijevane vrijednosti argumenata moraju navesti u deklaraciji funkcije.

Ovo je naravno važno ako su deklaracija i definicija funkcije razdvojene. Treba li u tom slučaju ponoviti podrazumijevanu vrijednost u definiciji funkcije? Ne, jer prevoditelj ne uspoređuje međusobno vrijednosti koje se pridružuju argumentu u deklaraciji, odnosno definiciji, tako da će pokušaj pridruživanja u definiciji interpretirati kao pokušaj promjene podrazumijevane vrijednosti (čak i ako su one jednake):

```
// ...

// deklaracija funkcije:
long faktorijel(int, int = 1);

// ...

// definicija funkcije:
long faktorijel(int n, int stojBroj = 1) {  // pogreška
            // redeklaracija podrazumijevane vrijednosti
// ...
```

Podrazumijevane vrijednosti argumenta navode se u deklaraciji funkcije. Ako su deklaracija i definicija funkcije odvojene, tada se podrazumijevana vrijednost u definiciji funkcije ne smije ponavljati.

```
}
```

Prema tome, u prethodnoj definiciji funkcije treba izostaviti pridruživanje podrazumijevane vrijednosti:

```
// deklaracija funkcije:
long faktorijel(int, int = 1);

// definicija funkcije:
long faktorijel(int n, int stojBroj) {
                        // podrazumjevana vrijednost je
                        // već pridružena u deklaraciji
// ...
}
```

Uoèimo kako je u deklaraciji izostavljeno ime formalnog argumenta kojem se pridružuje podrazumijevana vrijednost. Iako djeluje neobièno, u deklaraciji je to dozvoljeno.

Druga važna stvar o kojoj treba voditi računa pri pridruživanju podrazumijevanih vrijednosti jest:

Argumenti s podrazumijevanom vrijednošæu moraju se nalaziti na kraju liste argumenata funkcije.

To znaèi da ni u kom sluèaju našu funkciju `faktorijel()` ne možemo deklarirati kao:

```
long faktorijel(int stojBroj = 1, int n);   // pogreška
```

jer pridruživanje argumenata pri pozivu funkcije teèe s lijeva na desno, pa æe u pozivu funkcije samo s jednim argumentom drugi argument ostati neinicijaliziran. Ako se u deklaraciji funkcije podrazumijevane vrijednosti pridružuju nekolicini argumenata, tada svi oni moraju biti grupirani na kraj liste parametara:

```
void GoodBadUgly(int, const char* = "Clint",
                 const char* = "Ellie",
                 const char* = "Lee");
```

U ovom primjeru ukazano je i kako treba paziti prilikom pridruživanja pokazivaèa. Praznina izmeðu `*` i `=` je neophodna, jer bi u sluèaju da napišemo

```
void redatelj(const char *= "Sergio");      // pogreška
```

prevoditelj prijavio pogrešku, interpretirajuæi da se radi o operatoru `*=`.

7.4.8. Funkcije s neodreðenim argumentima

Za neke funkcije je nemoguæe unaprijed toèno znati broj i tip argumenata koji æe se proslijediti u pozivu funkcije. Takve funkcije se deklariraju tako da se lista argumenata zakljuèuje s tri toèke (`...`), koje upuæuju da prilikom poziva mogu uslijediti dodatni podaci. Èitatelj koji je ikada vidio ili pisao programe u jeziku C zasigurno poznaje standardiziranu funkciju `printf()` koja se koristi za ispis podataka (u jeziku C ne postoje ulazni i izlazni tokovi `cin`, odnosno `cout`). Ona se može koristi za ispis bilo kojeg broja podataka razlièitih tipova. Funkcija `printf()` u jeziku C++ mogla bi se deklarirati kao:

```
int printf(const char* , ...);
```

Ovime je odreðeno da funkciji `printf()` prilikom njena poziva treba prenijeti barem jedan argument tipa `char *` (pokazivaè na znakovni niz). Ostali argumenti nisu neophodni i mogu biti proizvoljnog tipa. Zarez iza zadnjeg deklariranog argumenta nije neophodan, tako da smo funkciju `printf()` mogli deklarirati i kao:

```
int printf(const char* ...);    // isto, ali bez zareza
```

Ovako deklariranu funkciju možemo pozivati na razne naèine. Na primjer:

```
printf("Za C++ spremni!\n");    // ispis samo poruke
printf("Mercedes %d %s\n", broj, oznaka);
                      // ispis imena, cjelobrojne varijable
                      // 'broj' i znakovnog niza 'oznaka'
```

U posljednjem pozivu funkciji `printf()` prenose se tri argumenta. Prvi argument je znakovni niz koji, uz tekst koji treba ispisati, sadrži i podatke (`%d` i `%s`) o tipu preostala dva argumenta: cjelobrojnom argumentu `broj` i znakovnom nizu `oznaka`. Umjesto simbolièkih imena mogli smo navesti i stvarne vrijednosti:

```
printf("Mercedes %d %s\n", 300, "SE");
```

Buduæi da je jedan argument deklariran, izostavljanje argumenata u pozivu funkcije ili navođenje argumenta koji se ne može svesti na deklarirani tip prouzroèit æe pogrešku prilikom prevođenja:

```
printf();        // pogreška: nedostaje argument
printf(13);      // pogreška: neodgovarajući tip argumenta
```

Prilikom prevođenja prevoditelj ne èita sadržaj poèetnog znakovnog niza u kojem su definirani tipovi podataka koji slijede, tako da ne može provjeriti da li su tipovi podataka koji slijede odgovarajuæe definirani. Na primjer:

```
printf("2 + 2 je %s\n", 2 + 2);     // nepredvidiv rezultat
```

Prevoditelj æe gornju naredbu prevesti bez poruke o pogreški, ali æe izvođenje te naredbe dati nepredvidivi ispis.

Buduæi da za funkcije s neodređenim argumentima prevoditelj ne može provesti provjeru potpisa funkcije, ispravnost poziva u najveæoj mjeri ovisi o autoru kôda.

Kako argument nije deklariran, prevoditelj ne može provjeriti tipove argumenata u pozivu funkcije, a time niti provesti konverzije tipova. Stoga se `char` i `short` implicitno pretvaraju i prenose kao `int`, a `float` se prenosi kao `double` (što èesto nije ono što korisnik oèekuje).

U dobro pisanim programima koristi se zanemariv broj funkcija s neodređenim argumentima. Da bi se izbjegle zamke koje uzrokuju funkcije s neodređenim brojem argumenata i poboljšala provjera tipa, valja koristiti funkcije s podrazumijevanim argumentima i preopterećene funkcije (vidi odjeljak 7.8).

Funkcije s neodređenim argumentima se redovito koriste u deklaracijama funkcija iz biblioteka pisanih u jeziku C, u kojem nije bilo niti podrazumijevanih argumenata niti preopterećenja funkcija.

Za dohvaćanje nespecificiranih argumenata programeru su na raspolaganju makro naredbe iz standardne `stdarg.h` datoteke. Ilustrirajmo njihovu primjenu našom inačicom funkcije `printf()`:

```
#include <iostream.h>
#include <stdarg.h>

int spikaNaSpiku(char *format, ...) {
    va_list vl;                 // podatak tipa va_list
    va_start(vl, format);       // inicijalizira vl

    int i = 0;
    while (*(format + i)) {
        if (*(format + i) == '%') {
            switch (*(format + (++i)++)) {
              case 'd':
              case 'i': {
                int br = va_arg(vl, int);
                cout << br;
                break;
              }
              case 's': {
                char *zn_niz = va_arg(vl, char*);
                cout << zn_niz;
                break;
              }
              case 'c': {
                char zn = va_arg(vl, char);
                cout << zn;
                break;
              }
              default:
                cerr << endl
                     << "Nedefinirani tip podataka! "
                     << endl;
                va_end(vl);
                return 0;
            }
        }
        else
            cout << (*(format + i++));
    };
    va_end(vl);
    return 1;
}
```

Prvo je deklariran objekt `vl` tipa `va_list` koji je definiran unutar datoteke zaglavlja `stdarg.h`. Taj objekt na neki naèin sadrži sve argumente funkcije, te æemo pomoæu njega pristupati pojedinom argumentu (sama struktura objekta korisniku nije važna – on

je “crna kutija”[†]). Prije nego što zapoènemo èitati vrijednosti pojedinih argumenata, objekt `vl` je potrebno inicijalizirati pozivom makro funkcije `va_start()` (takoðer iz `stdarg.h`). Slijedi petlja koja prolazi poèetni znakovni niz, ispituje ga i ispisuje, znak po znak. Nailaskom na znak `%`, ispituje se slijedi li ga neki od znakova: `d`, `i`, `s` ili `c` (popis se može proširiti) – ako nije niti jedan od navedenih znakova, ispisuje poruku o pogreški i prekida izvoðenje funkcije, vraæajuæi pozivnom kôdu nulu. U protivnom, pomoæu makro funkcije `va_arg()` uzima sljedeæi neimenovani argument. Funkciji `va_arg()` je potrebno prenijeti tip parametra koji se želi proèitati, što u našem primjeru nije problem buduæi da ga znamo na osnovu podataka o formatu iz poèetnog niza (ako bi tip i u ovom trenutku bio nepoznat, najjednostavnije bi ga bilo uèitati kao pokazivaè na niz i potom provesti odgovarajuæa ispitivanja). Makro naredba æe vratiti vrijednost argumenta te æe se pomaknuti na sljedeæi argument iz liste. Prilikom sljedeæeg poziva funkcije `va_arg()` dobit æemo vrijednost sljedeæeg argumenta.

Prije povratka iz funkcije nužno je pozvati makro `va_end()`. Naime, funkcija `va_arg()` može modificirati stog (na koji je, između ostalog, pohranjena i povratna adresa), tako da povratak u pozivajući kôd bude onemogućen; `va_end()` uklanja te izmjene na stogu i osigurava pravilan povratak iz funkcije. Prilikom uspješnog povratka, funkcija vraća pozivnom kôdu 1; povratna vrijednost (0 ili 1) iskorištena je ovdje kao pokazatelj uspješnosti izvođenja funkcije.

Uočimo vitičaste zagrade uz pojedine `case` grane kojima su naredbe grupirane u zasebne blokove. One su neophodne, jer u pojedinim granama deklariramo lokalne (privremene) varijable različitih tipova, ovisnih o smjeru grananja, odnosno o tipu podatka koji se očekuje kao sljedeći argument.

Evo kako bi mogao izgledati poziv naše velemožne inačice standardne funkcije `printf()`:

```
spikaNaSpiku("Dalmatinac i %d %s%c", 101, "dalmatine", 'r');
```

Zadatak. *Proširite funkciju `spikaNaSpiku()` tako da prihvaća i sljedeće formate: `%f` za brojeve s pomičnim zarezom, `%x` za ispis brojeva u heksadekadskom formatu, `%o` za ispis brojeva u oktalnom formatu. Također osigurajte pravilan ispis posebnih znakova: `\t`, `\n`, `\\`, `\"`, `\'`.*

7.5. Pokazivaèi i reference kao povratne vrijednosti

Treba li funkcija pozivnom kôdu vratiti neki brojèani rezultat, najjednostavnije je definirati funkciju tako da je onog tipa koji odgovara tipu rezultata, a u `return` naredbi navesti željenu povratnu vrijednost. Na primjer, trebamo li funkciju koja æe na osnovi poznatih odsjeèaka na apscisi i ordinati kao rezultat vraæati koeficijent smjera pravca (tj. tangens kuta), definirat æemo funkciju

[†] Za one koji ipak neæe moæi zaspati dok ne spoznaju što je `va_list` – to je pokazivaè na stog. Makro funkcija `va_start()` inicijalizira taj pokazivaè tako da pokazuje iza zadnjeg prenesenog parametra, dok `va_arg()` preko njega pristupa pojedinom argumentu.

```
double nagib(float x, float y) {
    return y / x;
}
```

Prilikom poziva ove funkcije naredbom

```
float k = nagib(3.2, -12.1);
```

na stogu æe se stvoriti lokalne varijable `x` i `y` kojima æe biti pridružene vrijednosti stvarnih argumenata (3.2, odnosno –12.1). Vrijednosti tih lokalnih varijabli æe se potom podijeliti, a rezultat dijeljenja æe biti pohranjen na stog. Prilikom povratka iz funkcije, pozivajuæi kôd æe taj rezultat skinuti sa stoga i pridružiti ga varijabli `k`. Ovo pridruživanje podrazumijeva da se sadržaj pohranjen na stogu preslikava na mjesto gdje se nalazi sadržaj varijable `k`, uz eventualne konverzije tipa (u gornjem primjeru `double` u `float`).

Nakon povratka u pozivajući kôd svi podaci na stogu koji su privremeno bili stvoreni prilikom izvršavanja funkcije se gube, ali oni nam ionako nisu više bitni budući da smo uspjeli dohvatiti i pokupiti rezultat funkcije. Naravno da ako funkciju `nagib()` pozovemo tako da njen rezultat ne pridružimo odmah nekoj varijabli, rezultat funkcije će ostati bespovratno izgubljen, iako ga je ona izračunala i pohranila na stog:

```
nagib(2., 5.);
```

Poteškoæe, a èesto i pogreške nastaju kada se iz funkcije želi prenijeti lokalni objekt generiran pri pozivu funkcije. Uzmimo da smo poželjeli funkciju koja æe sadržaj jednog znakovnog niza “nalijepiti” na drugi znakovni niz i novonastali niz vratiti kao rezultat. Nazovimo tu funkciju `dolijepi()`; argumenti te funkcije bit æe pokazivaèi na nizove koje želimo spojiti, a povratna vrijednost æe biti referenca na novostvoreni niz. Deklaracija funkcije bi prema tome izgledala kao:

```
char &dolijepi(const char*, const char*);
```

Pokušajmo definirati funkciju na sljedeæi naèin:

```
char *dolijepi(const char *prvi, const char *drugi) {
    char spojeni[80];           // lokalno polje znakova
    char *indeks = spojeni;
    while (*prvi)
        *(indeks++) = *(prvi++);
    while (*drugi)
        *(indeks++) = *(drugi++);
    *indeks = '\0';
    return spojeni;             // pogreška: pokazivač
                                //  na lokalni objekt
}
```

Unutar funkcije se generira lokalno polje `spojeni` u koje se stapaju znakovni nizovi na koje pokazuju argumenti funkcije. Pri izlasku iz funkcije prenosi se pokazivaè na to polje pozivnom kôdu. Meðutim, kako se lokalno polje `spojeni` izlaskom iz funkcije gubi iz vidokruga, rezultat je nepredvidiv[†].

Kao rezultat funkcije se nikada ne smije vraæati referenca ili pokazivaè na lokalni objekt generiran unutar funkcije.

Želimo li ipak da gornja funkcija kao rezultat vrati znakovni niz, morat æemo kao parametar proslijediti pokazivaè na memorijsko podruèje u koje želimo smjestiti rezultat:

```
char *dolijepi(const char *prvi, const char *drugi,
               char *spojeni);
```

Time smo "vruæe kestenje" prebacili na pozivajuæi kôd, koji sada preuzima brigu o alociranju i oslobaðanju prostora za znakovni niz, a pozvana funkcija æe samo mijenjati sadržaj alociranog prostora.

Zadatak: *Napišite kôd za ovako deklariranu funkciju* `dolijepi()`*. Obratite pažnju na vraćanje vrijednosti.*

7.6. Život jednog objekta

I objekti nisu besmrtni – postoji mjesto na kojima se stvaraju i gdje im se dodjeljuje memorija, te mjesto gdje se uništavaju, odnosno gdje se memorija oslobaða. Ima više vrsta objekata s obzirom na njihovo trajanje. Također, neki objekti mogu u određenom trenutku postojati, ali ne moraju biti dostupni. Zbog toga, C++ jezik specificira nekoliko *smještajnih klasa* (engl. *storage classes*) koje određuju naèin na koji æe se objekt stvoriti i uništiti, te kada æe objekt biti dostupan. Smještajna klasa objekta se specificira prilikom njegove deklaracije.

7.6.1. Lokalni objekti

U poglavlju o argumentima funkcija spoznali smo da se prilikom poziva funkcije generiraju privremeni objekti kojima se pridružuju vrijednosti stvarnih argumenata. Buduæi da funkcija barata s preslikama podataka, za podatke prenesene po vrijednosti promjene unutar funkcije neæe imati nikakvog odraza na izvornike u kôdu koji je funkciju pozvao. Ovo je velika prednost kada se žele podaci saèuvati od nekontroliranih izmjena u pozivajuæim funkcijama.

Također, pojedina funkcija u svojem radu može za realizaciju željenog algoritma iziskivati niz pomoćnih objekata. Ti objekti nemaju smisla izvan same funkcije te je

[†] Neki prevoditelji æe na ovakav pokušaj vraæanja vrijednosti prijaviti pogrešku, onemoguæavajuæi daljnje prevoðenje, odnosno povezivanje.

njihov život vezan za izvođenje funkcije. Takvi objekti se zovu lokalni objekti, jer vrijede lokalno za samu funkciju. Objekt nije vezan striktno za funkciju, već za blok u kojemu je deklariran (vidi poglavlje 3.1), no funkcija u biti nije ništa drugo nego blok. Na primjer:

```
void sortiraj(int *polje, int duljina) {
    int i, j;
    for (i = duljina - 1; i > 0; i--)
        for (j = 0; j < i; j++)
            if (polje[j] > polje[j + 1]) {
                int priv = polje[j + 1];
                polje[j + 1] = polje[j];
                polje[j] = priv;
            }
}
```

U gornjem primjeru varijable `i` i `j` su brojaèi koji nemaju smisla izvan funkcije. Zbog toga se oni deklariraju kao lokalni. Memorija se dodjeljuje na ulasku u funkciju i oslobaða na izlasku iz nje. Varijabla `priv` je takoðer lokalna, s time da je ona deklarirana u ugnježðenom bloku. Ona živi samo unutar tog bloka, stvara se na poèetku bloka i uništava na kraju.

Za lokalne varijable se kaže da imaju *automatsku smještajnu klasu* (engl. *automatic storage class*) koja se ispred deklaracije može eksplicitno navesti ključnom riječi `auto`:

```
auto int priv;
```

Automatska smještajna klasa se podrazumijeva ako se ništa drugo ne specificira, pa se zbog toga kljuèna rijeè `auto` koristi vrlo rijetko.

Lokalne varijable mogu imati i *registarsku smještajnu klasu* (engl. *register storage class*) koja se naznačava tako da se ispred deklaracije navede ključna riječ `register`:

```
register int priv;
```

Time se prevoditelju daje na znanje da se neka varijabla koristi vrlo intenzivno, te da se radi performansi programa preporuèa smještanje varijable u registar procesora umjesto u glavnu memoriju. Prevoditelj to može poslušati, ali i ne mora. Jedna od situacija u kojima æe prevoditelj gotovo sigurno zanemariti registarsku smještajnu klasu jest ako se uzme adresa varijable: podatak u registru nema adresu. Bolji prevoditelji æe i bez eksplicitnog navoðenja automatski varijable smještati u registre ako to mogu uèiniti.

7.6.2. Globalni objekti

Ponekad je ipak poželjno da promjene na varijablama imaju odraza i izvan tijela funkcije. Pretpostavimo da u nekom programu imamo tri cjelobrojne varijable: `dan`, `mjesec` i `godina` koje sadrže podatke o tekuæem datumu. Funkcija `noviMjesec`

zadužena je za promjenu varijable `mjesec`, ali i za promjenu varijable `godina` na kraju kalendarske godine:

```
int noviMjesec(int mjesec, int godina) {
    if (++mjesec > 12) {
        mjesec = 1;
        godina++;       // bez efekta
    }
    return mjesec;
}

int main() {
    int dan = 30;
    int mjesec = 1;
    int godina = 1997;

    mjesec = noviMjesec(mjesec, godina);
    return 1;
}
```

Oèito je da æe se promjena mjeseca, preko povratne vrijednosti, odraziti u pozivnom kôdu. Meðutim, promjena godine æe se po povratku u glavnu funkciju izgubiti.

Poneki čitatelj će se dosjetiti, te umjesto po vrijednosti datumske varijable prenositi kao reference ili kao pokazivače. Funkcija će tada baratati (posredno) s izvornicima, pa povratna vrijednost funkcije `noviMjesec` nije niti potrebna. Na žalost, i ovakvo rješenje tek djelomično uklanja problem. Ilustrirajmo to sljedećom situaciju: recimo da se funkcija `noviMjesec` ne poziva izravno iz glavne funkcije, već se prethodno poziva funkcija `noviDan` koja povećava dan u tekućem datumu. Očito će funkcija `noviDan` tek povremeno pozivati funkciju `noviMjesec`. Čak i kada bismo `dan`, `mjesec` i `godina` prenosili preko pokazivača na njih, trebali bismo pokazivač na `godina` prenijeti funkciji `noviDan` samo zato da bi ona taj pokazivač mogla dalje prenijeti funkciji `noviMjesec`:

```
int noviMjesec(int *mjesec, int *godina);
int noviDan(int *dan, int *mjesec, int *godina);
```

U deklaraciji funkcije æe se pojaviti dodatni argumenti koji samoj funkciji nisu potrebni, što æe u svakom sluèaju èiniti kôd glomaznijim i neèitljivijim. Zamislimo samo kako bi to sve izgledalo za više razina poziva funkcija. Suvišni argumenti bi se gomilali, i veæ nakon nekoliko razina neke funkcije bi imale desetak argumenata.

Jednostavno rješenje ovakvog problema je da se objekti koji trebaju biti dohvaćani iz nekoliko različitih funkcija deklariraju kao globalni. Da bi neka varijabla postala globalna, treba ju deklarirati izvan tijela funkcije. Vrlo je vjerojatno da će datumske varijable trebati i drugim funkcijama, pa ćemo ih definirati ispred funkcije `main()`:

```
int noviMjesec();           // deklaracija funkcije
```

```
int dan, mjesec, godina;    // globalne varijable

int main() {
    dan = 30;
    mjesec = 1;
    godina = 1997;
    mjesec = noviMjesec();
    return 1;
}

int noviMjesec() {
    if (++mjesec > 12) {
        mjesec = 1;
        godina++;
    }
    return mjesec;
}
```

Varijable `dan`, `mjesec` i `godina` deklarirane su kao globalne te su sada vidljive funkcijama `main()` i `noviMjesec()` koje slijede iza deklaracije objekata, tako da ih ne moramo posebno prenositi kao argumente u pozivu funkcija. Štoviše, u gornjem primjeru nema potrebe da se vrijednost mjeseca vraæa kao rezultat.

Objekti deklarirani izvan funkcije vidljivi su u svim funkcijama èije definicije slijede u datoteci izvornog kôda.

Da smo kojim sluèajem u gornjem kôdu, deklaracije varijabli datumskih varijabli stavili iza funkcije `main()`, ona ih ne bi mogla dohvaæati, pa bi prevoditelj javio pogrešku da te varijable nisu deklarirane u funkciji `main()`.

Globalna deklaracija ujedno je i definicija. Naime, svi globalni objekti se umeću u izvedbeni kôd. Oni žive od početka izvođenja programa do njegovog kraja. Prilikom deklaracije objekta moguće je odmah provesti i inicijalizaciju po pravilima koja smo do sada naučili.

Ako nisu eksplicitno inicijalizirani, globalni objekti æe se inicijalizirati na vrijednost 0. Lokalni objekti koji nisu eksplicitno inicijalizirani poprimit æe neku sluèajnu vrijednost.

To znaèi da æe pri izvoðenju sljedeæeg programa:

```
#include <iostream.h>

int globalna;       // neinicijalizirana globalna varijabla

int main() {
    int lokalna;// neinicijalizirana lokalna varijabla
```

```
    cout << globalna << endl;
    cout << lokalna << endl;
}
```

prva naredba za ispis na zaslonu ispisati nulu, a druga neki neodreðeni broj koji æe varirati ovisno o uvjetima pod kojima je program preveden i pokrenut.

Zadatak: *Razmislite i provjerite što će biti ispisano u gornjem primjeru, ako se* `globalna` *i* `lokalna` *deklariraju kao pokazivači na znakovne nizove.*

Oèito se deklariranjem varijabli kao globalnih pojednostavnjuje pristup njima. Međutim, time se one izlažu opasnosti od nekontroliranih promjena unutar razlièitih funkcija.

Globalne varijable treba koristiti èim je moguæe rjeðe.

Podaci koji trebaju biti dostupni svim funkcijama, ali se ne smiju mijenjati (npr. univerzalne matematièke ili fizièke konstante) deklariraju se kao globalne konstante, dodavanjem modifikatora `const` ispred deklaracije tipa.

Globalni objekti mogu imati dvije smještajne klase: vanjske ili statičke. Ako se kôd proteže preko nekoliko modula, globalni objekti deklarirani na gore opisani način bit će vidljivi u svim modulima. Više modula može koristiti isti objekt, s time da tada objekt smije biti definiran isključivo u jednom modulu. U preostalim modulima objekt je potrebno samo deklarirati, ali ne i definirati. Deklaracija se provodi tako da se ispred naziva objekta stavi ključna riječ `extern`:

```
extern int varijabla_u_drugom_modulu;
```

Objekti deklarirani s `extern` imaju *vanjsko povezivanje* (engl. *external linkage*), što znaèi da æe biti dostupni drugim modulima prilikom povezivanja. Ako se ništa ne navede, podrazumijevano je da se radi o eksternoj deklaraciji i definiciji.

Poneki objekti mogu biti od koristi samo unutar jednog modula. Takvi objekti se mogu učiniti statičkima, čime se otvara mogućnost da i drugi moduli deklariraju objekte s istim nazivom. Ti objekti su zapravo lokalni za modul i imaju *unutarnje povezivanje* (engl. *internal linkage*). Objekt se može učiniti statičkim tako da se ispred deklaracije umetne ključna riječ `static`:

```
static int SamoMoj = 8;
```

Konstantni objekti, ako se drukèije ne navede, automatski imaju statièku smještajnu klasu. Ako takve objekte želimo koristiti u drugim modulima, potrebno je objekt deklarirati eksternim.

Ako se u funkciji deklarira lokalni objekt istog imena kao i globalni objekt, globalni objekt će biti skriven. Na primjer:

```
int a = 10;             // globalna varijabla

int main() {
    float a = 50;       // lokalna varijabla
    cout << a << endl;  // ispisuje 50
}
```

U gornjem primjeru globalna varijabla `a` je unutar funkcije `main()` skrivena: navoðenje samo identifikatora `a` rezultira pristupom lokalnom objektu. No moguæe je pristupiti globalnom objektu tako da se ispred naziva objekta navede operator `::` (dvije dvotoèke) – *operator za odreðivanje podruèja* (engl. *scope resolution operator*). Naime, svi globalni objekti pripadaju globalnom podruèju, a ovim operatorom se eksplicitno odreðuje pristup tom podruèju:

```
int a = 10;                 // globalna varijabla

int main() {
    float a = 50;           // lokalna varijabla
    cout << a << endl;      // lokalna varijabla:  50
    cout << ::a << endl;    // globalna varijabla: 10
}
```

7.6.3. Statièki objekti u funkcijama

Lokalne varijable unutar funkcije se inicijaliziraju svaki puta prilikom poziva funkcije. Međutim, postoje situacije kada je poželjno da se vrijednost neke varijable inicijalizira samo pri prvom pozivu funkcije, a između poziva te funkcije da se vrijednost èuva. Istina, takva se varijabla može deklarirati kao globalna, ali æe tada biti dohvatljiva i iz drugih funkcija i izložena opasnosti od nekontrolirane promjene. Rješenje za ovakve sluèajeve pružaju *statièki objekti* (engl. *static objects*). Ove objekte treba razlikovati od statièkih globalnih objekata iz prethodnog odsjeèka – ovdje se radi o statièkim objektima unutar funkcija.

Dodavanjem ključne riječi `static` ispred oznake tipa, objekt postaje statički – on se inicijalizira samo jednom, prilikom prevođenja, te se takav član pohranjuje u datoteku zajedno s izvedbenim kôdom. Ilustrirajmo to sljedećim primjerom:

```
#include <iostream.h>
#include <stdlib.h>

void VoliMeNeVoli() {
    static bool VoliMe = false;         // statički objekt
    VoliMe = !VoliMe;
    if (VoliMe)
        cout << "Voli me!" << endl;
    else
        cout << "Ne voli!" << endl;
```

```
}

int main() {
    int i = rand() % 10 + 1;    // slučajni broj od 1 do 10
    while (i--)
        VoliMeNeVoli();
}
```

U glavnoj funkciji pomoæu standardne funkcije `rand()` (deklarirane u `stdlib.h`) generira se sluèajni broj između 1 i 10, te se toliko puta poziva funkcija `VoliMeNeVoli()`. U funkciji `VoliMeNeVoli()` deklarirana je statièka varijabla `VoliMe` tipa `bool`. Kod prevođenja ona se inicijalizira na vrijednost `false` (*neistina*), ali se prilikom svakog poziva funkcije `VoliMeNeVoli()` njena vrijednost mijenja – to æe prouzroèiti naizmjenièni ispis obiju poruka i na zaslonu æe se pojaviti slijed oblika

```
Voli me!
Ne voli!
Voli me!
Ne voli!
```

Da je izostavljena deklaracija `static`, varijabla `VoliMe` bi bila pri svakom ulasku u funkciju inicijalizirana na istu vrijednost (`false`) te bi svaki poziv funkcije dao ispis iste poruke:

```
Voli me!
Voli me!
Voli me!
Voli me!
```

Statièki objekti u funkcijama žive za vrijeme cijelog izvođenja programa, ali su dostupni samo unutar funkcije u kojoj su deklarirani. Te statièke objekte možemo shvatiti kao da su deklarirani izvan funkcije koristeæi smještajnu klasu `static`, s time da im se može pristupiti samo iz funkcije u kojoj su deklarirani. Zato se i koristi ista kljuèna rijeè `static`: smještaj ovakvih objekata i globalnih statièkih objekata se provodi na isti naèin (u *podatkovni segment* programa), pa æe i statièki objekti imati poèetnu vrijednost 0 ako se ne inicijaliziraju drukèije.

7.7. Umetnute funkcije

Èesto se koriste funkcije koje imaju vrlo kratko tijelo. Sam poziv, tijekom kojeg se stvaraju i inicijaliziraju lokalne varijable, za takve kratke funkcije može trajati znatno dulje od njenog izvršavanja. Veæ smo imali primjer funkcije `kvadrat()` koja se sastoji samo od jedne naredbe:

```
double kvadrat(float x) {
    return x * x;
}
```

Izvođenje prevedenog kôda bilo bi brže kada bi svaki poziv funkcije `kvadrat()` u programu bio jednostavno zamjenjen naredbom za međusobno množenje dva ista broja.

Dodavanjem ključne riječi `inline` ispred definicije funkcije daje se naputak prevoditelju da pokuša svaki poziv funkcije nadomjestiti samim kôdom funkcije. Takve se funkcije nazivaju *umetnute funkcije* (engl. *inline function*). Evo primjera:

```
inline double kvadrat(float x) {
    return x * x;
}
```

Valja naglasiti da je to samo naputak prevoditelju, a ne i naredba. Prevoditelj æe shodno svojim moguænostima i procjeni taj naputak provesti u djelo ili æe ga ignorirati. Svaki bolji prevoditelj æe generirati umetnuti kôd za ovakvu funkciju `kvadrat()`. Međutim, ako je tijelo funkcije složenije, posebice ako sadrži petlju, uspjeh uklapanja funkcije æe znaèajno varirati od prevoditelja do prevoditelja.

Definicija umetnute funkcije mora prethoditi prvom pozivu funkcije – u ovom slučaju prevoditelju nije dovoljna deklaracija funkcije, budući da pri nailasku na prvi poziv umetnute funkcije već mora znati čime će poziv te funkcije nadomjestiti (ako će to uopće uraditi). Stoga se definicija umetnute funkcije obično navodi zajedno s deklaracijama ostalih funkcija i klasa, najčešće u zasebnoj datoteci (o tome će biti govora u poglavlju o organizaciji kôda).

Korištenjem umetnutih funkcija eliminira se neophodni utrošak vremena koji nastaje prilikom poziva funkcije. Međutim, svaki poziv umetnute funkcije nadomješta se tijelom funkcije, što može znatno povećati duljinu izvedbenog kôda. Pretjerana uporaba umetnutih funkcija može značajno produljiti postupak prevođenja, posebice ako se programski kôd rasprostire kroz nekoliko odvojenih datoteka. Promijeni li se tijelo umetnute funkcije, prevoditelj mora proći cjelokupni kôd da bi preslikao tu promjenu na sva mjesta poziva. Naprotiv, pri promjeni tijela funkcije koja nije umetnuta, postupak prevođenja treba ponoviti samo u datoteci u kojoj se nalazi definicija dotične funkcije.

Zbog navedenih činjenica, očigledno treba biti vrlo obziran s korištenjem `inline` funkcija i njihovu uporabu ograničiti na najjednostavnije funkcije. Najčešće se kao umetnute definiraju funkcije koje vraćaju vrijednost ili referencu na podatkovni član unutar nekog objekta – o tome će biti više riječi u poglavlju o klasama.

Jedan od nedostataka umetnutih funkcija je i nemogućnost praćenja toka programom za simboličko otkrivanje pogrešaka (engl. *debugger*) u izvedbenom kôdu. Budući da se pozivi funkcije nadomještaju njenim tijelom, funkcija u izvedbenom kôdu ne postoji, pa se (najčešće) ne može ni analizirati programom za simboličko otkrivanje pogrešaka. Zbog toga je, prilikom pisanja programa, najbolje za prvu ruku sve funkcije napraviti kao obične. Tek nakon što se kôd pokaže ispravnim, neke se funkcije redefiniraju kao umetnute, te se ispita pripadajući dobitak u izvedbenom programu.

7.8. Preopterećenje funkcija

Ako tip stvarnog argumenta funkcije u pozivu ne odgovara tipu navedenom u deklaraciji, tada se provodi konverzija tipa i stvarni argument se svodi na tip formalnog argumenta. Primjerice, u donjem kôdu funkcija `kvadrat()` deklarirana je za argument tipa `double`, a poziva se sa cjelobrojnim argumentom:

```
float kvadrat(float x) {
    return x * x;
}

int main() {
    int i = 16;
    i = kvadrat(i);
}
```

Prilikom pridruživanja stvarnog argumenta (cjelobrojne konstante 16) formalnom argumentu koji je tipa `float`, provodi se konverzija tipa argumenta. Unutar funkcije barata se argumentom tipa `float` i kao rezultat se vraća podatak tog tipa. Međutim, u pozivnom kodu povratna se vrijednost opet svodi na `int`. Kao što vidimo, pri pozivu funkcije provedene su dvije suvišne konverzije koje u suštini ne utječu na točnost rezultata, a još k tomu produljuju poziv i izvođenje funkcije jer se množenje realnih brojeva odvija dulje nego množenje cijelih brojeva.

Očito bi bilo daleko prikladnije definirati funkciju za kvadriranje cijelih brojeva koja bi baratala sa cijelim brojem i vraćala cjelobrojni rezultat. Ta funkcija bi općenito mogla imati bilo koje ime, ali je za razumijevanja kôda najbolje kada bi se i ona zvala `kvadrat()`:

```
int kvadrat(int x) {
    return x * x;
}
```

No što učiniti ako su nam u programu potrebna oba oblika funkcije `kvadrat()`, tj. kada želimo kvadrirati i cijele i decimalne brojeve? `float` inačica funkcije će podržavati oba tipa podatka, ali će, kao što smo već primijetili, iziskivati nepotrebne konverzije tipa.

Programski jezik C++ omogućava *preopterećenje funkcija* (engl. *function overloading*) – korištenje istog imena za različite inačice funkcije. Pritom se funkcije moraju međusobno razlikovati po potpisu, tj. po tipu argumenata u deklaraciji funkcije. Prevoditelj će prema tipu argumenata sam prepoznati koju inačicu funkcije mora za pojedini poziv upotrijebiti. Tako će u kôdu:

```
float kvadrat(float);        // deklaracije preopterećenih
int kvadrat(int);            // funkcija

int main() {
    float x = 0.5;
    int n = 4;
```

```
    cout << kvadrat(x) << endl;      // float kvadrat(float)
    cout << kvadrat(n) << endl;      // int kvadrat(int)
}
```

prevoditelj prvom pozivu funkcije `kvadrat()` s argumentom tipa `float` pridružiti `float` inaèicu funkcije, dok æe drugom pozivu s argument tipa `int` pridružiti `int` inaèicu funkcije.

Gledano sa stanovišta prevoditelja, jedino što je zajedničko gornjim funkcijama jest ime. Preopterećenje imena samo olakšava programeru da smisleno poistovjeti funkcije koje obavljaju slične radnje na različitim tipovima podataka. Ovo je naročito praktično kod funkcija koje koriste općeprihvaćena imena, poput `sin`, `cos`, `sqrt`. Pritom valja uočiti da se odluka o pridruživanju pojedine funkcije donosi tijekom prevođenja kôda; na sam izvedbeni kôd to nema nikakvog odraza te bi izvedbeni kôd bi (teoretski) trebao biti potpuno identičan kao i da su pojedine preopterećene funkcije imale međusobno potpuno različita imena. Stoga je u biti točnije govoriti o *preopterećenju imena* funkcija.

Međutim, preopterećenje imena postavlja dodatne probleme za prevoditelja: samo ime funkcije više nije dovoljno da bi se jednoznačno odredila funkcija koja će se koristiti za određeni poziv. Što ako u gornju glavnu funkciju ubacimo poziv funkcije `kvadrat()` s nekim drugim tipom argumenta (primjerice `double`) za koji nije definirana funkcija?

```
float kvadrat(float);        // deklaracije preopterećenih
int kvadrat(int);            // funkcija

int main() {
    double dupliX = 3456.54321e49;
    cout << kvadrat(dupliX) << endl;        // pogreška!
}
```

Buduæi da nije deklarirana funkcija `kvadrat(double)`, prevoditelj je prisiljen zadovoljiti se postojeæim funkcijama, tj. mora provesti odgovarajuæu konverziju tipa stvarnog argumenta. Na raspolaganju su dvije funkcije `kvadrat()`: za argument tipa `float` i za argument tipa `int`. Pretvorba `double` podatka u bilo koji od tih tipova u opæenitom sluèaju podrazumijeva "kresanje" podatka tako da su konverzije `double` u `float`, te `double` u `int` ravnopravne – ovakav poziv æe izazvati dvojbu kod prevoditelja i on æe prijaviti pogrešku. Slièna situacija je i u sljedeæem kôdu:

```
long kvadrat(long);
float kvadrat(float);

int main() {
    kvadrat(42);     // pogreška: nedoumica je li
                         // kvadrat(long) ili kvadrat(float)
    kvadrat(3.4);        // isti problem
}
```

Brojèanim konstantama nije eksplicitno odreðen tip pa ih prevoditelj (implicitno) interpretira kao podatke tipa `int`, odnosno `double`, koje prilikom poziva funkcije treba pretvoriti u `long` ili `float`. Pretvorbe u ta dva tipa su ravnopravne tako da prevoditelj ostaje u nedoumici i prijavljuje pogreške za oba poziva.

Naprotiv, neke pretvorbe tipova (npr. `float` u `double` ili `int` u `long`) su trivijalne, tako da će sljedeći kôd biti preveden bez prijave o pogreški:

```
long apasalutno(long);          // samo deklaracije
double apasalutno(double);

int main() {
    float x = -23.76;
    cout << apasalutno(x) << endl;
    cout << apasalutno(-473) << endl;
}
```

Oèito je da moraju postojati toèno odreðena pravila po kojima prevoditelj prepoznaje koju preoptereæenu funkciju mora prilikom pojedinog poziva pridružiti. Pretraživanje se provodi sljedeæim redoslijedom:

1. Traži se funkcija za koju postoji egzaktno slaganje parametara. To podrazumijeva slaganje kod kojeg nije potrebna nikakva konverzija tipa ili su potrebne samo *trivijalne konverzije* (konverzija imena polja u pokazivaè, imena funkcije u pokazivaè na funkciju, konverzije tipa u referencu i obrnuto, konverzija tipa u `const`).
2. Traži se funkcija kod koje treba provesti samo cjelobrojnu promociju (`char` u `int`, `short` u `int`, odnosno u njihove `unsigned` ekvivalente, pobrojenja u `int`) i pretvorbu `float` u `double`.
3. Traži se funkcija za koju treba provesti standardne konverzije (na primjer `int` u `double`, `int` u `unsigned int`, pokazivaè na izvedenu klasu u pokazivaè na osnovnu klasu).
4. Traži se funkcija za koju su neophodne korisnièki definirane konverzije (o njima æe biti rijeèi kasnije, u poglavlju o klasama).
5. Traži se funkcija s neodreðenim argumentima (`...`) u deklaraciji

Ako se u nekoj od toèaka naiðe na dvije ili više funkcija koje zadovoljavaju dotièni uvjet, poziv se proglašava neodreðenim i prevoditelj prijavljuje pogrešku. Valja imati na umu da pri traženju odgovarajuæe funkcije redoslijed deklaracija preoptereæenih funkcija nema nikakav utjecaj na odabir.

Zadatak. *Odredite za gornje primjere s pogreškom u kojim je točkama navedenog redoslijeda nastupila neodređenost.*

Ako funkcija ima više argumenata, tada se gornji postupak nalaženja odgovarajuæe funkcije provodi za svaki od argumenata. Ako jedna od funkcija osigurava bolje slaganje za jedan od argumenata, a barem jednako dobro slaganje za ostale argumente, ona æe biti odabrana.

Budući da bilo koji tip `T` (na primjer `int`) i referenca na taj tip `T &` (na primjer `int &`) prihvaćaju jednake inicijalizirajuće vrijednosti, funkcije čiji se argumenti razlikuju samo po tome radi li se o vrijednosti ili referenci na nju, ne mogu se preopterećivati. Zato će prevoditelj prilikom prevođenja sljedećeg kôda prijaviti pogrešku:

```
void f(int i) {
    //...
}

void f(int &refi) {     // pogreška: nedovoljna razlika
    //...
}
```

Isto vrijedi i za razliku između argumenata tipa `T`, `const T` i `volatile T`, koje prevoditelj neæe razluèiti. Nasuprot tome, razliku između argumenata tipa `T &` (referenca na tip `T`), `const T &` i `volatile T &`, odnosno između `T *` (pokazivaè na tip `T`), `const T *` i `volatile T *` prevoditelj prepoznaje, tako da se funkcija s obzirom na te argumente smije preoptereæivati.

Pomutnju kod preopterećenja mogu izazvati podrazumijevani argumenti. Dvije deklaracije parametara koje se razlikuju samo u podrazumijevanim argumentima su potpuno ekvivalentne. Zbog toga će poziv funkcije `f` u glavnoj funkciji izazvati nedoumicu kod prevoditelja:

```
#include <iostream.h>

void f(int i = 88) {
    cout << i << endl;
}

void f() {
    cout << "Ništa!" << endl;
}

int main() {
    f();        // pogreška: f(int) ili f()?
```

Funkcije se preoptereæuju samo prema tipovima argumenata, ali ne i prema tipu povratne vrijednosti.

```
}
```

To znaèi da je dozvoljeno funkciju `kvadrat()` preopteretiti funkcijama koje vraæaju jednake tipove podataka:

```
long kvadrat(int);
long kvadrat(long);
```

Meðutim, na pokušaj preoptereæenja:

```
double kub(float);          // pogreška: obje funkcije
float kub(float);           //  su kub(float)
```

prevoditelj æe dojaviti pogrešku o pokušaju ponovne deklaracije funkcije, buduæi da su argumenti obiju funkcija potpuno jednakih tipova.

Pisanje ovako trivijalnih preopterećenih funkcija, u kojima je algoritam potpuno isti za svaki tip podataka je bitno jednostavnije ako se koriste *predlošci funkcija* (vidi poglavlje 7.13). Međutim, ponekad se algoritmi za različite operacije svode pod isto nazivlje. Primjerice, apsolutna vrijednost (modul) kompleksnih brojeva se računa prema bitno drukčijem algoritmu nego primjerice realnih ili cijelih brojeva. Također, optimizacija kôda ponekad iziskuje korištenje različitih algoritama. Ilustrirajmo to funkcijom za potenciranje `mocnica()`: za decimalne potencije ćemo definirati funkciju koja će potenciju računati pomoću formule

$$e^{y \cdot \ln(x)} \quad (\text{za } x > 0)$$

dok æemo za cjelobrojne potencije koristiti petlju u kojoj æemo mantisu množiti odreðeni broj puta[†]. Za raèunanje prirodnog logaritma i eksponencijalne funkcije koristit æemo standardne funkcije `log()`, odnosno `exp()` deklarirane u `math.h` datoteci zaglavlja.

```
#include <iostream.h>
#include <math.h>

double mocnica(double x, double y) {
    if (x <= 0.) {
        cerr << "Nemoćan sam to izračunati!" << endl;
        return 0.;
    }
    return exp(y * log(x));
}

double mocnica(double x, int y) {
    int brojac = y > 0 ? y : - y;
    double rezultat = 1.;
    while (brojac-- > 0)
        rezultat *= x;
    return y > 0 ? rezultat : 1. / rezultat;
}
```

† U biblioteci matematièkih funkcija `math` veæ postoji funkcija `pow(double, double)` koja obavlja istu zadaæu.

Ovakvo preoptereæenje funkcije osigurava toèan rezultat potenciranja cjelobrojnim eksponentom i za negativne baze:

```
cout << mocnica(-3, 3) << endl;        // ispisuje -27
cout << mocnica(-3., 3) << endl;       // ponovno
cout << mocnica(-2., 3.) << endl;      // ispisuje poruku
```

Prvom i drugom pozivu prevoditelj pridružuje funkciju `mocnica(double, int)`, dok treæem pozivu funkciju `mocnica(double, double)`.

Zanimljivo je uočiti da unatoč (prividno) kraćem izvornom kôdu funkcije `mocnica(double, double)`, njeno izvođenje traje znatno dulje. Uzrok tome su logaritamska i eksponencijalna funkcija koje se u njoj koriste, a izračunavanje kojih uključuje daleko kompleksnije operacije nego što je množenje u petlji druge funkcije. Stoga preopterećenje funkcijom `mocnica(double, int)` možemo shvatiti i kao određenu optimizaciju izvedbenog kôda koja za određene vrijednosti (tipove) argumenata omogućava brže računanje potencije.

Iz ovih razmatranja je uočljivo koliko zbog složenih pravila pronalaženja odgovarajuće funkcije valja biti oprezan prilikom preopterećenja funkcija. U svakom slučaju treba izbjegavati situacije u kojima nije očito koja će funkcija biti izabrana iz skupa preopterećenih funkcija.

Spomenimo na kraju da se osim funkcija mogu preopterećivati i operatori. Pažljiviji čitatelj će se odmah sjetiti kako su aritmetički operatori definirani za različite tipove podataka i da je shodno tipovima operanada i rezultat različitog tipa. Podsjetimo se kako operator `+` primijenjen na dva podatka tipa `int` daje rezultat tipa `int`, a ako se primijeni na dva podatka tipa `float`, rezultat će biti toga tipa, itd. Prema tome, više je nego očito da je operator zbrajanja preopterećen za parove svih ugrađenih tipova podataka – u slučaju da su operandi različitih tipova, konverzijama se podaci svode na zajednički tip i primjenjuje odgovarajući (preopterećeni) operator.

Budući da su operatori definirani za sve ugrađene tipove podataka, ne možemo ih dodatno preopterećivati, osim ako uvodimo korisnički definirane tipove. Stoga ćemo o preopterećivanju operatora govoriti nakon što upoznamo *klase*.

Zadatak. *Napišite funkciju* `usporedi()` *koja će uspoređivati dva argumenta koji joj se prenose. Ako je prvi argument veći od drugoga, funkcija treba kao rezultat vratiti 1, ako su oba argumenta međusobno jednaka, rezultat mora biti 0, a ako je drugi argument veći, rezultat mora biti –1. Preopteretite funkciju tako da za argumente prihvaća brojeve tipa* `double`*, te pokazivač na znakovni niz:*

```
int usporedi(const double prviBr, const double drugiBr);
int usporedi(const char *prviNiz, const char *drugiNiz);
```

7.9. Rekurzija

Pozivom funkcije formiraju se unutar nje lokalne varijable koje su nedohvatljive izvan same funkcije. Neovisnost lokalnih varijabli od okolnog programa omoguæava da funkcija poziva samu sebe – ponovnim pozivom se stvaraju nove lokalne varijable potpuno neovisne o varijablama u pozivajuæoj funkciji. Ovakav proces uzastopnog pozivanja naziva se *rekurzija* (engl. *recursion*). Razmotrimo rekurziju na jednostavnom primjeru: funkciji za raèunanje faktorijela:

```
int faktorijel(int n) {
    return (n > 1) ? (n * faktorijel(n - 1)) : 1;
}
```

Pozovemo li tu funkciju naredbom

```
int nf = faktorijel(3);
```

funkcija `faktorijel()` æe pozivati samu sebe u tri "razine":

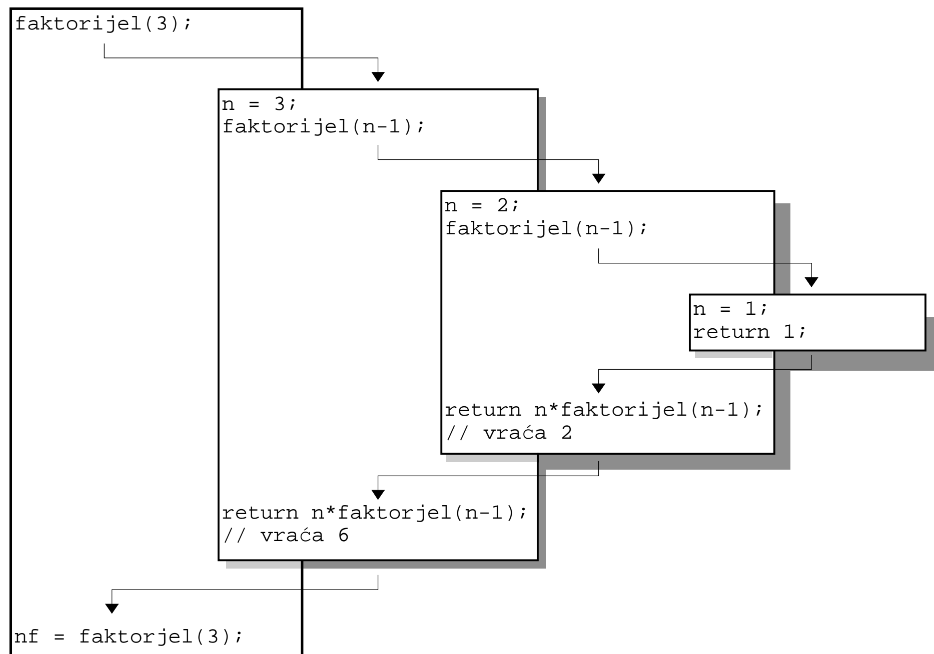
1. Prilikom prvog poziva formalni argument æe biti inicijaliziran na vrijednost `n = 3`. U uvjetnom izrazu æe funkcija `faktorijel()` biti pozvana po drugi put, ali sada sa stvarnim argumentom 2.
2. U drugom pozivu formalni argument se inicijalizira na `n = 2`. Argument pozivajuæe funkcije (`n = 3`) još uvijek postoji, ali je on nedohvatljiv i potpuno neovisan od argumenta drugopozvane funkcije. U uvjetnom izrazu æe funkcija `faktorijel()` biti treæi puta pozvana, ali ovaj puta sa stvarnim argumentom 1.
3. U treæem pozivu funkcije `faktorijel()` formalni argument se inicijalizira na vrijednost 1 (u pozivajuæim funkcijama formalni argumenti još uvijek postoje s neizmijenjenim vrijednostima). Buduæi da nije zadovoljen uvjet u uvjetnom izrazu, funkcija `faktorijel()` se više ne poziva, veæ se kao povratna vrijednost pozivnoj funkciji šalje broj 1.

Slijedi postepeni povratak do poèetnog poziva funkcije:

1. Vrijednost 1 koju je vratila treæepozvana funkcija množi se s vrijednošæu formalnog argumenta u drugopozvanoj funkciji (`n = 2`), te se taj umnožak vraæa kao rezultat prvopozvanoj funkciji.
2. Vrijednost 2 koju je vratila drugopozvana funkcija množi se s vrijednošæu formalnog argumenta u poèetnoj funkciji (`n = 3`), te se taj umnožak vraæa kao rezultat kôdu koji je prvi pozvao funkciju `faktorijel()` s argumentom 3.

Izvoðenje kôda može se simbolièki prikazati slikom 7.2.

Mnoge matematičke formule se mogu implementirati primjenom rekurzije. Međutim, kod primjene rekurzija valja biti krajnje oprezan: lokalne varijable unutar funkcija i adrese s kojih se obavlja poziv funkcije se pohranjuju na stog, pa preveliki broj uzastopnih poziva funkcije unutar funkcije može vrlo brzo prepuniti stog i onemogućiti daljnje izvođenje programa. Daleko je sigurnije (i nerijetko efikasnije)



Slika 7.2. Primjer rekurzije.

takve formule implementirati jednostavnim petljama. Rekurzije mogu biti vrlo efikasno sredstvo za rukovanje posebnim strukturama podataka.

7.10. Pokazivaèi na funkcije

Zadajmo si sljedeæi zadatak: želimo napisati opæenitu funkciju `Integral()` za numerièko integriranje pomoæu trapeznog pravila. Uz standardne ulazne parametre, kao što su donja i gornja granica integracije, želimo da funkcija `Integral()` prihvaæa i ime podintegralne funkcije kao parametar. To bi nam omoguæilo da jednom napisanu funkciju `Integral()` možemo koristiti za bilo koju podintegralnu funkciju, èije æemo ime prenijeti kao parametar prilikom poziva, nešto poput:

```
I1 = Integral(podintegralnaFunkcija, donjaGr, gornjaGr);
I2 = Integral(sin, 0., pi);
```

Ako to nije moguæe, bit æemo prisiljeni za razlièite podintegralne funkcije svaki puta mijenjati izvorni kôd funkcije `Integral()`.

Rješenje za probleme ovakvog tipa pružaju pokazivači na funkcije. Osim što funkcija može primati određene argumente i vraćati neki rezultat, ona u izvedbenom kôdu programa zauzima i određeni prostor u memoriji računala. Kada pozivamo neku

funkciju, tada izvođenje programa prebacujemo na memorijsku lokaciju na kojoj je funkcija smještena. To prebacivanje se odvija preko pokazivača na tu funkciju, iako to u dosadašnjim razmatranjima nismo eksplicitno naglasili. To znači da je ime svake funkcije u stvari pokazivač na lokaciju gdje počinje njen kôd. Deklaracija funkcije `Integral()` koja kao argumente prihvaća pokazivač na funkciju, te donju i gornju granicu integrala izgledala bi ovako:

```
float Integral(double (*f)(double), double x0, double xn);
```

Prvi argument funkcije `Integral()` je pokazivaè na funkciju `f()` tipa `double` koja kao argument prihvaæa samo jedan podatak tipa `double`. Ispred imena parametra u listi argumenata nalazi se operator dereferenciranja `*` koji ukazuje na to da æemo funkciji `Integral()` prenijeti pokazivaè, a ispred operatora dereferenciranja je identifikator povratnog tipa (`double`) funkcije `f()`. Iza pokazivaèa na funkciju `f()` unutar zagrada se navode tipovi argumenata funkcije. Vjerojatno ne treba naglašavati da se ti argumenti moraju slagati s argumentima funkcije koja æe se pozivati po broju i tipu (ili se stvarni argumenti moraju dati svesti na formalne ugraðenim ili korisnièki definiranim pretvorbama).

Uočimo zagrade oko oznake za pokazivač `f` na funkciju. One su neophodne, jer bi u protivnom:

```
float Integral(double *f(double), double x0, double xn);
                     // pogrešno: deklaracija funkcije f
```

zbog višeg prioriteta zagrada nad operatorom `*`, prvi argument bio interpretiran kao deklaracija funkcije `f(double)` koja kao rezultat vraæa pokazivaè na `double`.

Posvetimo se sada definiciji funkcije `Integral()`. Za početak se podsjetimo ukratko o čemu se u trapeznom pravilu radi. Interval $\langle x_0, x_n \rangle$ unutar kojeg se funkcija $f(x)$ želi integrirati podijeli se na n jednakih intervala, između kojih se funkcija aproksimira pravcima (slika 7.3). Integral funkcije (tj. površina ispod funkcije) približno je jednaka zbroju površina tako dobivenih trapeza (šrafirane plohe na slici):

$$\int_{x_0}^{x_n} f(x)\,\mathrm{d}x \approx h\left(\frac{f(x_0)}{2} + f(x_1) + f(x_2) + \ldots + f(x_{n-1}) + \frac{f(x_n)}{2}\right)$$

gdje je h širina podintervala:

$$h = \frac{x_n - x_0}{n}$$

Naravno da je toènost aproksimacije to bolja što je broj podintervala n veæi, jer se u tom sluèaju razlomljeni pravci više približavaju zadanoj funkciji. Radi opæenitosti æemo postupak automatizirati, tako da æe funkcija prvo raèunati aproksimaciju samo za jedan interval:

$$I_0 = h_0 \left[\frac{f(x_0)}{2} + \frac{f(x_1)}{2} \right], \qquad h_0 = x_n - x_0$$

a zatim æe broj intervala udvostruèavati, raèunajuæi sve toènije aproksimacije integrala:

$$I_1 = \frac{h_0}{2} \left[\frac{f(x_0)}{2} + f(x_1) + \frac{f(x_2)}{2} \right]$$

$$I_2 = \frac{h_0}{4} \left[\frac{f(x_0)}{2} + f(x_1) + f(x_2) + f(x_3) + \frac{f(x_4)}{2} \right]$$

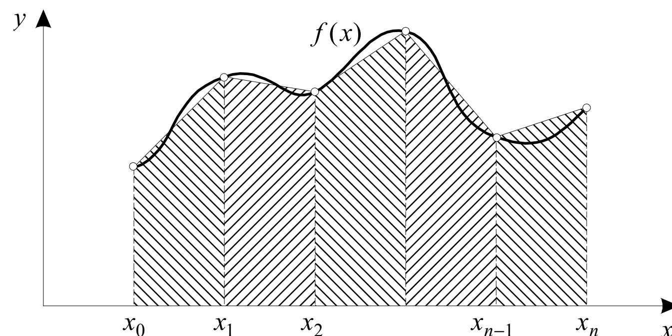
...

Buduæi da se broj podintervala podvostruèuje, u svakom koraku se raèunaju vrijednosti podintegralne funkcije i za sve toèke obraðene u prethodnom koraku. Da se izbjegne nepotrebno ponavljanje raèuna, podintegralnu funkciju je dovoljno raèunati samo za nove toèke i tu novu sumu (nazvat æemo ju `sumaNova`) pridodati sumi iz prethodnog koraka (`sumaStara`). Postupak æe se ponavljati sve dok su zadovoljena dva uvjeta:

1. relativna promjena sume je veæa od unaprijed dozvoljene vrijednosti koju funkciji predajemo kao parametar `relGrijeh`, te
2. broj podintervala je manji ili jednak od broja `CircusMaximus`, kojeg takoðer prenosimo funkciji kao parametar.

Kako korisnik ne bi morao svaki puta prenositi vrijednosti ova dva parametra, pridružit æemo im podrazumijevane vrijednosti. Pogledajmo cjelokupni izvorni kôd:

```
#include <iostream.h>
#include <math.h>

float Integral(double(*f)(double), double x0, double xn,
               double relGrijeh = 1.e-5,
               long CircusMaximus = 65536L);
```



Slika 7.3. Trapezno pravilo.

```
double Podintegralna(double x);

int main() {
    cout << Integral(sin, 0, 3.14159) << endl;
    cout << Integral(Podintegralna, 0, 1.) << endl;
    return 0;
}

float Integral(double(*f)(double), double x0, double xn,
               double relGrijeh, long CircusMaximus) {
    long n = 1;
    double h0 = xn - x0;
    double sumaStara = 0.;
    double sumaNova = f(x0) + f(xn);
    do {
        sumaStara = sumaNova / 2.;
        sumaNova = 0.;
        n *= 2;
        for (long i = 1; i < n; i += 2)
            sumaNova += f(h0 * i / n + x0);
        sumaNova = sumaStara + sumaNova / n;
    } while((n <= CircusMaximus) && (sumaNova != 0) &&
            (fabs(1. - sumaStara / sumaNova) > relGrijeh));
    return sumaNova * h0;
}

double Podintegralna(double x) {
    return sin(x) * sin(x);
}
```

Na poèetku kôda uključena je datoteka zaglavlja `math.h` koja sadrži deklaracije matematièkih funkcija; u programu se iz te biblioteke koriste funkcije `sin()` i `fabs()` koje vraæaju sinus, odnosno apsolutnu vrijednost argumenta. Osim toga, obratimo pažnju u gornjem kôdu na još dva detalja.

Prvo, primijetimo sufiks `L` iza podrazumijevane vrijednosti za `circusMaximus`; tim sufiksom se broj 65536 eksplicitno proglašava `long int` konstantom. Kako je taj broj veći od najvećeg mogućeg `int`, u protivnom bi se moglo dogoditi da uslijed nepredviđenih konverzija prevoditelj krivo pohrani taj broj.

Drugo, uočimo uvjet za ponavljanje `do-while` petlje. On je složen od tri uvjeta, navedenih sljedećim redoslijedom:

1. je li `n <= CircusMaximus`,
2. je li `sumaNova` razlièita od 0, te
3. da li se `sumaNova` relativno razlikuje od `sumaStara` za više od `relGrijeh`.

Redoslijed je ovdje važan i može utjecati na pravilan rad funkcije. Prvo se ispituje trivijalni uvjet (1) – upravo zbog svoje jednostavnosti i kratkoæe on je stavljen na poèetak. Tek ako je uvjet (1) ispunjen, prelazi se na uvjet (2), koji je umetnut zbog

moguæe zamke u uvjetu (3). Naime, ako se kojim sluèajem dogodi da `sumaNova` bude jednaka 0, dijeljenje u treæem uvjetu uzrokovat æe prekid programa (dijeljenje s nulom). S novododanim uvjetom (2) to se neæe dogoditi, jer u sluèaju da on nije zadovoljen, neæe se niti poèeti ispitivati treæi uvjet. Izvoðenje petlje æe se prekinuti, a èitatelju ostavljamo na razmišljanje hoæe li rezultat funkcije biti ispravan.

Radi bolje ilustracije, pogledajmo uvjet u sljedećem `if`-grananju (`log()` je funkcija koja vraća prirodni logaritam *ln* argumenta):

```
if ((x > 0) && (log(x) < 10) {
    //...
}
```

Iako funkcija *logièki-i* ima svojstvo komutacije, tj. logièki rezultat ne ovisi o tome koji je operand s lijeve, a koji s desne strane, samo æe gornji redoslijed operanada uvijek osiguravati pravilno izvoðenje gornjeg kôda. Ako nije ispunjen prvi uvjet (tj. ako je `x` negativan ili jednak 0), daljnje ispitivanje nema smisla i ono se ne provodi, tako da se uz navedeni redoslijed ne može funkciji `log()` prenijeti negativni argument.

Ekvivalentna situacija je i kada u uvjetu imamo *logički-ili*. Izvrnimo uvjet u gornjem `if`-grananju i napišimo kôd:

```
if ((x <= 0) || (log(x) < 10) {
    //...
}
```

U ovom sluèaju, èim je zadovoljen prvi uvjet (`x <= 0`), bit æe zadovoljen cijeli uvjet, tako da se daljnje ispitivanje ne provodi – opet je izbjegnuta moguænost da se funkciji `log()` prenese negativni argument.

Iz gornjeg primjera je vidljivo da je sintaksa za deklaraciju pokazivača na funkcije dosta zamršena. Ako se pokazivač na funkciju često koristi, jednostavnije je definirati novi sinonim za taj tip pomoću ključne riječi `typedef`:

```
typedef double (*pokFunkcija)(double);
```

U gornjoj deklaraciji je `pokFunkcija` pokazivaè na funkciju koja uzima jedan parametar tipa `double` i vraæa tip `dobule`. Sada ako želimo deklarirati pokazivaè na funkciju jednostavno možemo napisati

```
pokFunkcija mojaFunkcija
```

Takoðer, funkciju `Integral()` bismo mogli deklarirati ovako:

```
float Integral(pokFunkcija f, double x0, double xn,
               double relGrijeh = 1.e-5,
               long CircusMaximus = 65536L);
```

Pokazivaèe na funkciju možemo inicijalizirati tako da im dodijelimo adresu funkcije. Pri tome, slièno kao i kod polja, samo ime funkcije bez zagrada za parametre je ekvivalentno adresi funkcije:

```
double xsinx(double x);
mojaFunkcija = xsinx;
mojaFunkcija = &xsinx;  // ista stvar kao i gore
```

Funkcija se poziva preko pokazivaèa tako da se navede naziv pokazivaèa iza kojega se u zagradama specificiraju parametri. Pri tome pokazivaè se može prije poziva dereferencirati operatorom `*`, ali i ne mora (dereferenciranje se podrazumijeva):

```
mojaFunkcija(5.);
(*mojaFunkcija)(5.);    // isti poziv kao i redak iznad
```

Potpun tip pokazivaèa na funkciju određuju povratna vrijednost funkcije te broj i tip argumenata. Zbog toga nije moguæe pokazivaèu dodijeliti adresu funkcije koja ima drukèiji potpis:

```
double xy(double x, double y);
mojaFunkcija = xy;      // pogreška: nekompatibilni tipovi
```

Isto vrijedi i za međusobnu dodjelu dvaju pokazivaèa. Pokazivaèka aritmetika nema smisla niti je dozvoljena za pokazivaèe na funkcije.

Podrazumijevani parametri nisu dio potpisa funkcije, te je stoga moguće dodijeliti adresu neke funkcije pokazivaču koji ima drukčije podrazumijevane parametre:

```
void funkcija(int i = 100);

void (*pok)(int) = funkcija;    // dozvoljeno
```

Pri tome, ako sada funkciju želimo pozvati pomoæu pokazivaèa `pok` podrazumijevane parametre ne možemo koristiti:

```
pok(60);        // OK
pok();          // pogreška: pok nema podrazumijevanih
                // parametara
```

Pokazivaèu `pok` možemo èak dodijeliti drugi podrazumijevani parametar:

```
void (*pok)(int = 80) = funkcija;
pok();      // dozvoljeno te se poziva funkcija(80)
```

7.11. Funkcija `main()`

Funkcija `main()` po mnogim svojstvima odstupa od ostalih funkcija. U prvom poglavlju smo nauèili da svaki C++ program mora sadržavati (samo jednu) funkciju `main()` kojom se zapoèinje izvoðenje programa. Takoðer, valja naglasiti da se funkcija `main()` ne može pozivati. Postoje dvije moguæe varijante funkcije `main()`:

```
int main() {
    //...
}
```

koja ima praznu listu argumenata (ovaj oblik smo do sada iskljuèivo i koristili), ili

```
int main(int argc, char *argv[]) {
    //...
}
```

U ovom drugom obliku, prvi argument `argc` jest broj parametara koji se prenose programu iz radne okoline unutar koje je program pokrenut. Ako je `argc` razlièit od nule, tada se parametri koji se prenose mogu dohvatiti preko èlanova polja `argv`, od `argv[0]` do `argv[argc - 1]`. Ti èlanovi su pokazivaèi na znakovne nizove koji sadrže pojedine parametre. Tako je `argv[0]` pokazivaè na poèetak niza koji sadrži ime kojim je program pozvan ili prazan niz (""). `argv[argc]` je uvijek nul-pokazivaè.

Ilustrirajmo dohvaćanje argumenata funkcije `main()`, jednostavnim programom koji ispisuje rezultat aritmetičke operacije na dva broja, zadane kao parametra iza poziva programa:

```
#include <iostream.h>
#include <math.h>

int main(int argc, char* argv[]) {
    if (argc < 4) {
        cerr << "Nedovoljno parametara!" << endl;
        return 1;
    }
    float operand1 = atof(argv[1]);
    char operacija = *argv[2];
    float operand2 = atof(argv[3]);
    switch (operacija) {
      case('+'):
        cout << (operand1 + operand2) << endl;
        break;
      case('-'):
        cout << (operand1 - operand2) << endl;
        break;
      case('*'):
        cout << (operand1 * operand2) << endl;
        break;
```

```
      case('/'):
        cout << (operand1 / operand2) << endl;
        break;
      default:
        cerr << "Nepoznati operator " << operacija << endl;
        return 1;
    }
    return 0;
}
```

Za pretvorbu znakovnog niza u realni broj (`float`) u primjeru je korištena funkcija `atof()`, deklarirana u `math.h` datoteci.

Nazovemo li izvršni program koji se dobiva prevođenjem ovog kôda `racunaj`, tada će naredba

```
racunaj 2 / 3
```

(praznine između naziva programa i prvog operanda, odnosno između operanada i operatora su obvezatne) iz operacijskog sustava ispisati kvocijent brojeva 2 i 3 (0.666667).

Umetnemo li u gornji kôd naredbe za ispise argumenata, izvođenjem programa gornjom naredbom saznat ćemo da su argumentima pridružene sljedeće vrijednosti:

```
argc    = 4
argv[0] = "racunaj.exe"
argv[1] = "2"
argv[2] = "/"
argv[3] = "3"
argv[4] = 0
```

Prvi argument `argc` je jednak broju parametara koji slijede iza naredbe kojom je program pokrenut, uveæanom za 1. `argv[0]` jest pokazivaè na znakovni niz s imenom kojim je program pokrenut. Ponekad taj znakovni niz sadrži cijelu stazu do izvršnog programa (npr. `"C:\\knjiga.cpp\\primjeri\\racunaj.exe"`). Slijede pokazivaèi na parametre, a niz se zakljuèuje nul-pokazivaèem `argv[4]`.

Funkcija `main()` u pravilu završava naredbom `return` (koja mora sadržavati cjelobrojni parametar) koja uzrokuje uništenje svih automatskih objekata. Operacijskom sustavu se prenosi vrijednost navedena u `return` naredbi. Ako je izvođenje programa završilo ispravno, podrazumijeva se da povratna vrijednost bude 0; u protivnom se vraća neka vrijednost različita od 0 (vrijednost ovisi o operacijskom sustavu). Ako izvođenje funkcije `main()` nije zaključeno naredbom `return` (primjerice ako ju zaboravimo napisati), tada će ona skončati kao da se na kraju nalazi naredba

```
return 0;
```

Buduæi da u svakom C++ programu smije biti samo jedna `main()` funkcija, njeno se ime ne može preoptereæivati.

Ako ne želimo posebno obavještavati operacijski sustav o uspješnosti izvođenja našeg programa, funkciju `main()` možemo deklarirati tipa `void`:

```
void main() {
    // ...
}
```

U tom sluèaju se naredba `return` ne mora navesti, a prilikom povratka u operacijski sustav se vraæa vrijednost neka nedefinirana vrijednost. Ovakav oblik funkcije `main()` nije propisan standardom, ali æe ga mnogi prevoditelji bez problema prihvatiti.

7.12. Standardne funkcije

Jezik C++ nema funkcija koje su ugraðene u sam jezik, no programeru za obavljanje èitavog niza tipiènih operacija redovito na raspolaganju stoji mnoštvo funkcija u bibliotekama koje se isporuèuju zajedno s prevoditeljem. Veæina tih funkcija je obuhvaæena standardom, tako da se mogu koristiti bez bojazni po prenosivost kôda. Meðutim, uz veæinu prevoditelja se isporuèuju i biblioteke sa specifiènim funkcijama. Korištenje takvih funkcija u kôdu æe ponekad znatno olakšati pisanje programa, ali treba voditi raèuna da se takav kôd vrlo vjerojatno neæe moæi prevesti nekim drugim prevoditeljem.

Za funkcije iz priloženih biblioteka vrijede ista pravila kao i za funkcije koje sami definiramo. Između ostalog, to podrazumijeva da funkcija mora biti deklarirana prije prvog poziva. Priložene biblioteke s funkcijama su obično u *objektnom* obliku (već su prevedene) radi što bržeg povezivanja izvedbenog kôda (izvorni kôd se ne isporučuje radi zaštite autorskih prava). Stoga su deklaracije tih funkcija pohranjene u posebnim datotekama zaglavlja – da bi se funkcija iz neke biblioteke mogla pravilno koristiti, dovoljno je na početku izvornog kôda uključiti pripadajuću datoteku zaglavlja pretprocesorskom naredbom `#include`.

Demonstrirat ćemo upotrebu standardnih funkcija programom koji pretvara koordinate točke u ravnini zadane u pravokutnom koordinatnom sustavu u polarne koordinate. Taj zadatak se obavlja sljedećim formulama:

$$r = \sqrt{x^2 + y^2}$$

$$\varphi = \arctan\left(\frac{y}{x}\right)$$

Pri tome su x i y pravokutne koordinate toèke, a r i φ radijus i kut radij-vektora toèke. U kôdu æemo koristiti dvije standardne funkcije: `sqrt()` koja vraæa kvadratni korijen argumenta, te `atan2(double x, double y)` koja vraæa arkus tangensa za koordinate toèke na apscisi i ordinati. Obje funkcije su deklarirane u datoteci `math.h`

koju ukljuèujemo na poèetku kôda. Buduæi da rezultat želimo ispisati ukljuèit æemo i zaglavlje `iostream.h` u kojem su deklarirani ulazni i izlazni tok te preoptereæeni operator `<<`.

```
#include <iostream.h>
#include <math.h>

int main() {
    double x = -1;
    double y = -1;
    double r = sqrt(x * x + y * y);
    double fi = atan2(x, y);
    cout << "r =" << r << "  fi =" << fi << endl;
    return 0;
}
```

Valja uoèiti da postoji i funkcija `atan(double)` koja kao argument prihvaæa tangens kuta koji treba izraèunati. Međutim, funkcija `atan2()` je prikladnija, jer daje toèan kut u sva èetiri kvadranta (funkcija `atan()` daje rezultate samo u prvom i èetvrtom kvadrantu), te za kuteve vrlo blizu $\pi/2$ i $-\pi/2$, tj. kada je $x = 0$.

Ako se neka biblioteka zaboravi uključiti, prevoditelj će javiti pogrešku tijekom prevođenja. Uključivanje suvišnih biblioteka (tj. biblioteka iz koje nisu korištene funkcije) neće imati nikakve kobne posljedice na izvršni kôd, jedino će se sam program dulje prevoditi. Mnogi današnji povezivači (*linkeri*) provjeravaju koje funkcije se pozivaju u programu, te samo njih uključuju u izvršni kôd.

U tablici 7.1 su navedene neke od standardiziranih biblioteka te opis funkcija koji se u njima nalaze. Vjerujemo da su većini čitatelja najzanimljivije matematičke funkcije i funkcije za obradu znakovnih nizova, deklarirane u `math.h`, odnosno `string.h`. U prilogu B na kraju knjige zainteresirani čitatelj može naći popis najčešće korištenih funkcija, te njihovo kratko objašnjenje.

Tablica 7.1. Neke èešæe korištene standardne datoteke zaglavlja

Naziv	Opis
`complex.h`	Kompleksni brojevi i funkcija
`fstream.h`	C++ tokovi za datoteèni ulaz i izlaz
`float.h`	Podaci o realnim tipovima podataka
`iomanip.h`	Ulazno-izlazni manipulatori za C++ tokove
`iostream.h`	Osnovni ulazno-izlazni C++ tokovi
`limits.h`	Podaci o rasponima vrijednosti cjelobrojnih podataka
`locale.h`	Funkcija sa zemljopisno- i jezièno-specifiènim podacima
`math.h`	Matematièke funkcije
`stdarg.h`	Makro funkcije i definicije za poziv argumenata funkcija deklariranih s neodreðenim brojem argumenata (. . .)
`stdlib.h`	Rutine za konverziju, pretraživanje, sortiranje i sl.
`string.h`	Rutine za rukovanje znakovnim nizovima
`time.h`	Strukture za pohranu podataka o vremenu i datumu, te rutine za njihovu obradu

Korištenje veæine matematièkih funkcija je intuitivno jasno, tako da ih nema potrebe detaljnije opisivati. Funkcije za rukovanje znakovnim nizovima iziskuju posebnu pažnju zbog naèina na koji se oni pohranjuju. Vrlo široka primjena tih funkcija zahtjeva da im posvetimo malo više pažnje.

7.12.1. Funkcije za rukovanje znakovnim nizovima

Gotovo u svakom programu u kojem se ispisuju poruke, javlja se potreba za rukovanjem znakovnim nizovima, na primjer kopiranjem ili povezivanjem znakovnih nizova. Standardni matematièki operatori nisu definirani za znakovne nizove, tako da se (za razliku od nekih drugih jezika) u programskom jeziku C++ ne mogu vršiti jednostavna pridruživanja ili zbrajanja znakovnih nizova. Primjerice, pokušaj generiranja dva jednaka znakovna niza pomoæu operatora pridruživanja u sljedeæem primjeru neæe dati oèekivani rezultat:

```
char *zn1 = "Dora";
char *zn2 = zn1;            // preusmjerava pokazivač
```

Gornji odsjeèak æe samo preusmjeriti pokazivaè `zn2` na poèetak znakovnog niza `zn1`. Buduæi da se pritom neæe stvoriti novi znakovni niz, svaka promjena na nizu `zn2` æe se odraziti i na nizu `zn1`, što vjerojatno nije ono što bismo oèekivali od operatora pridruživanja. Takoðer, jednostavnim operatorom zbrajanja ne možemo nadovezati dva znakovna niza:

```
char *zn1 = "Dora ";
char *zn2 = "Krupićeva";
char *zn3 = zn1 + zn2;      // pogreška: nepravilno
                            //  zbrajanje pokazivača
```

Za operacije na znakovnim nizovima programeru je na raspolaganju èitav niz funkcija i makro naredbi deklariranih u datoteci `string.h`. U njoj su definirane funkcije za kopiranje nizova, nadovezivanje dvaju nizova, pretraživanje nizova, usporedbu dvaju nizova, odreðivanje duljine niza, pretvorbe velika-mala slova i slièno. Upotreba najkorisnijih funkcija pokazana je u sljedeæem kôdu:

```
#include <iostream.h>
#include <string.h>

int main() {
    char *prvi = "mali";
    char *drugi = "princ";
    char *praznina = " ";

    int ukupnaDuljina = strlen(prvi) + strlen(praznina) +
                        strlen(drugi);
    char *oba = new char[ukupnaDuljina + 1];
    strcpy(oba, prvi);
```

```
    strcat(oba, praznina);
    strcat(oba, drugi);
    int usporedba = strcmp(oba, prvi);
    if (usporedba) {
        if (usporedba > 0)
            cout << "\"" << oba  << "\" je veći od \""
                 << prvi << "\"" << endl;
        else
            cout << "\"" << prvi << "\" je veći od \""
                 << oba << "\"" << endl;
    }
    else
        cout << " \"" << prvi << "\" i \"" << oba
             << "\" su jednaki" << endl;
    return 1;
}
```

Funkcija `strlen()` izraèunava duljinu znakovnog niza. Ne smije se zaboraviti sljedeæe:

> Funkcija `strlen()` kao rezultat vraæa duljinu niza bez zakljuènog nul-znaka. Ukupno zauzeæe memorije jest za jedan znak veæe.

Zato je bilo neophodno prilikom alokacije prostora za znakovni niz `oba` zbroju duljina znakovnih nizova `prvi`, `drugi` i `praznina` pridodati 1.

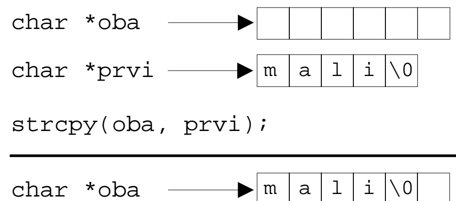
Zadatak. *Razmislite i provjerite što će ispisati sljedeća naredba:*

```
cout << strlen("\"101\"") << endl;
```

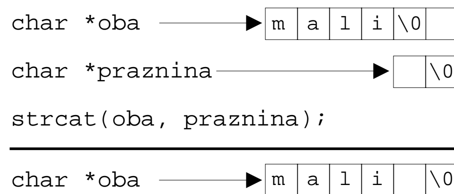
Funkcija `strcpy()`, èija je deklaracija oblika

```
char *strcpy(char *kamo, const char *odakle);
```

preslikava znakovni niz (zajedno sa zakljuènim nul-znakom) na koji pokazuje drugi argument, na lokaciju na koju pokazuje prvi argument (slika 7.4). Povratna vrijednost



Slika 7.4. Djelovanje funkcije `strcpy()`.



Slika 7.5. Djelovanje funkcije `strcat()`.

funkcije je pokazivaè na odredište preslikavanja (taj se podatak rijetko koristi). Iz deklaracije je oèito da niz koji se preslikava ostaje nepromijenjen. Međutim, buduæi da se preslikavanjem mijenja sadržaj memorije na koju pokazuje prvi argument, neophodno je osigurati dovoljan prostor (ukljuèujuæi i zakljuèni nul-znak) u koji æe se izvorni niz preslikati. Na primjer, kôd

```
char *replika;
strcpy(replika, "terminator");      // opasno!
```

æe biti preveden, ali æe prilikom izvođenja funkcija `strcpy()` preslikati znakovni niz u dio memorije koji ne pripada (samo) nizu `replika`, što æe zasigurno poluèiti neželjene efekte.

Zadatak. *Razmislite što ne valja u sljedećem kôdu:*

```
char broj[3];
strcpy(broj, "pet");        // potencijalna opasnost!
```

Ako niste sigurni u odgovor, isprobajte ovaj kôd dodajuæi naredbu za ispis niza `broj` nakon poziva funkcije `strcpy()`. Sam primjer je veæ sam za sebe potencijalno "opasan", ali èak i ako se preživi kopiranje, imat æemo problema ako niz `broj` kasnije pokušamo preslikati u neki drugi niz (razmislite zašto). Ovakva pogreška je vrlo èesta u C++ poèetnika.

Funkcija `strcat()` se koristi za nadovezivanje sadržaja dva znakovna niza. Ona ima deklaraciju sljedećeg oblika:

```
char *strcat(char *sprijeda, const char *straga);
```

Djeluje tako da nadovezuje znakovni niz `straga` na znakovni niz `sprijeda`. Prilikom preslikavanja niza `straga` preslikava se i njegov zakljuèni nul-znak (vidi sliku 7.5).

Kao i kod funkcije `strcpy()`, programer mora paziti je li za rezultantni niz (zajedno sa zaključnim nul-znakom) osiguran dovoljan prostor, kako se preslikavanjem ne bi zašlo u memorijski prostor rezerviran za druge varijable ili izvedbeni kôd. Stoga nije na odmet još jednom naglasiti:

Prije preslikavanja nizova funkcijama `strcpy()` i `strcat()` (ali i sliènima), obvezatno treba alocirati dovoljan prostor za odredišni niz, zajedno s njegovim zakljuènim nul-znakom.

Zadatak. *Zašto sljedeća naredba nije izvediva:*

```
strcat("Draš", "Katalenič");            // pogreška
```

Zadatak: *Što će biti ispisano izvođenjem sljedećeg kôda:*

```
char deponij[25];
strcpy(deponij, "kanal");
strcpy(deponij + 3, "tata u F-duru");
cout << deponij << endl;
```

Funkcija `strcmp()` je deklarirana kao

```
int strcmp(const char *prviNiz, const char *drugiNiz);
```

Ona obavlja abecednu usporedbu sadržaja dvaju niza, znak po znak sve dok su odgovarajuæi znakovi u oba niza međusobno jednaki ili dok ne naleti na zakljuèni nul-znak u jednom od nizova. Ako je prvi niz veæi (u abecednom slijedu dolazi iza drugoga), funkcija kao rezultat vraæa cijeli broj veæi od nule; ako su nizovi potpuno jednaki vraæa se nula, a ako je drugi niz veæi, vraæa se negativni cijeli broj.

Funkcija `strcmp()` razlikuje velika i mala slova. Preciznije, usporedba se obavlja prema ASCII nizu znakova u kojem sva velika slova prethode malim slovima. Zbog toga će usporedba nizova u sljedećem primjeru:

```
char *prvi = "mama";
char *drugi = "Tata";
cout << strcmp(prvi, drugi) << endl;
```

javiti da po abecednom slijedu niz `"Tata"` prethodi nizu `"mama"`, a to vjerojatno nije ono što bi korisnik oèekivao. Da bi se to izbjeglo, valja koristiti funkciju `strlwr()`, koja sva slova u nizu pretvara u mala, omoguæavajuæi korektnu abecednu usporedbu.

Kako bi se pojednostavnilo rukovanje znakovnim nizovima, pogodno je znakovni niz opisati pomoću objekta. Za taj objekt tada možemo definirati operatore koji bitno pojednostavnjuju gore navedene operacije. Tako se usporedba može obavljati standardnim poredbenim operatorima (`<`, `<=`, `==`, `!=`, `>=`, `>`), dodjela operatorom `=` i slično. Štoviše, dio C++ standarda je i klasa `String` koja već posjeduje svu navedenu funkcionalnost. Vjerujemo da će čitatelju to biti puno jasnije nakon što upozna osnove objektnog programiranja u narednim poglavljima.

7.12.2. Funkcija `exit()`

Složeni programi se sastoje od mnoštva funkcija koje se pozivaju iz funkcije `main()` ili iz neke druge korisnièki definirane funkcije. Struktura meðusobnog pozivanja funkcija u takvim programima nerijetko može biti vrlo složena. Teškoæe nastupaju ako tijekom izvoðenja programa nastupi pogreška tako da je potrebno trenutaèno prekinuti izvoðenje programa. Regularni povratak iz funkcija u takvim situacijama može biti vrlo mukotrpan ili èak nemoguæ.

Ilustrirajmo to primjerom programa koji poziva funkciju za računanje prirodnog logaritma. Kao što znamo, logaritamska funkcija definirana je samo za pozitivne brojeve veće od nule. Što, međutim, napraviti ako se kojim slučajem u tijeku proračuna toj funkciji proslijedi nula ili negativan broj? Kako će funkcija za računanje logaritma dati do znanja pozivajućoj funkciji da je rezultat nedefiniran, kad ona inače kao rezultat može vratiti bilo koji pozitivni ili negativni broj? Nije nam ostala na raspolaganju niti jedna "specijalna" vrijednost koja bi signalizirala pozivajućoj funkciji neispravnost rješenja.

Jedno rješenje je trenutačno prekinuti izvođenje programa, za što možemo iskoristiti funkciju `exit()` deklariranu u biblioteci `stdlib.h`:

```
void exit(int status);
```

Cjelobrojni argument `status` te funkcije predaje se operacijskom sustavu kao rezultat programa, ekvivalentno onome koji se vraæa u naredbi `return` na kraju glavne funkcije. Pogledajmo kako bismo mogli napisati kôd za navedeni program:

```
#include <iostream.h>
#include <math.h>
#include <stdlib.h>

// fja prirodni logaritam
double ln(double x) {
    if (x <= 0) exit(1);        // poziv funkcije exit()
    return log(x);
}

int main() {
    cout << ln(2) << endl;
    cout << ln(0) << endl;      // ciao da "program main"
    cout << ln(5) << endl;
    return 0;
}
```

U funkciji `ln()` poziva se standardna funkcija `log()` iz matematièke biblioteke. Prije poziva te funkcije provjerava se argument i ako je on manji ili jednak nuli, poziva se funkcija `exit()`. Zbog toga æe se prilikom drugog poziva funkcije `ln()` (argument jednak nuli) prekinuti izvoðenje programa, a treæi poziv funkciji `ln()` neæe niti biti upuæen.

Pozivom funkcije `exit()` se zatvaraju sve datoteke koje su bile otvorene tijekom izvođenja programa i uništavaju svi statički objekti, redoslijedom obrnutim od redoslijeda njihova stvaranja.

7.13. Predlošci funkcija

U poglavlju o preoptereæenju imena funkcija vidjeli smo kako se isti identifikator može koristiti za razlièite definicije funkcija – preoptereæene funkcije su imale jednaka imena, ali su se razlikovale prema argumentima. Preoptereæenje funkcija omoguæava svođenje na zajednièko ime sliènih operacija nad razlièitim tipovima podataka. Najoèitiji primjer za to je bila funkcija za potenciranje `mocnica()` koju smo razlièito definirali za cjelobrojne, odnosno decimalne potencije (primjer na str. 193).

Međutim, često su algoritmi za različite tipove podataka potpuno identični. Ako se takav problem rješava preopterećenje, potrebno je definirati funkciju za svaki mogući tip argumenta. Funkciju `kvadrat()` treba ponoviti za sve moguće tipove argumenata unatoč činjenici da one sadrže potpuno isti kôd:

```
inline int kvadrat(int x) {
    return x * x;
}

inline float kvadrat(float x) {
    return x * x;
}

inline double kvadrat(double x) {
    return x * x;
}
```

U ovakvim sluèajevima je umjesto preoptereæenih funkcija daleko praktiènije koristiti *predloške funkcija* (engl. *function templates*) koji omoguæavaju da se jednom definirani kôd prevede više puta za razlièite tipove argumenata. Sintaktièki, deklaracija predloška funkcije ima oblik:

```
template <argument_predloška,...> deklaracija_funkcije;
```

Svaki argument predloška se sastoji od ključne rijeèi `class` i imena argumenta, te se svaki argument mora pojaviti u listi parametara funkcije. Ako je deklaracija funkcije ujedno i njena definicija, tada je deklaracija predloška ujedno i definicija predloška.

Pogledajmo kako bismo funkciju `kvadrat()` napisali pomoću predložaka i time izbjegli višestruko pisanje kôda:

```
template <class Tip>
inline Tip kvadrat(Tip x) {
```

```
        return x * x;
    }
```

Ovom deklaracijom/definicijom je funkcija `kvadrat()` parametrizirana, tako da se tip argumenta i tip funkcije mogu prema potrebi mijenjati. Prilikom prevoðenja, prevoditelj æe za svaki poziv funkcije `kvadrat()` nadomjestiti `Tip` odgovarajuæim tipom podatka. Konkretno, sljedeæi pozivi æe uzrokovati generiranje triju razlièitih oblika funkcije `kvadrat()`:

```
    float f = 1.2;
    int i = 9;
    double d = 3.14159;
    cout << kvadrat(f) << endl;      // float kvadrat(float)
    cout << kvadrat(i) << endl;      // int kvadrat(int)
    cout << kvadrat(d) << endl;      // double kvadrat(double)
    cout << kvadrat(1.4) << endl;    // double kvadrat(double)
```

Uoèimo da se u posljednjem pozivu realna konstanta implicitno tretira kao podatak tipa `double`, te se shodno tome poziva funkcija s takvim tipom argumenta.

Argumenti predloška se mogu koristiti višekratno kao argumenti u deklaraciji funkcije. Primjerice, sasvim općenitu funkciju `zamijeni()` kojoj je zadaća zamjena sadržaja dvaju objekata, možemo deklarirati i definirati kao

```
    template <class Tip>
    void zamijeni(Tip &prvi, Tip &drugi) {
        Tip privremeni = prvi;
        prvi = drugi;
        drugi = privremeni;
    }
```

U ovom poglavlju su predlošci funkcija obraðeni samo informativno. Kako upotreba predložaka funkcija posebno dobiva na znaèaju uvoðenjem klasa i predložaka klasa, predlošci funkcija æe vrlo detaljno biti obraðeni u poglavlju 10.

7.14. Pogled na funkcije “ispod haube”

Programeri koji programe pišu u nekom višem programskom jeziku, kakav je i jezik C++, vrlo rijetko trebaju znati išta o organizaciji i radu raèunala. Prevoditelj i povezivaè æe (ponekad èak uspješno) prevesti njihov program pisan u jeziku razumljivom ljudima-programerima, u strojni kôd, jedini jezik koji procesor u raèunalu “razumije”. U tom strojnom jeziku ne postoje objekti, konstante i funkcije, veæ samo memorijske adrese. Unatoè èinjenici da golema veæina programera gotovo nikad neæe morati išta znati o organizaciji memorije i o tome kako se strojni kôd izvodi, za cjelovito razumijevanje problematike je ipak dobro razjasniti neke osnovne stvari. Èitatelji koji su familijarni s tim pojmovima ili to znanje smatraju suvišnim, slobodno mogu preskoèiti ovaj odjeljak.

Prilikom pokretanja nekog programa, izvedbeni kôd se s vanjske jedinice (diska, diskete, CD-ROM-a) učitava u radnu memoriju računala (RAM, engl. *random access memory*) te se na njega prenosi izvođenje. Prilikom učitavanja i pokretanja programa, dio memorije se iskorištava za smještaj samog izvedbenog kôda, a dio se koristi za pohranjivanje podataka. Pritom je dio za pohranjivanje podataka podijeljen na tri praktički neovisna područja: *podatkovni segment* (engl. *data segment*), *hrpu* (engl. *heap*) i *stog* (engl. *stack*). Podatkovni segment (može ih biti i više) je područje memorije u kojem su smještene globalni i statički objekti. Hrpa je dio memorije koji je dostupan izvedbenom kôdu u bilo kojem trenutku izvođenja – u taj dio se pohranjuju dinamički alocirani podaci. Stog je dio memorije na koji se pohranjuju lokalni podaci, dostupni samo funkciji čiji se kôd trenutačno izvodi (osim lokalnih podataka na stog se pohranjuju još i podaci potrebni prilikom poziva i povrata iz funkcija, no to ćemo još kasnije razjasniti).

Naravno da se sâm program ne može izvoditi bez "mozga" cijele akcije – *središnje procesorske jedinice* (engl. *Central Processing Unit, CPU*) ili kraće (*mikro*)*procesora*. On interpretira naredbe strojnog kôda i poduzima odgovarajuće akcije: dohvaća podatke iz memorije, obavlja operacije s njima i pohranjuje ih natrag u memoriju. Pritom on koristi svoju "internu" memoriju – *registre* (engl. *registers*). Radi se o vrlo skučenoj memoriji u koju stanu svega osnovni podaci (primjerice *int* i *float*), ali ona ionako služi samo za privremeno pohranjivanje podataka tijekom obrade. Osim nekoliko registara za podatke koji se obrađuju, za pravilan rad programa (a i cijelog računala) bitna su dva registra: *pokazivač instrukcija* (engl. *instruction pointer*) te *pokazivač stoga* (engl. *stack pointer*). Pokazivač instrukcija u svakom trenutku sadrži adresu memorije na kojoj se nalazi sljedeća instrukcija koju procesor mora izvesti. Procesor prilikom dohvaćanja instrukcije prvo očita sadržaj pokazivača instrukcija, čiji sadržaj upućuje na mjesto gdje se nalazi sljedeća izvedbena naredba. Pomoću adrese iz pokazivača instrukcija procesor očitava naredbu iz dijela memorije u kojoj je smješten izvedbeni kôd te pristupa njenom izvođenju. Istovremeno se poveća sadržaj pokazivača instrukcija tako da on pokazuje na sljedeću instrukciju programa.

Naiđe li se tijekom izvođenja programa na naredbu za skok na neku drugu memorijsku adresu, procesor će adresu sadržanu u instrukciji ubaciti u pokazivač instrukcija, čime će se izvođenje programa automatski prebaciti na željenu adresu. Teškoće nastupaju kod poziva funkcija, jer se nakon izvođenja funkcije valja vratiti na prvu naredbu iza naredbe koja je pozvala funkciju. Očito je da prilikom poziva funkcije treba pohraniti adresu naredbe na koju se procesor po povratku iz funkcije mora vratiti. Valja primijetiti da to pohranjivanje adrese nije trivijalno, budući da se unutar pozvane funkcije može pozvati jedna ili više drugih funkcija, a za svaki od tih poziva treba pohraniti pripadajuće povratne adrese.

Vjerojatno je svakom jasno da će prilikom uzastopnih poziva funkcija jedne unutar druge, prvo biti potrebna adresa vezana uz zadnji poziv funkcije, a tek potom ostalih funkcija, redoslijedom obrnutim od redoslijeda pozivanja funkcija. Zbog toga se povratne adrese pohranjuju na stog – posebni dio memorije u kojem se podaci dohvaćaju obrnutim redoslijedom od redoslijeda kojim su tamo ostavljani. To znači da će podatak koji je na stogu bio ostavljen posljednji, biti dohvatljiv kao prvi, a podatak koji je bio ostavljen prvi bit će dohvatljiv posljednji. Takva struktura se često označava

kraticom *LIFO*, od engl. *Last In, First Out* – posljednji unutra, prvi van. Pokazivač stoga pokazuje na zadnji umetnuti podatak, tj. na "vrh" stoga. Kada procesor u strojnom kôdu naleti na poziv funkcije, on će pohraniti adresu iz pokazivača instrukcija (to je adresa naredbe na koju se izvođenje mora vratiti po povratku iz funkcije) na adresu na koju pokazuje pokazivač stoga, te će sadržaj pokazivača stoga povećati. Ako se unutar funkcije pozove neka druga funkcija, postupak će se ponoviti – povratne adrese se postepeno slažu u stog, a pokazivač stoga se pritom uvećava. Pri povratku iz zadnje funkcije, preko pokazivača stoga se očita posljednja povratna adresa, te se pokazivač smanji. Postupak se ponavlja pri svakom povratku iz funkcija u nizu, te se stog "prazni".

Budući da su podaci na stogu dohvatljivi isključivo preko pokazivača stoga, treba voditi računa o redoslijedu punjenja i pražnjenja stoga. Srećom, sve operacije vezane uz stog sređuje prevoditelj, oslobađajući programera od prizemnog posla. Međutim, ono o čemu autor programa mora voditi računa jest da se taj stog ne prepuni, jer će to izazvati prekid rada programa. Na prvi pogled programer nema ni tu velikog utjecaja budući da veličinu stoga određuje prevoditelj. Istina, svaki prevoditelj ima mogućnost da se veličina stoga za neki program poveća, ali to nije univerzalni lijek. Ipak, iz gornjih razmatranja je jasno da mnogobrojni uzastopni pozivi funkcija opterećuju stog. To se posebno odnosi na glomazne rekurzije, gdje broj poziva funkcije može postati vrlo velik. Uz to valja znati da se osim povratnih vrijednosti, na stog pohranjuju i argumenti funkcija (to je uostalom i razlogom zašto se polja ne prenose po vrijednosti) i lokalni objekti. Ako su objekti koji se prenose funkcijama veliki, stog će vrlo brzo biti ispunjen do vrha.

8. Klase i objekti

Ja ne vjerujem u klasne razlike, ali se srećom mišljenje mog sobara po tom pitanju razlikuje.

Marc, strip u londonskom "The Times"

Jezik C++ s kakvim smo se do sada upoznali ne razlikuje se znatno od bilo kojeg drugog proceduralnog jezika. Ono što ovaj jezik èini posebno istaknutim jest moguænost definiranja novih korisnièkih tipova – klasa, èime se uvodi koncept objektnog programiranja.

U nastavku će biti objašnjen način na koji se klase deklariraju. Bit će objašnjeni pojmovi kao što su: podatkovni i funkcijski članovi, prava pristupa, konstruktor, destruktor. Također, bit će razjašnjeni ključni koncepti objektnog programiranja, kao što su javno sučelje i implementacija klase.

8.1. Kako prepoznati klase?

Jezik C++ uvodi kao znaèajnu konceptualnu promjenu u naèinu programiranja novi tip podataka: *klase* (engl. *class*). One su temelj objektno orijentiranog pristupa programiranju. Sama ideja uvođenja objekata u programiranje, iako revolucionarna, došla je zapravo analiziranjem naèina na koji funkcionira stvarni svijet. Ako se malo pomnije udubimo u tokove podataka oko nas, doæi æemo do zakljuèka da se mnoge stvari mogu vrlo jednostavno modelirati pomoæu objekata. *Objekt* (engl. *object*) je naziv za skup svojstava koja možemo objediniti u smislenu cjelinu. Pravila koja propisuju od èega je pojedini objekt sagraðen te kakva su njegova svojstva nazivaju se klasama. Vrlo je važno uoèiti razliku između klase i objekta: klasa je samo opis, dok je objekt stvarna, konkretna realizacija napravljena na temelju klase.

Objekti međusobno izmjenjuju informacije i traže jedan od drugoga usluge. Pritom okolina objekta ništa ne mora znati o njegovom unutarnjem ustrojstvu. Svaki objekt ima *javno sučelje* (engl. *public interface*) kojim se definira njegova suradnja s okolinom. Ono određuje koje informacije objekt može dati te u kojem formatu. Također su definirane i sve usluge koje objekt može pružiti.

Interno se objekt sastoji od niza drugih objekata i interakcija među njima. Način na koji se reprezentacija objekta ostvaruje jest *implementacija objekta* (engl. *implementation*). Ona je najčešće skrivena od okoline kako bi se osigurala konzistentnost objekta. Klasa se, dakle, sastoji od opisa javnog sučelja i od implementacije. To objedinjavanje javnog sučelja i implementacije naziva se *enkapsulacija* (engl. *encapsulation*).

Kada je klasa jednom definirana, može se pomoću nje konstruirati neograničen broj objekata koji se zatim mogu koristiti. Kako sve stvari koje nas okružuju imaju svoj početak i kraj, i život jednog objekta ima svoj početak i svršetak. Kada se objekt stvori, potrebno mu je dati početni oblik. Postupak koji opisuje kako će se to napraviti dan je u specifikaciji klase. Funkcija koja inicijalizira objekt naziva se *konstruktor* (engl. *constructor*). Kada objekt više nije potreban, posebna funkcija klase pod imenom *destruktor* (engl. *destructor*) uredno će uništiti objekt.

Kako bismo ilustrirali činjenicu da objekti nisu samo tvorevine računalskih stručnjaka koji smišljaju načine kojima će zagorčavati programerima život, evo primjera koji pokazuje da se čitav život uistinu odvija kroz interakciju objekata.

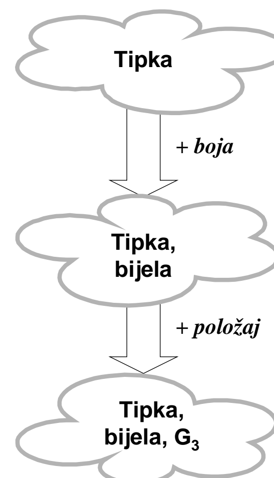
Zamislite maestra koji na klaviru izvodi Beethovenovu "Mjesečevu sonatu". Takva naoko jednostavna i svakidašnja situacija zapravo je primjer jedinstvene suradnje nekoliko stotina objekata. Kao prvo, tu je muzičar koji izvodi glazbu, zatim tu su klavir i note po kojima se svira. Pokušajmo pobliže identificirati klase na temelju kojih su stvoreni ti objekti. Muzičar je primjerak klase ljudi. Definiciju čovjeka ćemo prepustiti filozofima i teolozima, koji se doduše još danas spore o tome što zapravo čini čovjeka i kakvo je njegovo javno sučelje, no s klavirom je stvar puno jasnija. Uzmimo da se radi o *Steinwayu*. On je samo predstavnik opće klase klavira te čim identificiramo *Steinway* kao klavir odmah znamo ugrubo kako on izgleda. Osnovna svrha klavira je muziciranje (iako će se možda mišljenja ponekih sretnih vlasnika ovdje razlikovati). Njegovo javno sučelje je klavijatura pomoću koje korisnik ostvaruje svrhu klavira. Javno sučelje klavira sastoji se od daljnjeg skupa objekata: tipki. Općenita klasa tipki definira da svaka tipka može biti pritisnuta te da pritisak na nju proizvodi točno određeni ton. Nizovi tonova koje klavir proizvodi također spadaju u javno sučelje klavira.

No, kako klavir ostvaruje svoje javno sučelje? Da bismo doznali odgovor na to pitanje, moramo zaviriti pod poklopac – u implementaciju. To je područje koje ne spada u javno sučelje jer je teško zamisliti da bi imalo ozbiljan muzičar pokušao odsvirati "Mjesečevu sonatu" direktno koristeći implementacijske detalje, ručno udarajući batićima po žicama. No i unutar implementacije se i dalje nalazimo u svijetu sačinjenom od objekata. Unutrašnjost klavira sastoji se od žica, koje opet mogu pripadati određenoj klasi žica, te od batića. Funkcioniranje implementacije zasniva se na interakciji žica i batića. Tako bismo mogli nastaviti s nabrajanjem rastavljajući pojedine objekte na sve manje i manje detalje, do kvarkova i gluona. Što je dalje u implementaciji prepustit ćemo fizičarima i njihovim akceleratorima čestica.

Valja uočiti da neki drugi objekti ipak mogu imati pristup implementaciji klavira. Zamislimo da klavir nije dobro ugođen. Loše sviranje klavira u tom slučaju je posljedica pogreške u implemetaciji. Muzičar će, nakon što preko njemu dostupnog javnog sučelja ustanovi da klavir ne radi kako treba, vjerojatno vrlo iznerviran, dignuti ruke i pozvati ugađača klavira ili što bi to purgeri rekli, "klavir-štimera". Muzičar ništa ne zna o ustrojstvu klavira, te bi svojim miješanjem u implementaciju vjerojatno samo unio dodatni nered. No klavir-štimer je osoba koja ima pristup unutrašnjosti (slika 8.1), te se u terminologiji C++ jezika ona naziva *prijateljem klase* klavira (engl. *friend of a class*).

Slika 8.1. Objektni preludij za klavir i klavir-štimera u C++ duru

Obratimo nadalje pažnju na fenomen koji pokazuje da se jedan objekt može klasificirati postupno, od jednostavnijeg prema složenom. Možemo uvesti opæu klasu tipki koja definira tipku kao ploèicu koja se može pritisnuti i time proizvesti ton (slika **8.2**). Nadalje, pojedina se tipka može klasificirati kao crna ili bijela. Na kraju, može se uvesti klasa za svaku pojedinu tipku, na primjer klasa za tipku G_3. Ona definira objekt kao tipku bijele boje, odreðuje njenu poziciju na klavijaturi, te toèno definira ton koji se dobije pritiskom na nju. Ovime smo izveli hijerarhiju klasa koja izgleda kao stablo: u korijenu stabla se nalazi opæenita klasa tipki koja definira opæa svojstva tipke. Klase crnih i bijelih tipaka se izvode iz opæenite klase. To izvoðenje se obavlja tako da se zadrže opæa svojstva tipke koja se proširuju dodatnim svojstvima, primjerice bojom tipke. Neka svojstva osnovne klase, kao što je velièina tipke, redefiniraju se u novim klasama. Postupak kojim se izvodi nova klasa zove se *nasljeđivanje* (engl. *inheritance*) i jedan je od kljuènih elemenata objektno orijentiranog dizajna.



Slika 8.2. Nasljeđivanje

Tipka G_3 se može promatrati kao instanca klase G_3. No njena specifična svojstva koja ju svrstavaju u klasu G_3 se po potrebi mogu zanemariti te se tu tipku može promatrati sa stajališta koje je definirano klasom bijelih tipki. Pri tome se zanemare svojstva kao što je precizna definicija tona koji se proizvodi pritiskom na tipku pa se tipka promatra samo sa stajališta boje. Može se ići i dalje, te se ista tipka može promatrati sa stajališta osnovne klase, dakle, samo kao objekt koji se može pritisnuti i

time proizvesti neki ton. Mogućnost promatranja objekta kroz različite klase u stablu nasljeđivanja se zove *polimorfizam* (engl. *polymorphism*) i treće je vrlo važno svojstvo svih objektno orijentiranih jezika.

Nakon što smo definirali osnovna svojstva javnog sučelja klavira, uočavamo da smo iz cijele priče izostavili važne elemente javnog sučelja, a to su pedale za produljenje tona. Objektno orijentirano programiranje je vrlo koristan alat koji može značajno povećati produktivnost programera, ali prije nego što se prijeđe na samo kôdiranje klase, za njegovu uspješnu primjenu potrebno je više pripremnog rada. Apstraktni model klavira je potrebno detaljno proučiti prilikom definicije klase kako bi se precizno opisalo njegovo javno sučelje i izbjegle pogreške u projektiranju. Na primjer, ako na početku zamislimo klavir tako da mu tipke smjestimo pod poklopac, teško ćemo kasnije uspostaviti zadovoljavajuće javno sučelje. Ispravljanje ovakvih fundamentalnih konstrukcijskih pogrešaka u C++ jeziku je znatno složenije nego u klasičnim proceduralnim jezicima.

Promotrimo ukratko akcije koje je potrebno provesti prilikom konstrukcije pojedinog objekta. I dok opet nailazimo na probleme definiranja konstruktora klase muzičara, odnosno ljudi, s klavirima je stvar jasnija. Kada se klavir napravi, potrebno je dovesti njegove implementacijske detalje u početno stanje, odnosno pozvati klavir-štimera koji će ga odmah ugoditi. Ta akcija se definira kao konstruktor klase klavira. Vjerujem da će maštovit čitatelj sam pronaći mnogo ilustrativnih primjera za proceduru koja uništava klavir kada on više nije potreban, odnosno njegov destruktor. Postupak koji će biti izabran kao najprikladniji vjerojatno ovisi o tome koliko je (ne)rado dotični čitatelj išao u muzičku školu.

Tri osnovna svojstva objektno orijentiranih jezika – enkapsulacija, nasljeđivanje i polimorfizam vrlo su uspješno ostvarena u C++ jeziku. Program se gradi od niza objekata koji međusobno surađuju, baš kao u našem primjeru s klavirom. Pri tome se programer mora voditi određenim formalizmom koji nameće sam jezik, no osnovna bit modeliranja pomoću klasa je dosljedno provedena kroz sve elemente jezika.

8.2. Deklaracija klase

U C++ terminologiji klasa nije ništa drugo nego poseban tip podataka koji je potrebno deklarirati prije poèetka korištenja. Deklaracija se sastoji od dva osnovna dijela: zaglavlja i tijela. Zaglavlje se sastoji od kljuène rijeèi `class` iza koje slijedi naziv klase, a tijelo slijedi iza zaglavlja i omeðeno je parom vitièastih zagrada:

```
class naziv_klase { // ovo je zaglavlje
    // ovdje dolazi tijelo
};
```

Klasu klavira možemo deklarirati ovako:

```
class Klavir {
    // ...
```

Deklaracija klase se uvijek završava znakom `;` (toèka-zarez).

```
};
```

Naziv klase je identifikator koji mora biti jedinstven u važeæem *podruèju imena* (engl. *name scope* - bit æe objašnjeno kasnije). Objekti se mogu stvoriti tako da se navedu ime klase iza kojeg se navedu imena objekata odvojena zarezima:

```
Klavir Steinway, Petroff;
```

U ovom primjeru definirana su dva objekta `Steinway` i `Petroff` klase `Klavir`. Tijelo klase može sadržavati podatkovne èlanove, funkcijske èlanove, deklaracije ugniježðenih klasa i specifikacije prava pristupa.

8.2.1. Podatkovni èlanovi

Svaka klasa može sadržavati podatkovne èlanove. Evo primjera klase koja može biti dio grafièkog suèelja i definira objekt `Prozor`:

```
class Prozor {
    int koordX1, koordY1, koordX2, koordY2;
    char *naziv;
    Prozor *vlasnik;
};
```

Ova klasa se sastoji od èetiri cjelobrojna èlana koji pamte koordinate gornjeg lijevog i donjeg desnog kuta prozora, zatim od pokazivaèa na naziv prozora te od pokazivaèa na neki drugi objekt klase `Prozor`. Taj pokazivaè pokazuje na objekt `Prozor` koji je vlasnik promatranog prozora.

Svi su tipovi dozvoljeni prilikom deklaracije klase, s time da oni moraju biti deklarirani do tog mjesta u programu. Nije moguće deklarirati kao član klase objekt klase koja se upravo definira, na primjer:

```
class Prozor {
    int koordX1, koordY1, koordX2, koordY2;
    char *naziv;
    Prozor vlasnik; // neispravno
};
```

Ovakva rekurzivna deklaracija definira klasu `Prozor` koji sadrži objekt klase `Prozor` koji sadrži objekt klase `Prozor` koji sadrži... Oèito se definicija proteže u beskonaènost pa stoga takav objekt nije moguæe prikazati u (još uvijek) konaènoj memoriji raèunala.

Klasa ne može sadržavati samu sebe, ali može sadržavati pokazivač ili referencu na objekt iste klase.

Klasa može sadržavati pokazivaè ili referencu na element iste klase, kao u prvom primjeru.

Ponekad je potrebno napraviti klasu koja æe sadržavati pokazivaè na drugu klasu, dok æe ta druga klasa sadržavati pokazivaè na poèetnu klasu. Proširimo primjerice klasu `Prozor` pokazivaèem na *izbornik* (engl. *menu*) koji je povezan s prozorom. Da bi klasa izbornika funkcionirala, mora sadržavati pokazivaè na prozor koji posjeduje izbornik. To se može napraviti tako da se samo definira ime klase izbornika, zatim se definira klasa `Prozor`, a potom se definira klasa `Izbornik` sa svim pripadajuæim èlanovima. Takva vrsta deklaracije zove se *deklaracija unaprijed* (engl. *forward declaration*). Evo primjera:

```
class Izbornik; // ovo je deklaracija imena unaprijed

class Prozor {
    // ovo je deklaracija klase Prozor
    Izbornik *pokIzbornik; // pozivanje na deklaraciju
                           // unaprijed
    // ...
};

class Izbornik {
    // ovo je potpuna definicija klase Izbornik gdje se
    // unaprijed definiranom imenu dodaju detalji
    Prozor *pokProzor;
    // ...
};
```

Kada je neka klasa samo deklarirana unaprijed, mogu se definirati iskljuèivo reference i pokazivaèi na objekte klase, ali ne i objekti klase. Naravno, u programu koji slijedi iza potpune definicije klase mogu se definirati i objekti te klase. To znaèi da bi pokušaj deklariranja objekta `Izbornik` u sklopu klase `Prozor` izazvao pogrešku prilikom prevoðenja.

8.2.2. Dohvaæanje èlanova objekta

Kako bi èlanovi objekta bili od koristi, mora postojati naèin kojim okolni svijet može pristupiti pojedinom èlanu objekta. Za to se koriste *operatori za pristup èlanovima* (engl. *member selection operators*).

Prije nego što pristupimo članu nekog objekta, moramo znati čijim članovima želimo pristupati. Neki objekt se referira pomoću naziva objekta, reference ili pokazivača na objekt.

Kada objekt identificiramo pomoću naziva objekta ili reference, za pristup članovima se koristi operator `.` (točka), tako da se s lijeve strane operatora navede naziv objekta ili referenca, a s desne strane ime člana kojemu se pristupa. Deklarirat ćemo klasu `Racunalo` te objekt i referencu tog tipa:

```
class Racunalo {
public:
    int kBMemorije;
    int brojDiskova;
    int megahertza;
};

Racunalo mojKucniCray;
Racunalo &refRacunalo = mojKucniCray;
```

U gornjoj deklaraciji ključna riječ `public` označava da će članovi klase biti javno dostupni (o tome više riječi u odsječku 8.2.6). Sada bismo memoriju objekta `mojKucniCray` proširili na sljedeći način:

```
mojKucniCray.kBMemorije = 64;
refRacunalo.brojDiskova = 5;    // odnosi se na isti objekt
```

Ako imamo pokazivač na objekt klase `Racunalo`, članovima se pristupa pomoću operatora `->` (minus i veće od), tako da se s lijeve strane operatora navede pokazivač, a s desne strane naziv člana:

```
Racunalo *pokRacunalo = &mojKucniCray;
pokRacunalo->megahertza = 16;
```

Za pristup članovima preko objekata i referenci koristi se operator `.` (točka), a u slučaju pristupa preko pokazivača operator `->` (minus i veće od).

8.2.3. Funkcijski članovi

Dok su podatkovni članovi klase analogni članovima struktura poznatih još iz jezika C, u jeziku C++ elementi klase mogu biti i *funkcijski članovi* (engl. *member functions*), ponegdje još nazvani i *metodama* (engl. *methods*). Oni definiraju skup operacija koje se mogu obaviti na objektu.

Uzmimo za primjer program koji intenzivno koristi vektorski račun. Radi jednostavnosti promatrat ćemo samo slučaj vektora u ravnini. Svaki vektor se može

prikazati jednim objektom klase `Vektor`. Pratit ćemo vektore u Descartesovom koordinatnom sustavu te će svaki vektor biti predstavljen pomoću dva realna broja: `ax` kao projekcija na os *x* i `ay` kao projekcija na os *y*. Želimo li ostvariti množenje vektora skalarom, to bi se u standardnom proceduralnom C-u moglo obaviti na sljedeći način[†]:

```
struct Vektor {
    float ax, ay;
};

void MnoziVektorSkalarom(struct Vektor v, float skalar) {
    v.ax *= skalar;
    v.ay *= skalar;
}

int main() {
    struct Vektor v;
    // ...
    MnoziVektorSkalarom(v, 5);
    return 0;
};
```

Iako je to sasvim ispravno rješenje, ono ima niz nedostataka. Operacija množenja je svojstvena samom vektoru. Naprotiv, gore navedena funkcija za množenje razdvaja operaciju od objekta nad kojim se obavlja – zato smo u funkciji za pristup pojedinim elementima vektora morali koristiti `v.ax` ili `v.ay`. C++ pruža elegantnije rješenje ovog problema:

```
class Vektor {
public:
    float ax, ay;
    void MnoziSkalarom(float skalar);
};

void Vektor::MnoziSkalarom(float skalar) {
    ax *= skalar;
    ay *= skalar;
}
```

Definicija jednog vektora i njegovo množenje u tom sluèaju izgleda ovako:

```
Vektor v;
// ...
v.MnoziSkalarom(5.0);
```

[†] Strukture æe biti obraðene u sljedeæem poglavlju, pa se èitatelj koji nije vièan jeziku C ne mora previše zamarati ovim primjerom.

Pristup pojedinim funkcijama klase se obavlja veæ opisanim operatorima za pristup èlanovima (odjeljak 8.2.2). S lijeve strane operatora se navodi objekt, referenca ili pokazivaè na objekt, dok se s desne strane nalazi naziv funkcijskog èlana. Iza naziva u zagradi je potrebno navesti stvarne argumente.

Prilikom poziva funkcije `MnoziSkalarom()` nije potrebno navoditi kao parametar vektor na koji se množenje odnosi, jer se on već nalazi s lijeve strane operatora za pristup. Članovi `ax` i `ay` koji se pozivaju u funkciji će zapravo biti članovi varijable `v`. U nastavku možemo definirati dodatne vektore kao

```
Vektor normala, a2;
```

Njih množimo skalarima na sljedeæi naèin:

```
normala.MnoziSkalarom(6.7);
a2.MnoziSkalarom(4.0);
```

Naziv funkcijskog èlana mora biti jedinstven u podruèju imena unutar klase.

To znaèi da možemo imati obiènu funkciju `MnoziSkalarom()` izvan klase `Vektor`, koja je potpuno nezavisna od funkcijskog èlana klase `Vektor`. No unutar klase ne smije postojati èlan ili ugniježðena klasa istog imena. Nazivi pridruženih funkcija nisu vidljivi izvan klase, tako da poziv

```
MnoziSkalarom(7.6);
```

sam za sebe bez operatora `.` nema smisla, jer nije jasno èijim `ax` i `ay` se pristupa u toku izvoðenja. Kako ime funkcije nije vidljivo izvan klase, ovakva naredba æe rezultirati pogreškom u prevoðenju "Nepoznata funkcija MnoziSkalarom(float)".

U našem primjeru unutar definicije klase naveli smo samo prototip funkcije `MnoziSkalarom()`, dok smo njenu definiciju naveli izvan klase. Kako imena funkcije nisu vidljiva izvan klase, prilikom definiranja funkcije koristili smo `::` (dvije dvotočke) operator za *razlučivanje područja imena* (engl. *scope resolution operator*). Cijelo ime funkcijskog člana je zapravo `Vektor::MnoziSkalarom()`, pa smo takvo ime morali navesti prilikom definicije funkcije. Valja primijetiti da smo je mogli također pozivati navodeći puno ime

```
v.Vektor::MnoziSkalarom(5.0);
```

Puno ime èlanova deklariranih unutar klase ukljuèuje naziv klase koji se od naziva èlana razdvaja operatorom `::`.

Iako ispravno, ovakvo pisanje je složenije. Objekt `v` je klase `Vektor`, pa prevoditelj sam zakljuèuje da je funkcija `MnoziSkalarom()` iz klase `Vektor`.

8.2.4. Kljuèna rijeè `this`

Možda se pitate kako uistinu funkcionira pozivanje funkcijskih èlanova. Naposljetku, raèunalo na svom najnižem nivou uopæe ne zna ništa o objektima i funkcijama. Ono barata najjednostavnijim pojmovima kao što su adrese, pokazivaèi i brojevi. Objekti su podatkovni tipovi vrlo visokog nivoa apstrakcije te se moraju moæi predstaviti elementarnim raèunalnim tipovima podataka.

Trik je u tome što se prilikom poziva svakog funkcijskog člana klase kao skriveni parametar prosljeđuje pokazivač na objekt kojemu taj član pripada. Tom skrivenom parametru se može pristupiti unutar same funkcije pomoću ključne riječi `this`. Prevoditelj prilikom analiziranja funkcije `MnoziSkalarom()` nju zapravo interpretira kao

```
void Vektor::MnoziSkalarom(float skalar, Vektor *this) {
    this->ax *= skalar;
    this->ay *= skalar;
}
```

a poziv funkcijskog èlana nad objektom `v` interpretira kao

```
Vektor::MnoziSkalarom(5, &v);
```

Velika prednost C++ jezika leži upravo u tome što prevoditelj skriva `this` od korisnika i sam obavlja pristupanje preko pokazivaèa. U veæini primjena korištenje te kljuène rijeèi neæe biti potrebno. Ipak, u nekim specifiènim sluèajevima bit æe potrebno unutar funkcijskog èlana saznati adresu objekta za koji je èlan pozvan. To se tada može uèiniti pomoæu kljuène rijeèi `this`. Važno je napomenuti da se `this` ne smije deklarirati kao formalni parametar funkcijskog èlana. On je automatski dostupan u svakom èlanu te ga nije potrebno posebno navoditi.

Ključna riječ `this` se često koristi kada je potrebno vratiti pokazivač ili referencu na objekt. Da bismo to demonstrirali, proširit ćemo klasu `Vektor` funkcijom `ZbrojiSa()` koja zbraja vektor s nekim drugim vektorom. Promijenit ćemo povratni tip funkcije na `Vektor &`, čime će funkcijski član vraćati referencu na objekt za koji je pozvan.

```
class Vektor {
public:
    float ax, ay;
    Vektor &MnoziSkalarom(float skalar);
    Vektor &ZbrojiSa(float zx, float zy);
};

Vektor &Vektor::MnoziSkalarom(float skalar) {
```

```
        ax *= skalar;
        ay *= skalar;
        return *this;
    }

    Vektor &Vektor::ZbrojiSa(float zx, float zy) {
        ax += zx;
        ay += zy;
        return *this;
    }
```

Ovime je omogućeno da u jednom redu pomnožimo neki vektor sa skalarom te mu dodamo neki drugi vektor, na primjer:

```
Vektor v;
v.ax = 4;
v.ay = 5;
v.MnoziSkalarom(4).ZbrojiSa(5, -7);
```

Operator `.` se izvodi slijeva na desno čime se postiže željeni redoslijed operacija. Ključna riječ `this` se također često koristi prilikom razvoja objekata koji služe kao spremište drugih objekata, kao na primjer dvostruko povezana lista (pojam vezanih lista će biti detaljnije objašnjen u odjeljku 9.2):

```
class ElementListe {
private:
    ElementListe *Sljedeci, *Prethodni;
public:
    void Dodaj(ElementListe *Novi);
    // ...
};

void ElementListe::Dodaj(ElementListe *Novi) {
    Novi->Sljedeci = Sljedeci;
    Novi->Prethodni = this;
    Sljedeci->Prethodni = Novi;
    Sljedeci = Novi;
}
```

8.2.5. Umetnuti funkcijski članovi

Ako je funkcija kratka tako da njena definicija stane u nekoliko redaka, onda je njenu definiciju moguće navesti unutar same deklaracije klase. Time dobivano *umetnutu definiciju* (engl. *inline definition*) funkcijskog člana. Prepišimo našu klasu `Vektor` tako da funkcija `MnoziSkalarom()` bude umetnuta.

```
class Vektor {
public:
    float ax, ay;
    void MnoziSkalarom(float skalar) {
        ax *= skalar;
        ay *= skalar;
    }
};
```

Razlozi radi kojih je poželjno funkcijske èlanove uèiniti umetnutima su isti kao i za

Funkcijski član definiran unutar deklaracije klase je uvijek umetnut, bez eksplicitnog korištenja ključne riječi `inline`.

obiène funkcije, te su detaljno opisani u odsjeèku 5.7, a svode se na brže izvoðenje kôda te (u principu) kraæi prevedeni kôd.
Ako je nezgodno smjestiti umetnutu funkciju u samu definiciju klase, isti efekt se može postiæi kljuènom rijeèi `inline` kao u sljedeæem primjeru.

```
class Vektor {
public:
    float ax, ay;
    inline void MnoziSkalarom(float skalar);
};

void Vektor::MnoziSkalarom(float skalar) {
...
}
```

Rezultat æe prilikom ovakve deklaracije biti isti. Kljuèna rijeè `inline` se mogla ponoviti prije definicije funkcijskog èlana. Umetnuti funkcijski èlanovi su vrlo važni u C++ jeziku, jer se èesto primjenjuju za postavljanje i èitanje podatkovnih èlanova koji nisu javno dostupni (vidi sljedeæi odsjeèak). Prevoditelj koji ne podržava dovoljno dobro umetnute funkcijske èlanove može vrlo lako generirati kôd znaèajno degradiranih performansi.

8.2.6. Dodjela prava pristupa

U prethodnim primjerima pažljiv èitatelj je sigurno primijetio upotrebu kljuène rijeèi `public` koja do sada nije bila objašnjena. Ona služi za definiranje prava pristupa èlanovima klase. Svaki èlan može imati jedan od tri moguæa naèina pristupa: *javni* (engl. *public*), *privatni* (engl. *private*) i *zaštiæeni* (engl. *protected*). Prava pristupa se dodjeljuju tako da se u tijelu klase navede jedna od tri kljuène rijeèi `public`, `private` i `protected` iza kojih se stavlja dvotoèka. Svi elementi, dakle funkcijski i podatkovni èlanovi te deklaracije ugniježðenih klasa koji slijede iza te rijeèi imaju pravo pristupa

definirano tom rijeèi. Zatim se može navesti neka druga od gore navedenih kljuènih rijeèi, iza koje slijede elementi koji imaju drukèije pravo pristupa. Evo primjera:

```
class Pristup {
public:
    int a, b;
    void Funkcija1(int brojac);
private:
    int c;
protected:
    int d;
    int Funkcija2();
};
```

Prava pristupa odreðuju koji æe èlanovi klase biti dostupni izvan klase, koji iz naslijeðenih klasa, a koji samo unutar klase. Time programer toèno odreðuje "publiku" za pojedini dio objekta.

- Javni pristup se dodjeljuje ključnom riječi `public`. Članovima koji imaju javni pristup može se pristupiti izvan klase. Cjelobrojni elementi `a` i `b`, te `Funkcija1()` se mogu pozvati tako da se definira neki objekt klase `Pristup` te im se pomoću operatora `.` pristupi.
- Član `c` ima privatni pristup dodijeljen ključnom riječi `private`. U vanjskom programu te u klasama koje su naslijeđene od klase `Pristup` on nije dostupan – njemu se može pristupiti samo preko funkcijskih članova klase `Pristup`.
- Zaštićeni pristup se određuje ključnom riječi `protected` te se time ograničava mogućnost pristupa članu `d` i funkcijskom članu `Funkcija2()`. Njima se ne može pristupiti iz vanjskog programa preko objekta. Dostupne su samo u funkcijskim članovima klase `Pristup` i klasama koje su naslijeđene od klase `Pristup`. Nasljeđivanjem se bavi posebno poglavlje 9 ove knjige.

Redoslijed navoðenja prava pristupa je proizvoljan, a pojedine vrste pristupa je moguæe ponavljati. Dozvoljeno je primjerice definirati prvo elemente s javnim, zatim elemente s privatnim te ponovo elemente s javnim pristupom. Èlanovi navedeni u deklaraciji klase odmah nakon otvorene vitièaste zagrade do prve kljuène rijeèi za specifikaciju prava pristupa (ili do kraja klase ako prava pristupa uopæe nisu navedena) imaju privatni pristup. Ako se negdje u programu pokuša nedozvoljeno pristupiti privatnom ili zaštiæenom elementu, dobit æe se pogreška prilikom prevoðenja "Za element 'xx' nema prava pristupa".

> Ako se pravo pristupa ne navede eksplicitno, za klase se podrazumijeva privatni pristup.

Da bismo ilustrirali korištenja prava pristupa, stvorit æemo objekt `x` klase `Pristup` te æemo pokazati kojim se podatkovnim èlanovima iz kojeg dijela programa može pristupiti.

```
Pristup x;

void Pristup::Funkcija1(int brojac) {
    // unutar funkcijskog člana objekta može se
    // pristupiti svim članovima ravnopravno
    a = brojac;
    c = a + 5;
    Funkcija2();
}

int main() {
    // ovo je OK: a i b imaju javni pristup
    x.a = x.b = 6;
    // i ovo je OK: član Funkcija1(int) također
    // ima javni pristup
    x.Funkcija1(1);
    // ovo nije OK: c ima privatni pristup te mu se ne
    // može pristupiti izvana
    cout << x.c << endl;
    // niti ovo ne valja: Funkcija2() ima zaštićeni
    // pristup
    x.Funkcija2()
    return 0;
}
```

Zadatak. *Zašto sljedeća klasa nije pretjerano korisna:*

```
class Macka {
    int ReciBrojGodina() { return brojGodina; }
private:
    int brojGodina;
};
```

(Odgovor leži u starom pravilu bon-tona, koje kaže da se pripadnice ljepšeg spola ne pita za godine!)

Zadatak. *Deklarirajte klasu `Pravac` s privatnim članovima `k` i `l` koji će sadržavati koeficijent smjera i odsječak pravca na osi y. Dodajte javne funkcijske članove `UpisiK()` i `CitajK()`, te `UpisiL()` i `CitajL()` za pristup podatkovnim članovima.*

8.2.7. Formiranje javnog suèelja korištenjem prava pristupa

Osobi koja se po prvi put susreæe sa C++ jezikom može se uèiniti da su komplikacije oko prava pristupa nepotrebne, te da samo unose dodatnu entropiju u veæ ionako dovoljno kaotièan naèin programiranja, koji više slièi *Brownovom gibanju*. To je i donekle toèno u programima od nekoliko stotina redaka koji se napišu u jednom popodnevu. No prilikom razvoja složenih aplikacija sa stotinama tisuæa linija kôda gdje

cijeli programerski tim radi na projektu, prava pristupa omoguæavaju programerima da odrede što je dostupno suradnicima koji koriste objekte, a što njima samima.

Članovi klase s javnim pristupom formiraju javno sučelje objekta. Do javnog sučelja programer dolazi analizom uloge pojedinog objekta i načina njegovog korištenja. Ono se zatim može obznaniti suradnicima koji točno znaju što objektu trebaju pružiti te što od njega mogu dobiti. Sadržaj objekta za njih predstavlja "crnu kutiju". Implementaciju klase programer piše na osnovu javnog sučelja, čime javno sučelje postaje neovisno o implementaciji. Kasnije se analizom može ustanoviti da neka druga implementacija omogućava bolje performanse programa. Objekt se može

Dobar pristup objektno orijentiranom programiranju nalaže da se javno sučelje objekta odvoji od njegove implementacije.

jednostavno preraditi, dok ostatak kôda ne treba dirati – on vidi samo javno sučelje objekta koje ostaje neizmijenjeno.
Da bismo to pojasnili, vratimo se na primjer s vektorima te preradimo klasu tako da njena implementacija bude neovisna od javnog suèelja:

```
class Vektor {
private:
    float ax, ay;
public:
    void PostaviXY(float x, float y) { ax = x; ay = y; }
    float DajX() { return ax; }
    float DajY() { return ay; }
    void MnoziSkalarom(float skalar);
};
```

Definicija funkcijskog èlana `MnoziSkalarom()` je izostavljena jer se ne mijenja. Implementacija klase pretpostavlja da se vektor pamti u Descartesovim koordinatama. Javno suèelje ništa ne govori o koordinatama, ono samo omoguæava da se svakom vektoru utvrdi njegova projekcija na *x* i na *y* os, što je moguæe neovisno o koordinatnom sustavu u kojem je vektor zapamæen. Sada je moguæe promijeniti koordinatni sustav u polarni, ustanovi li se da takav prikaz omoguæava bolja svojstva prilikom izvođenja. I na temelju takvog naèina pamæenja vektora mogu se implementirati svi elementi javnog suèelja. Funkcijski èlanovi `DajX()`, `DajY()` i `PostaviXY()` su napisani kao umetnuti, kako bi se postiglo brže izvršenje programa. Veæina suvremenih prevoditelja æe generirati isti kôd kao da se direktno pristupa èlanovima `ax` i `ay` tako da se performanse programa ne degradiraju.

Umetnuti funkcijski èlanovi se vrlo èesto koriste za dohvaæanje privatnih podatkovnih èlanova kako bi se implementacija razdvojila od javnog suèelja. Kako se time program ne bi usporio, poželjno je koristiti umetnute funkcijske èlanove.

Zadatak. *Modificirajte klasu* `Vektor` *tako da se vektori pamte u polarnom koordinatnom sustavu. Dodajte nove elemente javnog sučelja za postavljanje i čitanje komponenata vektora u tom sustavu, ali zadržite i sve stare elemente.*

8.2.8. Prijatelji klase

Ponekad može biti nužno da neka funkcija, funkcijski èlan ili pak cijela klasa ima pravo pristupa privatnim i zaštiæenim èlanovima neke druge klase. Pojedina klasa u tom sluèaju može dodijeliti željenoj funkciji, funkcijskom èlanu neke klase ili èitavoj klasi pravo pristupa vlastitim privatnim i zaštiæenim èlanovima. Dio programa kojemu je dodijeljeno pravo pristupa zove se *prijatelj klase* (engl. *a friend to a class*) koja mu je to pravo dala.

Definirajmo za primjer funkciju koja će zbrajati dva vektora. Ona bi to mogla učiniti pomoću javnog sučelja vektora, no znatno je prirodnije pristupati direktno članovima klase `Vektor`. Klasa se tada može redefinirati ovako:

```
class Vektor {
    friend Vektor ZbrojiVektore(Vektor a, Vektor b);
private:
    float ax, ay;
public:
    // ...
};

Vektor ZbrojiVektore(Vektor a, Vektor b) {
    Vektor c;
    c.ax = a.ax + b.ax;
    c.ay = a.ay + b.ay;
    return c;
}
```

Deklaracija neke funkcije kao prijatelja poèinje kljuènom rijeèi `friend` iza koje se navede puni prototip funkcije kojoj se dodjeljuju prava pristupa. Nije potrebno deklarirati funkciju prije nego što se ona uèini prijateljem. Tako je u našem primjeru sama funkcija deklarirana i definirana nakon deklaracije klase. Ako je potrebno više funkcija uèiniti prijateljem, tada se svaka funkcija navodi zasebno. Nije moguæe nakon jedne kljuène rijeèi `friend` navesti više deklaracija. Iako se deklaracije prijatelja mogu pojaviti bilo gdje u tijelu klase, uvriježeno je pravilo da ih se navodi odmah iza deklaracije klase.

Prava pristupa je moguće dodijeliti i pojedinim funkcijskim članovima drugih klasa. Tako je moguće definirati klasu `Matrica` koja će sadržavati funkcijski član koji množi matricu vektorom. Klasa `Vektor` će dodijeliti prava pristupa funkcijskom članu `Matrica::MnoziVektorom()` na sljedeći način:

```
class Vektor;
```

```
class Matrica {
private:
    float a11, a12, a21, a22;
public:
    // ...
    Vektor MnoziVektorom(Vektor &v);
};

class Vektor {
    friend Vektor Matrica::MnoziVektorom(Vektor &v);
    // ...
};

Vektor Matrica::MnoziVektorom(Vektor &v) {
    Vektor rez;
    rez.ax = a11 * v.ax + a12 * v.ay;
    rez.ay = a21 * v.ax + a22 * v.ay;
    return rez;
}
```

Prilikom dodjele prava pristupa funkcijskom èlanu klase postoji pravilo da se pravo ne može dodijeliti ako funkcijski èlan nije veæ prethodno deklariran. Zato je deklaracija klase `Matrica` prethodila deklaraciji klase `Vektor`, èime je postignuto da u trenutku deklariranja klase `Vektor` funkcijski èlan kojem se žele dodijeliti prava veæ bude deklariran. Definicija funkcijskog èlana `Matrica::MnoziVektorom()` je stavljena nakon definicije klase `Vektor` iz dva razloga: struktura klase `Vektor` mora biti poznata da bi se moglo pristupati njegovim èlanovima te klasa `Vektor` mora dodijeliti pravo pristupa funkcijskom èlanu kako bi mogao pristupiti privatnim èlanovima.

Ponekad je potrebno dodijeliti pravo pristupa svim funkcijskim članovima jedne klase. Tada se cijela klase može deklarirati prijateljem druge klase. U tom slučaju ta klasa ima pristup svim privatnim i zaštićenim tipovima, pobrojenjima i ugniježđenim klasama. To se čini tako da se iza ključne riječi `friend` navede ključna riječ `class` za kojom slijedi naziv klase kojoj se prava dodjeljuju. Klasa kojoj se dodjeljuju prava pristupa mora u trenutku dodjele biti deklarirana, makar i nepotpuno (unaprijed), kao u primjeru:

```
class Matrica;

class Vektor {
    friend class Matrica;
    // ...
};
```

Svi privatni i zaštiæeni tipovi definirani unutar klase `Vektor` mogu se sada koristiti i u definiciji funkcijskih èlanova klase `Matrica`.

Unutar deklaracije `friend` se funkcija može i definirati – tada će ona biti umetnuta. Time je postignuta sličnost s funkcijskim članovima koji se također mogu definirati u

deklaraciji te na taj način postaju umetnuti. Na primjer, funkciju `ZbrojiVektore()` možemo definirati ovako:

```
class Vektor {
    friend Vektor ZbrojiVektore(Vektor a, Vektor b) {
        Vektor c;
        c.ax = a.ax + b.ax;
        c.ay = a.ay + b.ay;
        return c;
    }
private:
    float ax, ay;
public:
    // ...
};
```

Funkcija `ZbrojiVektore()` æe biti umetnuta, što osigurava njeno brzo izvoðenje. Iako je ona definirana unutar klase, važno je primijetiti da ona nije èlan klase – ovakav naèin pisanja funkcije samo nam omoguæava da uštedimo jednu kljuènu rijeè `inline`. Funkcija je inaèe posve jednaka kao da je napisana izvan klase.

Funkcijski članovi druge klase se ne mogu definirati unutar `friend` deklaracije. To je i razumljivo, jer se oni moraju definirati unutar klase kojoj pripadaju.

Zadatak. *Deklarirajte hipotetsku klasu `Covjek`, te klasu `Pas`. Pri tome klasu `Pas` postavite prijateljem klase `Covjek`. Je li time klasa `Covjek` prijatelj klase `Pas`?*

Zadatak. *Deklarirajte klasu `Tocka` s privatnim podatkovnim članovima `x` i `y` koji će pamtiti koordinatu točke u pravokutnom koordinatnom sustavu. Napišite funkciju za računanje udaljenosti točke od pravca koja će biti prijatelj i klase `Pravac` (vidi zadatak na stranici 228) i klase `Tocka`, pristupat će izravno njihovim podatkovnim članovima, te će biti ovako deklarirana:*

```
float UdaljenostTockaPravac(Pravac &p, Tocka &t);
```

8.3. Deklaracija objekata klase

Iako je stvaranje objekata klase veæ bilo pokazano na primjerima, u ovom odjeljku æemo tome posvetiti dodatnu pažnju.

Deklaracija klase ne uzrokuje nikakvu dodjelu memorije – memorija se dodjeljuje tek kad se deklarira konkretan objekt. Klasa nije ništa drugo nego tip podataka, pa deklaracija objekata slijedi uobičajenu sintaksu gdje se iza identifikatora tipa navode identifikatori objekata odvojeni zarezom:

```
Vektor strelicnik;
Vektor normala = strelicnik, polje[10];
Vektor *pokStrelicnik = &strelicnik, &refNormala = normala;
```

Ovom naredbom je definirano pet objekata. Prvi nema inicijalizacijski kôd pa elementi vektora `strelicnik` po njegovom stvaranju imaju vrijednosti koje su navedene u *podrazumijevanom konstruktoru klase* (vidi odjeljak 8.4.2). Objekt `normala` ima inicijalizator koji kaže da se objekt inicijalizira tako da bude logièki jednak objektu `strelicnik`. Ako klasa drukèije ne specificira, inicijalizacija se provodi tako da se meðusobno pridruže svi elementi objekta `strelicnik` objektu `normala`. Klasa može definirati poseban postupak za inicijalizaciju pomoæu *konstruktora kopije* (vidi odjeljak 8.4.4). Objekt `polje` je zapravo polje od deset vektora. Mogu se definirati pokazivaèi i reference na objekte. Tako je `pokStrelicnik` pokazivaè na vektor i inicijalizira se tako da pokazuje na objekt `strelicnik`. Referenca `refNormala` se inicijalizira tako da pokazuje na objekt `normala`. Ponovo napominjemo da se referenca uvijek mora inicijalizirati prilikom deklaracije.

Naziv objekta mora biti jedinstven u području imena u kojem je deklaracija navedena. To znači da se za naziv objekta ne može koristiti naziv neke postojeće klase ili nekog drugog objekta.

8.4. Stvaranje i uništavanje objekata

Kao što je veæ reèeno, klasa može sadržavati funkcijske èlanove koji imaju posebne namjene. Oni se pozivaju prilikom stvaranja i uništavanja objekata, kopiranja objekata, prosljeðivanja parametara i slièno.

8.4.1. Konstruktor

Kada se stvori novi objekt neke klase, njegovi èlanovi su neinicijalizirani. Vrijednosti pojedinih èlanova ovise o podacima koji su se sluèajno našli na tom memorijskom prostoru prije nego što je objekt stvoren. Kako je rad s podacima èija je vrijednost odreðena stohastièki potencijalno vrlo opasan, C++ nudi elegantno rješenje. Svaka klasa može imati konstruktorski funkcijski èlan koji se automatski poziva prilikom stvaranja objekata. *Konstruktor* (engl. *constructor*) se deklarira kao funkcijski èlan koji nema specificiran povratni tip te je imena identiènog nazivu klase.

Konstruktor ima isto ime kao i klasa kojoj pripada, te nema povratni tip.

Dodajmo klasi `Vektor` konstruktor koji æe svaki novostvoreni vektor inicijalizirati na nul-vektor:

```
class Vektor {
private:
    float ax, ay;
public:
    // definirana su dva konstruktora:
    Vektor();
```

```
    Vektor(float x, float y) { ax = x; ay = y; }
    void PostaviXY(float x, float y) { ax = x; ay = y; }
    float DajX() { return ax; }
    float DajY() { return ay; }
    void MnoziSkalarom(float skalar);
};

Vektor::Vektor() {
    ax = 0;
    ay = 0;
}
```

Kao što vidimo u gornjem primjeru, konstruktor klase može biti preoptereæen – definirana su dva konstruktora: `Vektor::Vektor()` koji nema parametara te `Vektor::Vektor(float, float)`. Naime, pojedina klasa može biti inicijalizirana na razlièite naèine. Za svaki od njih imat æemo zasebni konstruktor kojih uzima specifiène parametre. Prvi konstruktor inicijalizira vektor na nul-vektor, dok drugi za parametre uzima dva realna broja èime se vektor inicijalizira na željenu vrijednost. Prilikom preoptereæenja konstruktora vrijede ista pravila kao i za preoptereæenje obiènih funkcija. Ukratko, mora biti moguæe jednoznaèno razluèiti koji konstruktor je pozvan na osnovu proslijeðenih mu parametara. Konstruktor može imati i podrazumijevane parametre.

Konstruktor se automatski poziva prilikom stvaranja objekta klase. Pri tome se parametri konstruktora navode nakon naziva objekta u zagradama. Na primjer, vektor inicijaliziran na vrijednost (3.0, –7.0) će se stvoriti na sljedeći način:

```
Vektor mojInicijaliziraniVektor(3.0, -7.0);
```

Nakon alokacije memorijskog prostora za objekt, pozvat æe se konstruktor kojemu æe se proslijediti parametri navedeni u zagradama i konstruktor æe provesti inicijalizaciju.

Ako se želi pozvati konstruktor bez parametara, ne smije se navesti par `()` nakon naziva objekta. Prevoditelj to ne bi shvatio kao deklaraciju objekta, već deklaraciju funkcije bez parametara koja vraća objekt.

U sljedeæem primjeru, `bezParametara` je vektor inicijaliziran konstruktorom bez parametara, dok je `oopsla`, suprotno oèekivanju, deklaracija funkcije bez parametara koja vraæa `Vektor`:

```
Vektor bezParametara;   // deklaracija vektora
Vektor oopsla();        // deklaracija funkcije bez parametara
                        // koja vraća Vektor
```

Primjena konstruktora posebno dolazi do izražaja u sluèaju da klasa sadrži pokazivaèe te koristi dinamièku alokaciju memorije. Uzmimo na primjer da želimo napraviti klasu `Tablica` koja sadrži niz cijelih brojeva. Tablicu je potrebno moæi pretraživati te

provjeravati na kojoj se poziciji određeni broj nalazi, a potrebni su i funkcijski èlanovi za dodavanje i brisanje elemenata. Evo moguæeg rješenja:

```
#include <assert.h>

class Tablica {
private:
    int *Elementi;
    int BrojElem, Duljina;
public:
    Tablica();
    Tablica(int BrElem);
    void PovecajNa(int NovaDulj);
    void DodajElem(int Elt);
    void BrisiElem(int Poz);
};
```

Osnovna zadaæa konstruktora jest inicijalizirati objekt tako da se dodijeli određeni memorijski prostor za tablicu:

```
Tablica::Tablica() :
            Elementi(new int[10]),
            BrojElem(0), Duljina(10) {
    assert(Elementi != NULL);
}

Tablica::Tablica(int BrElem) :
            Elementi(new int[BrElem]),
            BrojElem(0), Duljina(BrElem) {
    assert(Elementi != NULL);
}
```

Preostali funkcijski èlanovi za pristup i održavanje tablice izgledaju ovako:

```
void Tablica::PovecajNa(int NovaDulj) {
    int *Nova=new int[NovaDulj];
    assert(Nova != NULL);
    for (int i = 0; i < BrojElem; i++)
        Nova[i] = Elementi[i];
    Duljina = NovaDulj;
    delete [] Elementi;
    Elementi = Nova;
}

void Tablica::DodajElem(int Elt) {
    if (Duljina == BrojElem)
        PovecajNa(Duljina + 10);
    Elementi[BrojElem++] = Elt;
}
```

```
void Tablica::BrisiElem(int poz) {
    assert(poz < BrojElem);
    for (int i = poz; i < BrojElem - 1; i++)
        Elementi[i] = Elementi[i+1];
    BrojElem--;
}
```

Osim dodjele memorije, zadatak konstruktora je postavljanje podataka o broju elemenata u tablici te podešavanje njene duljine. Prvi konstruktor nema parametara i on inicijalizira tablicu na duljinu od deset elemenata. Drugi konstruktor ima kao parametar poèetnu duljinu tablice te on inicijalizira tablicu na željenu duljinu.

Važno je uočiti činjenicu da konstruktor ne alocira memoriju za smještanje samog objekta (tj. za smještanje podatkovnih članova `Elementi`, `BrojElem`, `Duljina`). Ta memorija se dodjeljuje izvan konstruktora (u gornjem slučaju prevoditelj će sam generirati kôd koji će to obaviti). Konstruktor na već alociranom objektu samo obavlja

Konstruktor ne alocira memoriju za pohranjivanje članova objekta, no može dodatno alocirati memoriju koju objekt koristi.

posao inicijalizacije. Postupak inicijalizacije klase `Tablica` zahtijeva dodjelu posebnog područja memorije za pohranjivanje tablice.
Pažljivi èitatelj je zasigurno primijetio makro funkciju `assert()` koja do sada nije objašnjena. Još pažljiviji èitatelj je vjerojatno i naslutio da ona služi za provjeru ispravnosti dobivenog pokazivaèa. Makro funkcija `assert()` je definiran u standardnoj datoteci zaglavlja `assert.h`, a funkcionira tako da provjerava uvjet koji je zadan kao parametar. Ako je uvjet zadovoljen, program se normalno nastavlja, a ako nije, program se prekida i ispisuje se poruka o pogreški, uvjet koji ju je izazvao, te podaci o mjestu na kojem se pogreška dogodila. Takav naèin pisanja programa je vrlo praktièan prilikom razvijanja programa jer se tada mogu provjeriti svi važni uvjeti koji æe garantirati ispravno funkcioniranje programa. Takvi testovi, međutim, nepotrebno usporavaju program, pa kada je program jednom istestiran, poželjno ih je iskljuèiti iz izvedbenog kôda. Tada naredbom

```
#define NDEBUG
```

možemo definirati simbol `NDEBUG` (što je skraæenica od *No Debug*). Program je potrebno ponovo prevesti, a svi pozivi `assert()` makro funkcije bit æe zamijenjeni praznom naredbom. Tako se jednim potezom mogu iz programa ukloniti sve nepotrebne provjere. O tome æe još biti dodatno rijeèi u poglavlju 14 koje se bavi pretprocesorom.

Kao što iz kôda klase `Tablica` vidimo, definicija konstruktora se razlikuje od definicije obične funkcije. Naime, prije nego što se uđe u tijelo konstruktora, postoji mogućnost inicijalizacije pojedinih podatkovnih članova, tako da se iza imena konstruktora stavi dvotočka te se navede naziv podatkovnog člana i u zagradi njegova

početna vrijednost. Ako je potrebno inicijalizirati više članova, oni se razdvajaju zarezima. Ta inicijalizacijska lista završava vitičastom zagradom koja označava početak tijela konstruktora. U gornjem primjeru smo na taj način inicijalizirali podatkovne članove `Elementi` i `BrojElem`:

```
Tablica::Tablica(int BrElem) :
            Elementi(new int[BrElem]),
            BrojElem(0), Duljina(BrElem) {
// ...
```

Ovakav naèin inicijalizacije nije obavezan, ali ima prednosti prilikom izvođenja programa. Postavljanje vrijednosti se može obaviti i unutar tijela konstruktora. No ako se izostavi inicijalizacija nekog èlana, C++ prevoditelj æe sam umetnuti kôd za inicijalizaciju. U tom sluèaju se inicijalizacija obavlja dva puta: jednom zato što nije

Inicijalizacija članova prije ulaska u tijelo konstruktora je dobra programerska praksa.

eksplicitno navedena poèetna vrijednost èlana i drugi put u samom konstruktoru. Jasno je da je èitav kôd zbog toga sporiji.
No postoji jedna zamka koja neupuæenom programeru može zadati glavobolju, a to je redoslijed inicijalizacije. Naime, pravila C++ jezika kažu da se podatkovni èlanovi ne inicijaliziraju redoslijedom kojim su navedeni u inicijalizacijskoj listi, nego redoslijedom kojim su navedeni u samoj deklaraciji. Tako konstruktor

```
Tablica::Tablica() : BrojElem(10),
                Elementi(new int[BrojElem]),
                Duljina(BrElem) { /* ... */ }
```

unatoè svim oèekivanjima radi pogrešno. Kako je podatkovni èlan `BrojElem` u deklaraciji klase stavljen iza deklaracije pokazivaèa `Elementi`, on æe biti inicijaliziran na vrijednost 10 tek nakon izvođenja operatora `new` i dodjele memorije. Prilikom inicijalizacije èlana `Elementi` èlan `BrojElem` æe sadržavati nedefiniranu vrijednost i zbog toga æe dodjela memorije biti pogrešno obavljena.

Objekti se ne inicijaliziraju prema redoslijedu navođenja u inicijalizacijskoj listi konstruktora, već prema slijedu deklaracije u klasi.

Ranije navedeni redoslijed pozivanja konstruktora vrijedi i u sluèaju da klasa sadrži objekte druge klase. Njihovi konstruktori se pozivaju prije nego što se uđe u tijelo konstruktora klase koja sadrži objekte. Uzmimo za primjer klasu `Par` koja sadrži dva vektora:

```
class Par {
    Vektor prvi, drugi;
public:
    Par(float x1, float y1, float x2, float y2) :
            drugi(x2, y2), prvi(x1, y1) {}
    // ...
};
```

Konstruktor klase `Par::Par()` poziva konstruktore `prvi` i `drugi` objekata èlanova klase. Buduæi da se èlanovi inicijaliziraju prema redoslijedu deklaracije, prvo æe se inicijalizirati èlan `prvi`, a tek onda `drugi`. U sluèaju da je konstruktor klase `Par` bio napisan kao

```
Par(float x1, float y1) : prvi(x1, y1) {}
```

odnosno da je izostavljen konstruktor za èlan `drugi`, prevoditelj bi automatski pozvao konstruktor bez parametara za `drugi` – on bi bio inicijaliziran kao nul-vektor. Ako konstruktor bez parametara ne bi postojao, prijavila bi se pogreška prilikom prevođenja "Ne mogu pronaæi Vektor::Vektor()".

Svi objekti-članovi neke klase će biti inicijalizirani konstruktorom navedenim u inicijalizacijskoj listi konstruktora dotične klase. Ako je neki objekt izostavljen iz inicijalizacijske liste, pozvat će se podrazumijevani konstruktor.

Podatkovni èlanovi ugrađenih tipova (`int`, `float`, `char`) ne moraju biti navedeni u inicijalizacijskoj listi, te æe u tom sluèaju ostati neinicijalizirani.

Konstruktor se poziva i kod stvaranja dinamičkih objekata. Memorija za dinamički objekt se dodjeljuje operatorom `new` kojemu se kao argument navede tip koji se želi alocirati. Kao rezultat se vraća pokazivač na tip koji je alociran:

```
Vektor *usmjernik = new Vektor;
```

Ova naredba deklarira pokazivaè `usmjernik` na objekt klase `Vektor` te poziva operator `new` koji stvara novi dinamièki vektor u memoriji. Vrijednost pokazivaèa `usmjernik` se postavlja na adresu novostvorenog objekta. Kako nisu specificirani nikakvi parametri, poziva se konstruktor bez parametara. Ako želimo novopeèeni objekt inicijalizirati konstruktorom s parametrima, argumente konstruktora æemo navesti u zagradama iza naziva klase:

```
Vektor *usmjernik = new Vektor(12.0, 3.0);
```

U ovom sluèaju se poziva konstruktor `Vektor::Vektor(float, float)` te se `ax` i `ay` inicijaliziraju na 12.0 i 3.0. Kao i kod stvaranja automatskih objekata klase, kod korištenja operatora `new` potrebno je navesti parametre pomoæu kojih se može

jednoznaèno odrediti koji se konstruktor poziva. Operator `new` poziva konstruktor nakon što alocira potrebnu kolièinu memorije za smještaj objekta. Ako alokacija memorije ne uspije, konstruktor se ne izvodi, a `new` operator vraæa nul-pokazivaè.

Zadatak. *Klasu* `Pravac` *iz zadatka na stranici 228 proširite konstruktorom koji će pravac inicijalizirati pomoću dva objekta klase* `Tocka` *(vidi zadatak na stranici 232). Konstruktor mora imati sljedeću deklaraciju:*

```
Pravac::Pravac(Tocka &t1, Tocka &t2);
```

8.4.2. Podrazumijevani konstruktor

Konstruktor bez parametara se naziva *podrazumijevani konstruktor* (engl. *default constructor*). Ako u klasi nije eksplicitno definiran niti jedan konstruktor, prevoditelj æe automatski generirati podrazumijevani konstruktor u kojem æe pokušati inicijalizirati sve podatkovne èlanove njihovim podrazumijevanim konstruktorom. Klasa

```
class Podrazumijevani {
    int broj;
    Vektor v;
};
```

nema konstruktora. Zato prevoditelj generira podrazumijevani konstruktor koji æe ostaviti èlan `broj` neinicijaliziran, dok æe za èlan `v` pokušati pronaæi podrazumijevani konstruktor u klasi `Vektor`. Kako klasa `Vektor` ima definiran podrazumijevani konstruktor, prevoditelj æe moæi generirati podrazumijevani konstruktor za klasu `Podrazumijevani`. Ako za klasu `Vektor` podrazumijevani konstruktor ne bi bio definiran, prevoditelj bi prijavio pogrešku "Ne mogu pronaæi Vektor::Vektor()".

Automatska generacija podrazumijevanog konstruktora je onemogućena ako je za klasu definiran barem jedan konstruktor. To znači da u klasi

```
class ImaKonstruktor {
    int i;
public:
    ImaKonstruktor(int ii) : i(ii) {}
};
```

zbog toga što je definiran jedan konstruktor, prevoditelj neæe sam generirati podrazumijevani konstruktor. Kako definirani konstruktor uzima jedan parametar, klasa `ImaKonstruktor` neæe imati podrazumijevani konstruktor, te æe prilikom definiranja objekata klase uvijek biti neophodno navesti parametar.

Dobro je uočiti da prevoditelj, za klase koje imaju kao članove reference i konstantne objekte, ne može sam generirati podrazumijevani konstruktor. Razlog je u tome što se reference i konstantni članovi moraju inicijalizirati u inicijalizacijskoj listi konstruktora te im se vrijednost kasnije ne može mijenjati. Prevoditelj ne zna kako treba

inicijalizirati referencu, odnosno konstantan član, pa zbog toga ne stvara podrazumijevani konstruktor. Ako je podrazumijevani konstruktor potreban, korisnik ga mora definirati sam. Nažalost, prevoditelj neće javiti korisniku nikakvu pogrešku ili upozorenje kojim bi mu na to skrenuo pažnju.

Prevoditelj će generirati podrazumijevani konstruktor ako klasa nema definiran niti jedan drugi konstruktor, te ako ne sadrži konstantne članove ili reference.

Prilikom korištenja podrazumijevanih parametara treba biti oprezan. U primjeru

```
class Parametri {
public:
    Parametri(int a = 0, float b = 0.0, char c = ' ');
    // ...
};
```

konstruktor klase je ujedno i podrazumijevani konstruktor. Kako sva tri parametra imaju podrazumijevanu vrijednost te ih se može izostaviti, prevoditelj ne može ustanoviti razliku između tog konstruktora i konstruktora bez parametara. Pokušaj posebne definicije podrazumijevanog konstruktora završio bi pogreškom prilikom prevođenja jer te dvije funkcije nije moguæe razlikovati.

8.4.3. Poziv konstruktora prilikom definiranja objekata

Prilikom definiranja objekata moguæe je navesti parametre koji se prosljeđuju konstruktoru klase. Ti se parametri navode nakon naziva identifikatora objekta u zagradama, sliène parametrima funkcije, kao u sljedeæem primjeru:

```
Vektor normala1(12.0, 3.0)
Vektor normala2;
```

Nakon dodjele memorijskog prostora za objekt `normala1`, pozvat æe se konstruktor `Vektor::Vektor(float, float)` koji æe inicijalizirati podatkovne èlanove `ax` i `ay` na 12.0 i 3.0. Kako kod deklaracije objekta `normala2` nisu definirani parametri, bit æe pozvan podrazumijevani konstruktor pa æe objekt biti inicijaliziran kao nul-vektor.

Prevoditelj prilikom deklaracije objekta analizira parametre i pokušava pronaći odgovarajući konstruktor. Pri tome se primjenjuju pravila o preopterećenju funkcija i o konverziji parametara prilikom poziva funkcije. Ako je konstruktor pronađen, objekt se inicijalizira pomoću njega. U slučaju da nije pronađen, prevoditelj će javiti pogrešku prilikom prevođenja. Deklaracija objekta `normala2` ne bi mogla biti obavljena ako klasa `Vektor` ne bi sadržavala podrazumijevani konstruktor. Pri tome valja imati na umu da klasu koja nema niti jedan konstruktor, prevoditelj sam nadopunjuje podrazumijevanim konstruktorom.

Konstruktori za pojedine tipove objekata pozivaju se na raznim mjestima u programu, ovisno o vrsti objekta:

- za lokalne (automatske) objekte konstruktor se poziva prilikom deklaracije objekta,
- za statičke i globalne objekte prije ulaska u funkciju `main()`,
- za dinamičke objekte prilikom poziva operatora `new`.

8.4.4. Konstruktor kopije

Ponekad je potrebno stvoriti objekt koji je preslika veæ postojeæeg objekta. Takvu inicijalizaciju obavlja posebno definiran *konstruktor kopije* (engl. *copy constructor*). On se razlikuje od ostalih konstruktora po tome što ima samo jedan parametar i to referencu na konstantan objekt iste klase. Evo primjera za konstruktor kopije klase `Vektor`:

```
Vektor::Vektor(const Vektor &refVektor) :
         ax(refVektor.ax), ay(refVektor.ay) {}
```

Uloga konstruktora kopije je preslikavanje èlanova vektora `refVektor` u objekt koji upravo nastaje. Sada je moguæe napraviti sljedeæe deklaracije:

```
Vektor vektor1(12.0, 3.0);
Vektor vektor2 = vektor1;
Vektor vektor3(vektor2);
```

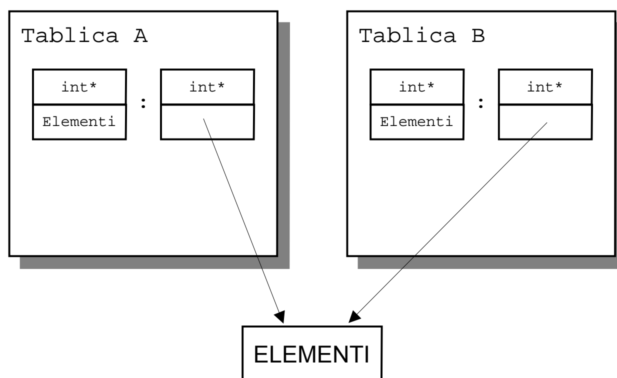
Objekti `vektor2` i `vektor3` se inicijaliziraju preko podataka objekta `vektor1` i na kraju oba sadrže 12 i 3 u svojim `ax` i `ay` èlanovima.

Konstruktor kopije se također koristi za stvaranje privremenih objekata prilikom prosljeđivanja objekata u funkciju i njihovog vraćanja, no to će biti zasebno objašnjeno.

Klasa ne mora definirati konstruktor kopije. Ako on nije eksplicitno definiran, prevoditelj će ga generirati sam, neovisno o tome postoji li još koji drugi konstruktor. Taj konstruktor kopije će pozvati konstruktor kopije za svaki pojedini podatkovni član. Za ugrađene tipove konstruktor kopije jednostavno dodjeljuje njihovu vrijednost odgovarajućem podatkovnom članu objekta. Za podatkovne članove koji su objekti korisnički definiranih klasa, koristi se konstruktor kopije definiran za tu klasu. Iz ovoga je jasno da bi podrazumijevani konstruktor kopije u slučaju klase `Vektor` bio dovoljan jer bi učinio isto što radi i naš eksplicitno definirani konstruktor.

No u slučaju klase `Tablica` stvari su drukčije. Ta klasa sadrži pokazivač na memoriju kojemu se vrijednost određuje prilikom stvaranja objekta. Ako bi konstruktor kopije jednostavno inicijalizirao sve članove kopije vrijednostima članova originala, oba bi objekta pokazivala na isti objekt. To se može ilustrirati slikom 8.3.

Imamo objekt `A` i od njega pomoću konstruktora kopije napravimo objekt `B`. Kako nismo definirali konstruktor kopije za klasu `Tablica`, prevoditelj sam generira podrazumijevani konstruktor kopije. On inicijalizira kopiju tako da prekopira vrijednost svakog člana pa to čini i s pokazivačem `Elementi`. Na kraju imamo dvije tablice koje



Slika 8.3. Objekti `A` i `B` pokazuju na isto memorijsko područje

pokazuju na isti komad memorije. Zbog toga se svaka promjena u tablici `A` odražava i na tablicu `B`. Najopasnija popratna pojava ovakvog kopiranja objekata je u tome što objekt `B` može biti uništen (na primjer završi li funkcija u kojoj je on bio alociran kao automatski objekt) čime će se osloboditi memorija na koju pokazuje pokazivač `Elementi`. No pokazivač u objektu `A` će i dalje pokazivati na isto mjesto koje je sada nevažeće. Pokušaj izmjene tablice `A` će promijeniti sadržaj memorije na koju `A` pokazuje, no ta memorija je u međuvremenu možda dodijeljena nekom drugom objektu. Vidimo da ovakav pristup vodi u sveopću zbrku, a programi pisani na ovakav način ne mogu funkcionirati ispravno. Ovakvo kopija se često naziva *plitka kopija* (engl. *shallow copy*), a situacija u kojoj pokazivači dvaju objekata pokazuju na isto memorijsko područje na engleskom se naziva *pointer aliasing*[†].
Zato æemo dodati konstruktor kopije klasi `Tablica` koji æe alocirati zasebno memorijsko podruèje za svaki novi objekt te prepisati sadržaj tablice. Time æemo dobiti tzv. *duboku kopiju* (engl. *deep copy*) objekta:

```
class Tablica {
// ...
public:
    Tablica(const Tablica &refTabl);
};

Tablica::Tablica(const Tablica &refTabl) :
            Elementi(new int[refTabl.Duljina]),
            Duljina(refTabl.Duljina),
            BrojElem(refTabl.BrojElem) {
    assert(Elementi != 0);
    for (int i = 0; i < BrojElem; i++)
```

[†] Uz najbolju volju, ovaj termin nismo nikako mogli suvislo prevesti. Na engleskom *alias* znaèi lažno ime, no termin *predstavljanje pokazivaèa lažnim imenom* æe prije izazvati provale smijeha, no opisati problem.

```
            Elementi[i] = refTabl.Elementi[i];
    }
```

Za konstruktore kopije vrijede ista pravila za inicijalizaciju èlanova kao i za obiène konstruktore.

Klase koje sadrže pokazivače na dinamički alocirane segmente memorije gotovo uvijek iziskuju korisnički definiran konstruktor kopije.

Zadatak. *Deklarirajte klasu* `ZnakovniNiz` *koja će sadržavati privatni pokazivač na znakovni niz. Dodajte konstruktor koji će kao parametar imati pokazivač na znakovni niz te inicijalizirati objekt pomoću proslijeđenog niza. Također, dodajte konstruktor kopije koji će omogućiti kopiranje objekata. Obratite pažnju na zaključni nul-znak.*

8.4.5. Inicijalizacija referenci i konstantnih èlanova

Klase mogu sadržavati konstantne èlanove i reference na objekte. Takvi èlanovi imaju poseban tretman prilikom inicijalizacije i pristupa njihovoj vrijednosti. Promotrimo pobliže o èemu se radi.

Iz prethodnih poglavlja znamo da se referenca mora inicijalizirati prilikom deklaracije. Ne postoji mogućnost ostaviti referencu neinicijaliziranu ili ju naknadno preusmjeriti. U klasama se referenca može deklarirati, ali za različite objekte iste klase ona može referirati različite podatke. Na primjer, možemo definirati klasu `Par` koja će opisivati uređeni par vektora, s time da vektori neće biti dio klase, već će svaki `Par` sadržavati po dvije reference na vektore. Evo deklaracije klase:

```
class Par {
public:
    Vektor &refPrvi, &refDrugi;
    // nastavak slijedi ...
};
```

Ako se koristi operator pridruživanja za pristup referenci, mijenja se vrijednost referiranog objekta, a ne reference:

```
Par p;          // ovakva inicijalizacija nije ispravna
Vektor v(12.0, 3.0);
p.refPrvi = v;  // mijenja se vrijednost referiranog objekta
```

Takoðer, ne možemo referencu inicijalizirati prilikom deklaracije, jer za svaki `Par` referenca ima drugu vrijednost. Jedino rješenje je inicijalizirati referencu u konstruktoru, prilikom stvaranja objekta. Kako bi se naznaèilo da se ne pristupa referiranom objektu, nego vrijednosti reference, ona se mora inicijalizirati u inicijalizacijskoj listi konstruktora. Ako klasa sadrži referencu te ako ju ne inicijaliziramo na taj naèin, dobit

æemo pogrešku prilikom prevođenja, jer reference ne smiju ostati neinicijalizirane. Evo kako to izgleda na primjeru klase `Par`:

```
class Par {
public:
    Vektor &refPrvi, &refDrugi;
    Par(Vektor &v1, Vektor &v2) : refPrvi(v1), refDrugi(v2) {}
};
```

Konstruktor klase `Par` kao parametar ima reference na objekte klase `Vektor` te se reference `refPrvi` i `refDrugi` inicijaliziraju tako da pokazuju na parametre. Obavezno je bilo staviti `v1` i `v2` kao reference: u suprotnom bi se prilikom poziva konstruktora stvarale privremene kopije stvarnih parametara, a reference bi se inicijalizirale adresom tih privremenih kopija. Nakon završetka konstruktora kopije bi se uništile, a reference bi pokazivale na memoriju u kojoj više nema objekta. U nastavku je dan primjer alociranja objekta klase `Par` i njegove ispravne inicijalizacije:

```
Vektor a(10.0, 2.0), b(-2.0, -5.0);
Par p(a, b);            // ispravna inicijalizacija
```

Reference se obavezno moraju inicijalizirati, te se to mora učiniti u inicijalizacijskoj listi konstruktora.

Na slième probleme nailazimo ako klasa sadržava konstantne èlanove: njih je također potrebno inicijalizirati u konstruktoru, a vrijednost im se kasnije ne može mijenjati. Na primjer, neka želimo klasu `Tablica` promijeniti tako da ona sadrži èlan `maxDuljina` koji oznaèava maksimalan broj èlanova u tablici. Taj èlan je poželjno uèiniti konstantnim, tako da se njegova vrijednost ne može kasnije mijenjati. Evo deklaracije klase:

```
class Tablica {
private:
    int *Elementi;
    int BrojElem, Duljina;
    const int maxDuljina;
public:
    Tablica();
    Tablica(int BrElem, int duljina);
    // ostali članovi se ne mijenjaju
};
```

Sada se èlan `maxDuljina` obavezno mora inicijalizirati, i to se mora uèiniti u inicijalizacijskoj listi konstruktora:

```
Tablica::Tablica() : maxDuljina(50),
                     Elementi(new int[10]),
                     BrojElem(0), Duljina(10) {

}

Tablica::Tablica(int BrElem, int duljina) :
            maxDuljina(duljina), Elementi(new int[10]),
            BrojElem(0), Duljina(BrElem) {
}
```

Prvi konstruktor inicijalizira èlan `maxDuljina` na vrijednost 50, dok ga drugi postavlja na proslijeðenu vrijednost.

Konstantni podatkovni članovi objekta se obavezno moraju inicijalizirati, te se to mora učiniti u inicijalizacijskoj listi konstruktora.

8.4.6. Konstruktori i prava pristupa

Konstruktori takoðer podliježu pravilima prava pristupa. Ovisno o mjestu u deklaraciji klase oni imaju javni, privatni ili zaštiæeni pristup. Prilikom deklaracije objekta prevoditelj odreðuje koji se konstruktor poziva. Ako je izabrani konstruktor nedostupan zbog, primjerice privatnog prava pristupa, objekt ne može biti deklariran i prevoditelj javlja pogrešku prilikom prevoðenja. Prepravimo klasu `Vektor` tako da podrazumijevani konstruktor bude privatni:

```
class Vektor {
private:
    float ax, ay;
    Vektor();
public:
    Vektor(float x, float y);
    void NekaFunkcija();
    // ... ostatak klase
};

void Vektor::NekaFunkcija() {
    Vektor priv; // OK
    // ...
}

int main() {
    Vektor vektor1, vektor2(12, 3);
    // vektor1 javlja pogrešku
    // ...
    return 0;
}
```

Deklaracija vektora `vektor1` u funkciji `main()` javlja pogrešku prilikom prevoðenja jer podrazumijevani konstruktor ima privatna prava pristupa. Naprotiv, vektor `vektor2` se može normalno deklarirati jer pozvani konstruktor ima javni pristup. Vektor `priv` u funkcijskom èlanu `NekaFunkcija()` se može deklarirati unatoè tome što je podrazumijevani konstruktor privatan – njemu se može pristupiti unutar klase.

Ako je potrebno onemogućiti stvaranje objekata neke klase (osim u “posvećenim” dijelovima programa, kao što su prijateljske funkcije i slično), to se može postići tako da se svi konstruktori učine privatnima ili zaštićenima. Slično vrijedi i za konstruktor kopije: ako želite onemogućiti kopiranje objekta, konstruktor kopije učinite privatnim.

8.4.7. Destruktor

Kada objekt više nije potreban, on se uklanja iz memorije. Pri tome se poziva *destruktor* (engl. *destructor*) koji je zadužen za oslobaðanje svih resursa dodijeljenih objektu. To je takoðer funkcija koja ima naziv identièan nazivu klase ispred kojeg stoji znak ~ (tilda). Tako destruktor za klasu `Tablica` ima naziv `~Tablica`. Destruktor, kao i konstruktor, nema povratni tip. Destruktor se automatski poziva u sljedeæim situacijama:

- za automatske objekte na kraju bloka u kojem je objekt definiran (kraj bloka je označen zatvorenom vitičastom zagradom),
- za statičke i globalne objekte nakon izlaska iz funkcije `main()`,
- za dinamičke objekte prilikom uništenja dinamičkog objekta operatorom `delete`.

Destruktor ne može biti deklariran s argumentima pa tako ne može niti biti preoptereæen. Takoðer nema specificiran tip koji se vraæa iz funkcije. Kao što konstruktor ne alocira memoriju za objekt nego se poziva nakon što je objekt alociran te inicijalizira objekt, tako destruktor ne dealocira memoriju nego se poziva prije nego što se memorija zauzeta objektom oslobodi te obavlja deinicijalizaciju objekta.

Destruktor ima naziv jednak nazivu klase ispred kojeg stoji znak ~ (tilda) te nema povratnog tipa. Za svaku klasu može postojati samo jedan destruktor.

Klasa `Vektor` ne zauzima nikakve dodatne resurse te zato nema potrebe za eksplicitnom definicijom destruktora. No klasa `Tablica` ima pokazivaè na dinamièku memoriju `Element`, stoga je prilikom uništavanja objekta potrebno osloboditi memoriju dodijeljenu za pohranjivanje èlanova tablice. Zato klasu moramo proširiti na sljedeæi naèin:

```
class Tablica {
    // ...
public:
    ~Tablica();
    // ...
};
```

```
Tablica::~Tablica() {
    delete [] Elementi;
}
```

U destruktoru se pomoæu operatora `delete` oslobaða memorija koja je bila zauzeta u konstruktoru. Korištenje klase `Tablica` bez definiranog destruktora prouzroèilo bi neispravan rad programa jer bi se memorija stalno trošila stvaranjem objekata klase `Tablica`, a nikada se ne bi oslobodila. Prije ili kasnije, takav program bi potrošio svu memoriju raèunala i prestao s radom.

Dinamički objekti se ne uništavaju automatski nakon izlaska iz bloka u kojem su stvoreni već o njihovom uništavanju mora voditi računa programer. Za uništavanje se koristiti operator `delete`. On prije dealokacije memorije poziva destruktor. Operator se koristi tako da se iza ključne riječi `delete` navede pokazivač na objekt koji se želi obrisati, na primjer:

```
// alokacija dinamičkog objekta
Vektor *pokNaStrelicnik = new Vektor(12.0, 3.0);
// dealokacija objekta
delete pokNaStrelicnik;
```

Obratite pažnju na to da dinamièke objekte klase treba eksplicitno brisati operatorom `delete` iako klasa `Vektor` ne mora imati (niti nema) destruktor. U suprotnom bi memorija alocirana prilikom stvaranja objekta ostala vjeèno zauzeta. Mnogi operativni sistemi imaju mehanizme pomoæu kojih nakon završetka programa oslobaðaju svu memoriju koju je on zauzeo, no na njih se ne valja oslanjati.

Pravilo dobrog programiranja je osloboditi svu zauzetu memoriju prije završetka programa.

Operator `delete` se može bez opasnosti primijeniti na pokazivaè koji ima vrijednost nula. U tom sluèaju se destruktor ne poziva.

Nema ograničenja na sadržaj destruktora. On može obaviti bilo kakav zadatak potreban da se objekt u potpunosti ukloni iz memorije. Najčešće se u destruktoru oslobađaju svi objekti dinamički alocirani u konstruktoru. Često je prilikom traženja pogrešaka u programu korisno u destruktor staviti naredbu koja će ispisati poruku o pozivanju destruktora, na primjer:

```
Tablica::~Tablica() {
    delete [] Elementi;
#ifndef NDEBUG
    cout << "~Tablica()" << endl;
#endif
}
```

U ovom primjeru ispis poruke može se iskljuèiti tako da se definira pretprocesorski simbol `NDEBUG`. Naziv tog simbola nije sluèajno izabran. Naime, definiranje tog simbola æe iskljuèiti sve testove makro funkcijom `assert()`. Ako i za naše provjere iskoristimo isti simbol, jednim potezom æemo "ubiti" sve provjere ispravnosti. U gornjem primjeru se koristi uvjetno prevođenje. Prevoditelj æe prevesti naredbe navedene između pretprocesorskih direktiva `#ifndef` i `#endif` ako je simbol `NDEBUG` nije definiran. U suprotnom naredbe neæe biti umetnute u program. Detaljnije o pretprocesorskim naredbama bit æe govora u zasebnom poglavlju.

Postoji situacija u kojoj je eksplicitni poziv destruktora potreban, a to je kada se objekt alocira na točno određenom mjestu u memoriji pomoću operatora `new`. U tom slučaju se zauzeta memorija ne oslobađa, nego se samo uništava objekt u njoj. Primjer za to će biti dan u sljedećem odjeljku.

Zadatak. *Klasi `ZnakovniNiz` iz zadatka na stranici 243 dodajte destruktor tako da se resursi zauzeti u konstruktoru ispravno oslobode.*

Zadatak. *Osim dealokacije memorije, destruktor može obavljati i druge operacije, na primjer za brojenje objekata. Deklarirajte klasu `Muha` koja će imati konstruktor i destruktor. Također, deklarirajte globalnu cjelobrojnu varijablu koja će u svakom trenutku sadržavati broj muha u kompjutoru. (Vjerujte nam na riječ, taj broj uvijek teži u beskonačnost!) Prilikom stvaranja objekta, sadržaj varijable se uvećava, a prilikom uništavanja smanjuje.*

8.4.8. Globalni i statièki objekti

Kao što je moguæe stvoriti globalne i statièke cjelobrojne ili realne varijable, moguæe je stvoriti globalne i statièke objekte korisnièki definiranih klasa. Na primjer:

```
class Kompleksni {
private:
    float r, i;
public:
    Kompleksni(float rr, float ii) : r(rr), i(ii) {}
    float DajRe() { return r; }
    float DajIm() { return i; }
    void Postavi(float rr, float ii) { r = rr; i = ii; }
};

Kompleksni a(10, 20);

Kompleksni sumiraj(Kompleksni &ref) {
    static Kompleksni suma(0, 0);
    suma.Postavi(suma.DajRe() + ref.DajRe(),
                 suma.DajIm() + ref.DajIm());
    return suma;
}

int main() {
```

```
        Kompleksni zbroj = sumiraj(a);
        // ...
    }
```

U gornjem programu funkcija `sumiraj()` zbraja sve argumente koji su joj proslijeđeni prilikom poziva. Suma se pamti u statièkom objektu `suma`, koji je deklariran unutar funkcije kako bi se onemoguæilo vanjskom programu da mijenja sumu direktno. Također, potrebno je objekt deklarirati statièkim kako bi se suma pamtila prilikom svakog poziva funkcije. Također, definiran je i globalni objekt `a`.

Objekti `a` i `suma` su definirani klasom, te je prilikom njihovog nastajanja potrebno provesti postupak konstrukcije. Uostalom, vidi se da oba objekta u svojim deklaracijama imaju parametre koji služe za početno postavljanje objekta. To postavljanje se može provesti jedino u konstruktoru. Također, prilikom završetka programa, oba objekta će biti potrebno uništiti – potrebno je pozvati destruktor.

Dakle, za sve statičke i globalne objekte konstruktor će se izvesti prije upotrebe bilo kojeg objekta ili funkcije u datoteci izvornog kôda. U praksi, to znači da će 99,99% prevoditelja konstruktore pozvati prije ulaska u funkciju `main()`. Destruktori će se

Konstruktori za globalne i statičke objekte će se pozvati prije ulaska u funkciju `main()`, a destruktori nakon završetka funkcije `main()` ili nakon poziva funkcije `exit()`.

pozvati nakon što funkcija `main()` završi ili ako se program "naprasno" završi pozivom funkcije `exit()`.
Konstruktori æe se izvesti u redoslijedu deklaracija unutar datoteke izvornog kôda, a destruktori æe se izvesti obrnutim redoslijedom. Standard ne definira redoslijed pozivanja konstruktora u programima koji imaju više datoteka izvornog kôda, pa je dobra navika ne oslanjati se u programima na redoslijed inicijalizacije objekata iz razlièitih datoteka.

8.5. Polja objekata

U programima se vrlo èesto koriste polja istovrsnih tipova podataka. Jezik C++ posjeduje mehanizme za efikasno definiranje i pristup elementima polja. Sintaksa definiranja polja objekata je usklađena s naèinom definiranja polja ugrađenih tipova. Polje od tri vektora deklarira se, analogno polju cijelih brojeva, na sljedeæi naèin:

```
    Vektor polje[3];
```

U ovom primjeru svaki element polja æe biti inicijaliziran podrazumijevanim konstruktorom. Ako takav konstruktor ne postoji, dobit æemo pogrešku prilikom prevođenja "Ne mogu pronaæi funkciju Vektor::Vektor()".

Ako se polje deklarira bez inicijalizacijske liste, objekti polja se inicijaliziraju podrazumijevanim konstruktorom.

No postoji i moguænost specifikacije konstruktora za svaki element polja posebno, kao u primjeru

```
Vektor polje[3] = { Vektor(12., 3.),
                    Vektor(),
                    Vektor(-3., 5.) };
```

U ovom sluèaju polje ima inicijalizacijsku listu u kojoj je specificiran konstruktor s pripadnim parametrima za svaki èlan polja. Prvi i treæi vektor inicijaliziraju se brojevima 12 i 3 odnosno –3 i 5, dok se drugi vektor postavlja na nul-vektor. Ukoliko ne postoji podrazumijevani konstruktor, eksplicitno navoðenje inicijalizacijske liste s konstruktorom za svaki èlan je jedini naèin alociranja polja objekata.

Moguće je također i dinamički alocirati polje objekata. Polje od tri vektora se može alocirati na sljedeći način:

```
Vektor *polje = new Vektor[3];
```

Kako prilikom dinamièke alokacije polja nema moguænosti navoðenja inicijalizacijske liste, podrazumijevani konstruktor mora postojati kako bi prevoditelj ispravno inicijalizirao èlanove polja.

Elementi dinamički alociranih polja se uvijek inicijaliziraju podrazumijevanim konstruktorom, koji mora postojati da bi dinamička alokacija mogla biti obavljena.

Uništavanje dinamièki alociranih polja objekata se obavlja operatorom `delete` s time što se nakon samog operatora stavljaju znakovi `[]` (otvorena i zatvorena uglata zagrada, kao i kod dinamièki alociranih polja ugraðenih tipova – vidi odjeljak 4.2.6), nakon èega se navodi pokazivaè na prvi element polja, kao u sljedeæem primjeru:

```
delete [] polje;
```

Upotreba znakova `[]` je obavezna, jer oni naznaèavaju prevoditelju da pokazivaè pokazuje na polje objekata alociran operatorom `new`, a ne samo na jedan objekt. Prilikom izvoðenja ove naredbe bit æe automatski dodan kôd koji æe ustanoviti koliko je èlanova bilo alocirano. Zatim æe se za svaki element posebno pozvati destruktor, a na kraju æe cjelokupna memorija biti osloboðena. Izostavljanjem uglatih zagrada kao u naredbi

```
delete polje;   // pogrešno: pozvat će se destruktor samo
                // za prvi član
```

bio bi pozvan samo destruktor za prvi element polja, a ne i za ostale, buduæi da prevoditelju nigdje nije naznaèeno da je `polje` polje. Doduše, ispravna kolièina memorije bi bila oslobođena, ali svi objekti, osim prvoga, ne bi bili ispravno uništeni destruktorom. Ukoliko bi primjerice sadržavali pokazivaèe na dodatno alocirane segmente memorije, oni bi zauvijek ostali u memoriji i nikada ne bi bili oslobođeni. Time dobivamo *memorijsku napuklinu* (engl. *memory leak*) gdje se kolièina slobodne memorije postupno smanjuje. Program naoko dobro funkcionira dok mu u jednom trenutku ne ponestane memorije i tada se zablokira. Takvim pogreškama je vrlo teško uæi u trag.

Prilikom dealokacije polja potrebno je koristiti operator `delete []` koji poziva destruktor za sve članove polja. U protivnom dobit ćemo memorijsku napuklinu, te će se količina raspoložive memorije stalno smanjivati.

Ponekad je baš izrièito potrebno alocirati polje vektora tako da je svaki vektor zasebno inicijaliziran. Operator `new` nudi rješenje: moguæe je smjestiti svaki novostvoreni objekt na toèno određenu adresu. To se postiže tako da se nakon kljuène rijeèi `new` prije specifikacije tipa u zagradama navede pokazivaè na `char` koji određuje mjesto na koje se objekt smješta, na primjer:

```
char *pokMjesto;
// inicijaliziraj pokMjesto

Vektor *pokPolje = new (pokMjesto) Vektor(5, 6);
```

Memorija na koju taj pokazivaè pokazuje mora biti alocirana na neki drugi naèin. Sam operator `new` ne obavlja alokaciju memorije nego samo poziva konstruktor za element i vraæa pokazivaè na njega. Da bismo mogli stvarati objekte na određenom mjestu u memoriji, potrebno je ukljuèiti datoteku `new.h`, jer ona sadrži deklaraciju preoptereæene verzije operatora `new` koji omoguæava alokaciju objekta na željenom mjestu u memoriji. Evo kompletnog primjera:

```
#include <new.h>

const int velicina = 5;
Vektor *AlocirajPolje() {
    int polje[velicina][2] = {{12, 3}, {0, 0}, {1, 1},
                              {6, -7}, {-2, -9}};

    // alocirat ćemo memoriju za cijelo polje
    char *pok = new char[sizeof(Vektor) * velicina];

    for (int i = 0; i < velicina; i++)
```

```
        // stvara se objekt klase Vektor
        // na mjestu određenom pokazivačem
        // pok + sizeof(Vektor) * i
        new (pok + sizeof(Vektor) * i)
                Vektor(polje[i][0], polje[i][1]);

    return (Vektor *)pok;
}
```

Alokacija polja se obavlja tako da se najprije zauzme ukupna memorija potrebna za sve vektore, a zatim se u nju umeæu pojedini vektori koji se inicijaliziraju pomoæu konstruktora s dva argumenta. Ovako alocirano polje nije moguæe dealocirati naredbom

```
delete [] pok;
```

Kako memorija nije alocirana naredbom

```
pok = new Vektor[5];
```

prevoditelj neæe moæi pozvati destruktore za èlanove polja. Dealokacija polja se mora obaviti eksplicitnim pozivanjem destruktora za svaki èlan polja:

```
void DealocirajPolje(Vektor *polje, int vel) {
    for (int i = 0; i < vel; i++)
        polje[i].~Vektor();
    delete [] (char *)polje;
}
```

Kao što se iz primjera vidi, za svaki element se eksplicitno poziva destruktor, a memorija se na kraju dealocira tako da se pokazivaè na prvi element pretvori u pokazivaè na `char` te se oslobodi.

Zadatak. *Potrebno je realizirati objekte koji se koriste u programu za računarsko igranje šaha. Deklarirajte klasu* `SahovskoPolje` *koja će opisivati jedno polje te će pamtiti koja se figura nalazi na polju. Figure su definirane pomoću pobrojenja:*

```
enum Figure {bijeliPjesak, bijeliTop, bijeliKonj,
        bijeliLovac, bijelaKraljica, bijeliKralj,
        crniPjesak, crniTop, crniKonj, crniLovac,
        crnaKraljica, crniKralj};
```

Pojedina fugura će se na polje postavljati pomoću funkcijskog člana `Postavi(Figure fig)`*. Oznaka figure na polju će se moći pročitati pomoću člana* `KojaFigura()`*. Zatim deklarirajte polje* `Sahovnica` *od 8 stupaca i 8 redaka objekata* `SahovskoPolje` *(pri tome podesite konstruktor klase* `SahovskoPolje` *tako da je moguće polje deklarirati). Također postavite polje tako da odražava početnu šahovsku poziciju.*

8.6. Konstantni funkcijski èlanovi

Objekti klasa mogu biti deklarirani kao konstante pomoæu kljuène rijeèi `const`. Takvi objekti se postavljaju na svoju inicijalnu vrijednost u konstruktoru te im se kasnije vrijednost ne može promijeniti. No vrijednost objekta se ionako najèešæe ne mijenja direktnim pristupom podatkovnim èlanovima, nego se koriste funkcijski èlanovi koji mogu promijeniti vrijednost nekog podatkovnog èlana. U tom sluèaju postoji opasnost da, iako je objekt deklariran kao konstantan, poziv funkcijskog èlana ipak promijeni objekt. Kako bi se onemoguæilo mijenjanje konstantnih objekata preko funkcijskih èlanova, uveden je koncept konstantnih funkcijskih èlanova kojima se može regulirati pristup konstantnim objektima.

Funkcijski član se deklarira konstantnim tako da se iza zatvorene zagrade koja završava definiciju liste parametara navede ključna riječ `const` kao u primjeru

```
class NekaKlasa {
    void FunkcijskiClan(int, float) const;
};
```

Time se prevoditelju daje na znanje da funkcijski èlan ne mijenja vrijednost objekta. Takav funkcijski èlan se može slobodno pozvati na konstantnom objektu. Konstantni funkcijski èlan je ogranièen na operacije koje ne mijenjaju vrijednost niti jednog podatkovnog èlana objekta – u suprotnom æe se dobiti pogreška prilikom prevođenja. Kada se objekt klase definira konstantnim, na njemu su dozvoljeni pozivi iskljuèivo

> Konstantni funkcijski članovi su ograničeni isključivo na čitanje podatkovnih članova, te se zbog toga mogu pozivati na konstantnim objektima.

konstantnih funkcijskih èlanova. Ukoliko se pokuša pozvati obièni funkcijski èlan, dobit æe se pogreška prilikom prevođenja.

Da bismo objasnili korištenje konstantnih funkcijskih èlanova modificirajmo klasu `Vektor` tako da funkcijske èlanove koji ne mijenjaju vrijednost objekta definiramo konstantnima:

```
class Vektor {
private:
    float ax, ay;
public:
    Vektor() : ax(0), ay(0) {}
    Vektor(float x, float y) : ax(x), ay(y) {}
    void PostaviXY(float x, float y) { ax = x; ay = y; }
    float DajX() const { return ax; }   // konstantni
    float DajY() const { return ay; }   // funkcijski članovi
};
```

Funkcije `DajX()` i `DajY()` samo èitaju vrijednosti vektora pa su definirane konstantnima. Funkcija `PostaviXY()` mijenja vrijednost objekta pa bi pokušaj njenog definiranja konstantnom funkcijom rezultirao pogreškom prilikom prevođenja. Konstruktori i destruktori ne mogu biti konstantni. Sada je moguæe definirati konstantne vektore:

```
int main() {
    const Vektor v(12, 3);
    cout << "X: " << v.DajX() << endl;  // OK
    cout << "Y: " << v.DajY() << endl;  // OK
    v.PostaviXY(4, 5);                  // pogreška
    return 0;
}
```

Konstantnost (engl. *constness*) ili nekonstantnost funkcijskog èlana je dio potpisa funkcije. To znaèi da klasa može sadržavati konstantnu i nekonstantnu verziju funkcijskog èlana s istim parametrima.

U sluèaju konstantnog objekta poziva se konstantni, a u sluèaju obiènog objekta obièni funkcijski èlan. Klasu vektora možemo modificirati tako da u sluèaju poziva èlana `PostaviXY()` na konstantnom objektu ispišemo poruku o pokušaju mijenjanja konstantnog objekta:

```
class Vektor {
    // ...
    void PostaviXY(float x, float y);
    void PostaviXY(float, float) const;
};

void Vektor::PostaviXY(float x, float y) {
    ax = x;
    ay = y;
}

void Vektor::PostaviXY(float, float) const {
    cout << "Promjena konstantnog objekta." << endl;
}
```

No moguæe je zamisliti situaciju u kojoj i konceptualno konstantna funkcija mora promijeniti vrijednost nekog podatkovnog èlana objekta. Na primjer, neka radi mjerenja brzine programa želimo brojati koliko puta smo pozvali funkcijski èlan `DajX()`. Funkcijski èlan konceptualno ostaje konstantan jer ne mijenja vrijednost objekta, no mora poveæati vrijednost brojaèa pomoæu kojeg se prati broj pristupa objektu. Brojaè æe biti podatkovni èlan klase, pa ga zbog toga neæe biti moguæe mijenjati u konstantnim funkcijskim èlanovima. Problem se može riješiti tako da se pomoæu eksplicitne dodjele tipa odbaci konstantnost pojedinog podatkovnog èlana, s tom

ogradom da klasa mora imati konstruktor. Evo kako to izgleda u funkcijskim èlanovima `DajX()` i `DajY()`:

```
class Vektor {
private:
    float ax, ay;
    int broji;

public:
    Vektor() : ax(0), ay(0), broji(0) {}
    Vektor(float x, float y) : ax(x), ay(y), broji (0) {}
    void PostaviXY(float x, float y) { ax = x; ay = y; }
    float DajX() const;
    float DajY() const;
};

float Vektor::DajX() const {
    ((int&)broji)++;// dodjelom se ukida konstantnost
    return ax;
}

float Vektor::DajY() const {
    ((int&)broji)++;    // dodjelom se ukida konstantnost
    return ay;
}
```

Nedavno je u standard C++ jezika ubaèena kljuèna rijeè `mutable` koja omoguæava još elegantnije rješenje gornjeg problema. Naime, pojedine nekonstantne nestatièke podatkovne èlanove možemo deklarirati tako da ispred deklaracije èlana umetnemo tu kljuènu rijeè. Ti èlanovi æe tada i u konstantnim objektima biti nekonstantni, te æe ih se moæi mijenjati u konstantnim funkcijskim èlanovima. Gornji primjer brojanja pristupa se sada može ovako riješiti:

```
class Vektor {
private:
    float ax, ay;
    mutable int broji;  // član koji neće biti konstantan
                        // niti u konstantnom objektu
public:
    Vektor() : ax(0), ay(0), broji(0) {}
    Vektor(float x, float y) : ax(x), ay(y), broji (0) {}
    void PostaviXY(float x, float y) { ax = x; ay = y; }
    float DajX() const;
    float DajY() const;
};

float Vektor::DajX() const {
    broji++;// dozvoljeno jer je broji deklariran kao
                // mutable
    return ax;
```

```
}

float Vektor::DajY() const {
    broji++;     // dozvoljeno jer je broji deklariran kao
                 // mutable
    return ay;
}
```

Ako se objekt deklarira konstantnim, tada se iz vanjskog programa ne može promijeniti sadržaj javnih podatkovnih èlanova. No èlanovi deklarirani pomoæu `mutable` se mogu mijenjati:

```
class NekaKlasa {
public:
    int neDiraj;
    mutable int radiStoHoces;
};

// ...
const NekaKlasa obj;         // objekt obj je konstantan
obj.neDiraj = 100;           // pogreška: obj je konstantan
obj.radiStoHoces = 101;      // OK: član je mutable
```

Razmotrimo ukratko koliko je korisno korištenje konstantnih objekata. Mnogi programeri smatraju konstantne objekte samo dodatnom gnjavažom te ih ne koriste. No intenzivnim korištenjem konstantnih objekata dobiva se dodatna sigurnosna provjera prilikom prevođenja.

Ako klase koje razvijate koristite isključivo za vlastite potrebe, tada možete ali ne morate koristiti konstantne funkcijske članove kao i konstantne parametre. No ako želite klase koje pišete staviti na raspolaganje i drugim programerima, tada je dobra praksa da se članovi koji ne mijenjaju objekt učine konstantnima. U suprotnom, programer koji inače koristi konstantne objekte je prisiljen ne koristiti ih, jer na konstantnim objektima ne može koristiti niti jedan funkcijski član. Korištenje konstantnih članova je dobra programerska navika.

Korištenje konstantnih funkcijskih èlanova je poželjno ako razvijate klase koje æe koristiti drugi programeri.

Zadatak. *Promijenite klasu* `SahovskoPolje` *tako da članove koji to dozvoljavaju deklarirate konstantnima. Također, deklarirajte polje* `Zavrsnica` *konstantnih objekata klase* `SahovskoPolje` *koje će odražavati raspored figura u nekoj završnici po vašem izboru. Uvjerite se da je za takve objekte moguće pozvati konstantne funkcijske članove tako da ispišete sadržaj šahovske ploče na ekran.*

8.7. Funkcijski èlanovi deklarirani kao `volatile`

Objekti klase mogu biti deklarirani pomoæu kljuène rijeèi `volatile`. Time se prevoditelju daje na znanje da se vrijednost objekta može promijeniti njemu nepoznatim metodama te zbog toga treba izbjegavati optimizaciju pristupa objektu. Takvim objektima se može pristupati iskljuèivo pomoæu funkcijskih èlanova deklariranih pomoæu kljuène rijeèi `volatile` (vidi odsjeèak 2.4.6).

Na objektima koji su definirani ključnom riječi `volatile` mogu se pozvati samo konstruktori, destruktor te funkcijski članovi koji su deklarirani kao `volatile`. Ključna riječ `volatile` se u deklaraciji funkcijskog člana stavlja iza zagrade koja završava listu parametara poput ključne riječi `const`. Funkcijski član može istodobno biti i `const` i `volatile`. I `volatile` kvalifikator je dio potpisa člana te se član može preopteretiti s `volatile` i običnom varijantom. To je ilustrirano sljedećim primjerom:

```
class PrimjerZaVolatile {
public:
    void Funkcija1();               // obična funkcija
    void Funkcija2() volatile;      // volatile funkcija
};

int main() {
    volatile PrimjerZaVolatile a;
    a.Funkcija1(); // pogreška: za volatile objekte se može
                   // pozvati samo volatile funkcijski član
    a.Funkcija2(); // OK: objekt je volatile i funkcijski
                   // član je volatile
    return 0;
}
```

8.8. Statièki èlanovi klase

Klase koje smo u dosadašnjim primjerima deklarirali imale su zasebne skupove èlanova za svaki objekt koji se stvorio. U nekim je sluèajevima, meðutim, potrebno dijeliti podatke izmeðu objekata iste klase. To se može ostvariti statièkim podatkovnim i funkcijskim èlanovima.

8.8.1. Statièki podatkovni èlanovi

Ponekad je potrebno definirati podatkovni èlan koji æe biti zajednièki svim èlanovima klase. To može, na primjer, biti brojaè koji broji koliko je stvoreno objekata te klase ili može biti statusni èlan koji odreðuje ponašanje klase. Taj problem bi se klasiènim pristupom mogao riješiti tako da se deklarira globalna varijabla kojoj pristupaju funkcijski èlanovi klase. No to rješenje nije elegantno jer je globalna varijabla dostupna i izvan klase te joj pristup ne može biti privatan.

Rješenje problema daju statički elementi klase. U tom slučaju varijabla se ne nalazi u globalnom području imena nego u području imena unutar klase te joj je pristup kontroliran, na primjer:

```
#include <iostream.h>

class Brojeni {
private:
    static int Brojac;      // statički podatkovni član
    int MojBroj;
public:
    Brojeni();
    ~Brojeni();
};

Brojeni::Brojeni() : MojBroj(Brojac++) {
    cout << "Stvoren element br.: " << MojBroj << endl;
}

Brojeni::~Brojeni() {
    cout << "Unisten element br.: " << MojBroj << endl;
}

int Brojeni::Brojac = 0;
```

Podatkovni èlan `Brojac` nije sadržan u svakom objektu klase nego je globalan za cijelu klasu. Takav èlan se deklarira tako da se prije specifikacije tipa ubaci kljuèna rijeè `static`. Za statièke èlanove vrijede standardna pravila pristupa. U gornjem sluèaju je podatkovni èlan `Brojac` deklariran privatnim, te mu se može pristupiti samo unutar klase.

Statički članovi su zajednički za sve objekte neke klase. Oni su slični globalnim varijablama s tom razlikom što su njihova imena vidljiva isključivo iz područja klase, te se podvrgavaju pravilima pristupa.

Deklaracija statièkog èlana unutar klase služi samo kao najava ostatku programa da æe negdje u memoriji postojati jedan èlan koji æe biti zajednièki svim objektima klase. Sam èlan time nije smješten u memoriju. Inicijalizacija èlana se mora eksplicitno obaviti izvan klase – tada se odvaja memorijski prostor i èlan se inicijalizira. U gornjem je primjeru to je uèinjeno ovom naredbom:

```
int Brojeni::Brojac = 0;
```

Ona odvaja potreban memorijski prostor i inicijalizira èlan na nulu. U C++ jeziku se deklaracija klase èesto izdvaja u zasebnu datoteku zaglavlja koja se ukljuèuje u sve segmente programa koji koriste klasu. Tako æe deklaracija klase svim dijelovima programa objaviti da æe u memoriji postojati jedan zajednièki èlan. U jednoj datoteci se mora naæi i inicijalizacija tog èlana ili æe se u protivnom dobiti pogreška prilikom

povezivanja kako èlan nije pronaðen u memoriji. Deklaraciju klase `Brojeni` možemo smjestiti u datoteku `broj.h`, a definiciju funkcijskih i statièkih èlanova u datoteku `broj.cpp`. Ako bi se inicijalizacija statièkih èlanova takoðer smjestila u datoteku zaglavlja, prilikom svakog ukljuèenja te datoteke pretprocesorskom direktivom `#include` stvorio bi se po jedan èlan i na kraju bismo prilikom povezivanja dobili poruku o pogreški jer je èlan `Brojeni::Brojac` višestruko definiran.

Iz funkcijskih članova klase statičkom se članu pristupa na isti način kao i običnom članu. Ukoliko statički član ima javno pravo pristupa, može mu se pristupiti i iz ostalih dijelova programa, i to na dva načina. Prvi je način pomoću objekta: navede se objekt, zatim operator `.` pa onda naziv člana. Tada sam objekt služi isključivo tome da se identificira klasa u kojoj je član definiran. Drugi način je pristup bez objekta, samo navodeći klasu. Ispred samog člana se mora navesti naziv klase i operator `::` da bi se naznačilo u kojoj se klasi nalazi član kojemu se pristupa. Opći oblik sintakse za pristup članovima je *`naziv_klase`* `::` *`ime_člana`*, na primjer:

```
class Brojeni {
public:
    static int Brojac; // javni statički član
    int MojBroj;
    // ...
    void IspisiBrojace();
};

void Brojeni::IspisiBrojace() {
    cout << "Brojac: " << Brojac << endl;
    // ovo je pristup iznutra
    cout << "Moj broj: " << MojBroj << endl;
}

int main() {
    Brojeni bb;
    Brojeni::Brojac = 5;     // pristup izvana operatorom ::
    bb.Brojac = 5;           // isto kao i prethodna naredba
    bb.IspisiBrojace();
    return 0;
}
```

Javnim statièkim èlanovima se može pristupati direktno (navodeæi cijelo ime èlana), jer za cijelu klasu postoji samo jedan statièki èlan. Naprotiv, obiènim podatkovnim èlanovima se ne može direktno pristupati, tako da pridruživanje

```
Brojeni::MojBroj = 5;
```

nije ispravno. Ova naredba izaziva pogrešku pri prevoðenju jer `MojBroj` postoji samo u kontekstu odreðenog objekta pa mu se može pristupiti jedino pomoæu objekta:

```
bb.MojBroj = 5;
```

No dozvoljeno je napisati

```
bb.Brojac = 5;
```

Objekt bb sada prevoditelju samo kazuje naziv klase u kojoj je statièki èlan smješten. Gornja naredba ništa ne pridružuju objektu bb, veæ pristupa statièkom èlanu.

Statički podatkovni članovi se dodatno razlikuju od običnih članova po tome što klasa može sadržavati statički objekt te klase. U slučaju običnih članova, klasa može sadržavati samo pokazivače i reference na objekte te klase, na primjer:

```
class Obj {
    // ovo je OK jer je član statički:
    static Obj statickiClan;
    // i ovo je OK jer su članovi referenca i pokazivač:
    Obj &ref, *pok;
    // ovo nije OK:
    Obj nestatickiClan;
};
```

Statièki èlanovi mogu biti navedeni kao podrazumijevani parametri funkcijskim èlanovima, dok nestatièki èlanovi ne mogu, kao u primjeru

```
int a;

class Param {
    int a;
    static int b;
public:
    // ovo je OK jer je član statički:
    void func1(int = b);
    // i ovo je OK jer se odnosi na globalni a:
    void func2(int = ::a);
    // ovo je greška jer se odnosi na nestatički član a:
    void func3(int = a);
};
```

8.8.2. Statièki funkcijski èlanovi

Postoje funkcijski èlanovi koji pristupaju samo statièkim èlanovima klase. Istina, oni se mogu realizirati kao obièni funkcijski èlanovi. Meðutim, ovakvo rješenje ima manu što se za poziv èlana mora stvoriti objekt samo da bi se poziv obavio. To je nepraktièno jer objekt u biti nije potreban; funkcija koju pozivamo pristupa iskljuèivo statièkim èlanovima i ne mijenja niti jedan nestatièki èlan koji ovisi o objektu. Takva se funkcija može deklarirati kao statièka funkcija klase, tako da se ispred povratnog tipa umetne kljuèna rijeè `static`. Za poziv takve funkcije nije potrebno imati objekt, nego se jednostavno navede naziv funkcije, kao da se radi o obiènoj funkciji.

Da bismo ilustrirali koncept statičkih funkcija, dodajmo klasi `Brojeni` statičku funkciju `VrijednostBrojaca()`, koja vraća vrijednost statičkog člana `Brojac`:

```
class Brojeni {
public:
    static int Brojac;
    int MojBroj;
    // ...
    static int VrijednostBrojaca() { return Brojac; }
};

// ...

int main() {
    Brojeni::Brojac = 5;
    cout << Brojeni::VrijednostBrojaca() << endl;
    return 0;
}
```

Iz primjera se vidi da prilikom poziva statièke funkcije nije potrebno navoditi objekt pomoæu kojeg se funkcija poziva, nego se jednostavno navede cijelo ime funkcije (u gornjem primjeru niti jedan objekt klase nije bio deklariran). Važno je uoèiti da statièke funkcije, slièno statièkim podatkovnim èlanovima, imaju puno ime oblika *`naziv_klase`* `::` *`ime_funkcije`*. Ime funkcije pripada podruèju imena klase, a ne globalnom podruèju imena. Zato je potrebno prilikom referenciranja na funkciju navesti ime klase. Za statièke èlanove klasa vrijede pravila pristupa, pa ako funkciju želimo pozivati izvan objekata klase, ona mora imati javni pristup.

Statički funkcijski članovi se mogu pozvati bez objekta, navodeći puno ime funkcije. Kompletno ime obuhvaća naziv klase, odvojen operatorom `::` od naziva člana.

Statièki funkcijski èlanovi se, kao i statièki podatkovni èlanovi, mogu pozvati i pomoæu objekta klase. Sljedeæi poziv je stoga potpuno ispravan:

```
int main() {
    Brojeni bb;
    Brojeni::Brojac = 5;
    cout << bb.VrijednostBrojaca() << endl;
    return 0;
}
```

Kako se statièke funkcije ne pozivaju pomoæu objekta, one niti ne sadržavaju implicitan `this` pokazivaè. Svaka upotreba te kljuène rijeèi u statièkoj funkciji æe rezultirati pogreškom prilikom prevođenja. Kako nema `this` pokazivaèa, jasno je da i pokušaj pristupa bilo kojem nestatièkom podatkovnom ili funkcijskom èlanu klase rezultira

pogreškom prilikom prevođenja. Statièki funkcijski èlanovi ne mogu biti deklarirani kao `const` ili `volatile`.

Dozvoljeno je koristiti pokazivače na statičke podatkovne i funkcijske članove, kao u primjeru

```
// ...
int *pok = &Brojeni::Brojac;
int (*pokNaFunkciju)() = Brojeni::VrijednostBrojaca;
```

Statièkim èlanovima klase može se pristupati i iz drugih modula, odnosno imaju vanjsko povezivanje. O tome æe biti govora u poglavlju o organizaciji kôda.

Zadatak. *U zadatku na stranici 248 deklarirali smo klasu* `Muha`*, te globalnu varijablu koja prati broj muha u računalu. Promijenite mehanizam brojenja muha tako da brojač bude privatan te se čita pomoću statičkog člana. Osim toga, često je potrebno imati vezanu listu svih muha u programu (na primjer, za ubijanje svih muha jednim udarcem). Dodajte statički član* `Muha *zadnja` *koji će pamtiti adresu zadnje stvorene muhe. Također, kako bi imali vezanu listu muha, dodajte nestatičke članove* `prethodna` *i* `sljedeca` *koji će se postavljati u konstruktoru i destruktoru te će pokazivati na prethodnu i na sljedeću muhu u nizu. Dakle, prilikom stvaranja objekta on će se automatski ubaciti u listu, a prilikom uništavanja on će se automatski izbrisati iz liste.*

8.9. Podruèje klase

Svaka klasa ima svoje pridruženo *podruèje* (engl. *scope*). Imena podatkovnih i funkcijskih èlanova pripadaju podruèju klase u sklopu koje su definirani. Dva èlana ne mogu imati isti naziv u jednom podruèju, no èlanovi istih imena mogu nesmetano koegzistirati ako su definirani u razlièitim podruèjima. Globalno definirane varijable, objekti, funkcije, tipovi, klase i pobrojenja pripadaju *globalnom ili datoteènom podruèju* (engl. *global or file scope*). Svi funkcijski èlanovi imaju pristup imenima podruèja klase kojoj pripadaju. To znaèi da, ako se navede ime nekog objekta, ono æe se prvo potražiti u podruèju klase. Ako je potrebno pristupiti objektu iz globalnog podruèja koristi se operator za razluèivanje podruèja `::` (dvije dvotoèke). Promotrimo sljedeæu situaciju:

```
int duljina = 10;

class Polje {
public:
    Polje() : { duljina = ::duljina;
                pokPolje = new int[duljina]; }
private:
    int duljina, *pokPolje;
};
```

U konstruktoru se navodi podatkovni èlan s imenom `duljina`. Iako je on deklariran iza definicije umetnutog konstruktora, prevoditelj æe prepoznati da ime bez operatora `::`

oznaèava ime iz podruèja klase. Sve definicije umetnutih funkcijskih èlanova unutar deklaracije klase se mogu promatrati kao da su navedene izvan deklaracije klase bez promjene u znaèenju, što znaèi da mogu pristupati èlanovima klase deklariranim nakon umetnute definicije. Ako se pak upotrebi operator za razluèivanje podruèja bez navođenja podruèja s lijeve strane, identifikator oznaèava ime iz globalnog podruèja. Identifikator `::duljina` oznaèava globalnu cjelobrojnu varijablu. Operator `::` se može koristiti i za pristup globalnoj varijabli u funkciji u kojoj je deklarirana istoimena lokalna varijabla.

Varijabla ne postoji u području do mjesta do kojeg je navedena njena deklaracija. U primjeru

```
int duljina = 12;

void funkcija() {
    int dulj = duljina;      // referencira se ::duljina
    int duljina = 24;        // globalna duljina postaje
                             // skrivena
    dulj = duljina;          // pridružuje se 24
    int d = ::duljina;       // pristupa se globalnoj varijabli
                             // i pridružuje se 12
}
```

Dobra je programerska praksa izbjegavati zbunjujuæe i nejasne deklaracije varijabli. Ako postoji moguænost nesporazuma prilikom razluèivanja podruèja, dobro je navesti podruèje pomoæu operatora za razluèivanje podruèja.

Tako u primjeru

```
int duljina = 12;

void func() {
    int duljina = duljina; // koja duljina?
}
```

imamo deklaraciju èiji je pravi smisao teško odgonetnuti na prvi pogled. U ovom sluèaju ponašanje prevoditelja ovisi o dosta kompleksnim pravilima na koja se nije dobro oslanjati. Ako i dobro razumijete ta pravila, ne znaèi da æe to razumjeti i ljudi koji æe eventualno biti prisiljeni èitati vaš kôd. U gornjem sluèaju pravilo kaže da se mjestom deklaracije smatra mjesto na kojem je naveden identifikator, pa se stoga `duljina` s desne strane znaka jednakosti odnosi na netom definiranu lokalnu varijablu. Varijabla `duljina` u biti ostaje nedefinirane vrijednosti umjesto da joj se dodijeli vrijednost istoimene globalne varijable.

Svaki funkcijski član također ima svoje područje u koje smješta identifikatore lokalnih varijabli i objekata definiranih u funkciji. Ako se u funkciji deklarira lokalna

varijabla istog imena kao podatkovni član klase tada je član klase sakriven. No može mu se pristupiti pomoću operatora za razlučivanje područja:

```
class Polje {
public:
    Polje() { duljina = 10; pokPolje = new int[duljina]; }
    int Zbroji();
private:
    int duljina, *pokPolje;
};

int Polje::Zbroji() {
    int duljina = Polje::duljina;   // dohvaća se statički
                                    // član klase
    int s = 0;
    while (duljina) s += pokPolje[--duljina];
    return s;
}
```

Iako se primjer mogao realizirati znatno elegantnije, dobro demonstrira upotrebu operatora za razluèivanje podruèja. Operator `::` se koristi tako da se s lijeve strane navede naziv klase èijem podruèju se želi pristupiti dok se s desne strane navede identifikator iz tog podruèja. Ako se ne navede ime klase, pristupa se globalnom podruèju.

Puni naziv pojedinog èlana klase ukljuèuje naziv klase koji se operatorom `::` odvaja od naziva èlana. Kompletan naziv se mora navesti ako želimo dohvatiti èlan u nekom drugom podruèju imena.

8.9.1. Razluèivanje podruèja

Kada se identifikator navede u programu bez izrièito navedenog podruèja kojem on pripada, koriste se pravila za *razluèivanje podruèja* (engl. *scope resolution*) kako bi se ono jednoznaèno odredilo. Taj postupak se provodi prema sljedeæem skupu pravila:

1. Pretražuju se deklaracije u tekućem bloku. Ako se identifikator pronađe, područje je određeno. U suprotnom se pretražuje područje unutar kojeg je blok smješten.
2. Ako je blok smješten unutar funkcijskog člana, pretražuje se područje funkcijskog člana. Pronađe li se lokalna deklaracija koja odgovara identifikatoru, područje je određeno. U protivnom se pretražuje područje klase.
3. Pretražuje se područje klase. Ako se pronađe podatkovni član istog naziva kao i identifikator čije se područje traži, identifikator se pridružuje tom podatkovnom članu te je područje određeno. U suprotnom se pretražuje nadređeno područje.
4. Ako je klasa nenaslijeđena i definirana u globalnom području, nadređeno područje je globalno područje. Ako se pronađe globalna deklaracija, područje je određeno. U suprotnom se prijavljuje pogreška da identifikator nije pronađen. Ako je klasa

nenaslijeđena i definirana kao ugniježđena klasa, nadređeno područje je područje klase u koju je klasa ugniježđena. Poglavlje o ugniježđenim klasama objašnjava ovaj slučaj detaljnije. Za nenaslijeđene klase definirane u lokalnom području nadređeno područje je područje bloka u kojem je klasa definirana. Za naslijeđene klase nadređeno područje čine sva područja klasa od kojih klasa nasljeđuje.

Ako je identifikator naveden pomoæu operatora za razluèivanje podruèja, pretražuje se samo ono podruèje koje je navedeno (podruèje klase ili globalno podruèje). Sve navedeno vrijedi, kako za identifikatore podatkovnih èlanova i varijabli, tako i za identifikatore funkcijskih èlanova i funkcija.

Ime u nekom području sakrit će isto ime u nadređenom području bez obzira što ta dva imena označavaju različite entitete.

Tako æe primjerice lokalna deklaracija objekta sakriti globalnu funkciju istog imena u nadreðenom podruèju. Funkcijski èlan istog imena kao i globalna funkcija sakrit æe identifikator globalne funkcije bez obzira na razliku u parametrima, na primjer:

```
int duljina, func1(), func2(int);

class IgraSkrivaca {
public:
    void func1(int);          // skriva ::func1()
    int func2;                // skriva ::func2(int)
    void duljina(int, float); // skriva ::duljina
};
```

Reference na globalne identifikatore `duljina`, `func1()` i `func2(int)` moraju unutar klase biti navedene kao `::duljina`, `::func1()` i `::func2(int)`. Na primjer

```
void IgraSkrivaca::duljina(int, float) {
    func1(); // pogreška: poziva se IgraSkrivaca::func1(int)
             // a ne ::func1()
}
```

8.10. Ugniježðene klase

Klase koje smo do sada deklarirali bile se smještene u globalnom podruèju, što znaèi da se njihovim imenima može pristupiti iz bilo kojeg dijela programa. Klase se, osim toga, mogu deklarirati u podruèju klase i lokalno. Tako deklarirane klase zovu se još i *ugniježðene klase* (engl. *nested classes*).

Primjenu ugniježđenih klasa ilustrirat ćemo primjerom. Zamislimo da želimo ostvariti mehanizam za praćenje alokacije memorije te ćemo napraviti objekt koji je sposoban pratiti listu alociranih dijelova memorije. Prilikom alokacije segmenta

memorije pozvat ćemo funkcijski član objekta i registrirati alokaciju. Kada memorija više ne bude potrebna, pozvat ćemo funkcijski član koji će izbrisati alokaciju iz liste alokacija. Koristi od takvog sustava mogu biti višestruke prilikom traženja pogrešaka u programu budući da je jedna od najčešćih pogrešaka neoslobađanje zauzetih segmenata memorije. Program normalno funkcionira sve do trenutka kada se količina slobodne memorije ne smanji tako da program ne može alocirati dodatnu memoriju. Pomoću sistema praćenja alokacije memorije mogli bismo na kraju programa provjeriti jesu li svi zauzeti segmenti vraćeni sistemu.

Bilo bi vrlo korisno također imati nekakav pokazatelj koji bi ukazivao na mjesto na kojem je alokacija bila učinjena. ANSI/ISO C++ specifikacija jezika nudi predefinirane globalne simbole `__LINE__` i `__FILE__` (dvostruki znak podcrtavanja se nalazi ispred i iza riječi). Simbol `__LINE__` ima numeričku vrijednost i pokazuje broj linije u kojoj je simbol naveden, dok simbol `__FILE__` ima vrijednost znakovnog niza i daje naziv datoteke u kojoj je linija navedena. Na kraju programa bismo mogli ispisati brojeve linija i nazive datoteka u kojima je obavljena alokacija.

Takav napredan sistem manipulacije memorijom realizirat ćemo klasom koja će objedinjavati mehanizme za dodavanje i brisanje alokacije te za ispisivanje liste alokacija. Svaka alokacija će biti predstavljena jednim objektom koji će se smještati u listu prilikom svakog alociranja. Klasa tog objekta ne mora biti dostupna ostatku programa – jedino klasa memorijskog alokatora može raditi s njom. Zbog toga je najbolje takvu klasu deklarirati u području klase alokatora te se time njeno ime uklanja iz globalnog područja. Deklaracija se smješta u javni, privatni ili zaštićeni dio deklaracije klase. Evo primjera kôda za memorijski alokator:

```
#include <alloc.h>
#include <string.h>
#include <iostream.h>

class MemorijskiAlokator {
private:

    // ugniježđena klasa
    class Alokacija {
    public:
        Alokacija *slijed, *pret;
        size_t velicina;
        void *pocetak;
        int linija;
        char *datoteka;

        Alokacija(size_t vel, void *pok, int lin,
                  char *dat);
        ~Alokacija();
        void Ispisi();
    };

    Alokacija *prva, *zadnja;
```

```
public:
    MemorijskiAlokator() : prva(NULL), zadnja(NULL) {}
    ~MemorijskiAlokator();
    void Dodaj(size_t vel, void *pok, int lin,
               char *dat = NULL);
    int Brisi(void *pok);
    void Ispisi();
};

MemorijskiAlokator::Alokacija::Alokacija(size_t vel,
    void *pok, int lin, char *dat) :
        velicina(vel), pocetak(pok), linija(lin),
        datoteka(new char [strlen(dat) + 1]),
        slijed(NULL), pret(NULL) {
    if (dat)
        strcpy(datoteka, dat);
    else
        datoteka[0] = 0;
}

MemorijskiAlokator::Alokacija::~Alokacija() {
    delete [] datoteka;
}

void MemorijskiAlokator::Alokacija::Ispisi() {
    cout << datoteka << ":" << linija << "  " << velicina;
    cout << "  " << pocetak << endl;
}
```

Klasa `Alokacija` sadrži podatke o svakoj alokaciji: broj linije i naziv datoteke, adresu poèetka bloka i velièinu. Posjeduje takoðer pokazivaèe na prethodni i na sljedeæi element. Vrlo je važno primijetiti da je naziv klase sada dio podruèja klase `MemorijskiAlokator`. Prilikom definicije èlanova klase izvan deklaracije potrebno je navesti punu stazu do imena klase, a to je `MemorijskiAlokator::Alokacija`. Klasa na uobièajeni naèin pristupa svojim podatkovnim èlanovima. Može se reæi da je klasa `Alokacija` ugniježðena, dok je klasa `MemorijskiAlokator` *okolna klasa* (engl. *enclosing class*). Okolna klasa nema nikakvih posebnih privilegija prilikom pristupa èlanovima ugniježðene klase i obrnuto. Slijedi definicija klase `MemorijskiAlokator`:

```
MemorijskiAlokator::~MemorijskiAlokator() {
    Alokacija *priv = prva, *priv1;
    while (priv) {
        priv1 = priv;
        priv = priv->slijed;
        delete priv1;
    }
}

void MemorijskiAlokator::Dodaj(size_t vel, void *pok,
                              int lin, char *dat) {
```

```
    Alokacija *alo = new Alokacija(vel, pok, lin, dat);
    alo->pret = zadnja;
    alo->slijed = NULL;
    if (zadnja)
        zadnja->slijed = alo;
    else
        prva = alo;
    zadnja = alo;
}

int MemorijskiAlokator::Brisi(void *pok) {
    Alokacija *priv = prva;
    while (priv) {
        if (priv->pocetak == pok) {
            if (priv->slijed)
                priv->slijed->pret = priv->pret;
            else
                zadnja = priv->pret;
            if (priv->pret)
                priv->pret->slijed = priv->slijed;
            else
                prva = priv->slijed;
            return 1;
        }
        priv = priv->pret;
    }
    return 0;
}

void MemorijskiAlokator::Ispisi() {
    Alokacija *priv = prva;
    while (priv) {
        priv->Ispisi();
        priv = priv->slijed;
    }
}
```

Ugniježðena klasa ne može direktno referencirati podatkovni ili funkcijski nestatièki èlan okolne klase. Važno je uoèiti da su objekti okolne i ugniježðene klase meðusobno neovisni. Na primjer, stvaranje objekta okolne klase ne implicira stvaranje objekta ugniježðene i obrnuto. Zato je pokušaj pristupa nekom èlanu okolne klase iz ugniježðene klase ekvivalentan pokušaju pristupa èlanu iz glavnog programa: niti jedan èlan nema smisla sam za sebe ukoliko se ne promatra u kontekstu nekog objekta. Drukèija je situacija sa statièkim èlanovima. Njima Ugniježðena klasa može pristupati direktno, ne navodeæi podruèje eksplicitno.

Područje ugniježđene klase je umetnuto u područje okolne klase. To znači da će član `prva` klase `MemorijskiAlokator` sakriti moguću globalnu varijablu `prva`. Na primjer:

```
int prva;

class MemorijskiAlokator {

    class Alokacija {
    // ...
        void Funkcija();
    };

    Alokacija *prva;
    // ...

};

void MemorijskiAlokator::Alokacija::Funkcija() {
    prva = 0;   // pogreška prilikom prevođenja
}
```

Pokušaj referenciranja globalne varijable `prva` rezultira pogreškom prilikom prevoðenja "Identifikator prva se ne može upotrebljavati bez objekta." Da bi se to ispravilo potrebno je referencirati globalno podruèje operatorom za razluèivanje podruèja. Korektna definicija funkcijskog èlana glasi:

```
void MemorijskiAlokator::Alokacija::Funkcija() {
    ::prva = 0;  // sada je OK
}
```

Upravo zbog takvog algoritma prilikom razluèivanja podruèja nije potrebno navoditi podruèje prilikom referenciranja statièkog èlana. Statièki èlan ima smisla i bez objekta klase. Ugniježðena klasa takoðer može referencirati pobrojenja, tipove i druge klase definirane u okolnoj klasi pod pretpostavkom da ugniježðena klasa ima pravo pristupa.

Ukoliko se ugniježđena klasa nalazi unutar javnog dijela tijela okolne klase, mogu se objekti te klase koristiti i u glavnom programu. Prilikom definiranja takvih objekata potrebno je navesti potpuno ime klase pomoću operatora za razlučivanje područja. Pretpostavimo da je deklaracija klase `MemorijskiAlokator` ovako napisana:

```
class MemorijskiAlokator {
private:
    // ... privatni clanovi
public:

    class Alokacija {
    public:
        Alokacija *slijed, *pret;
        // ...
    };

};
```

Sada je moguæe u glavnom programu deklarirati objekt klase `Alokacija`, pri èemu je potrebno navesti puno ime ugniježðene klase:

```
int main() {
    // deklaracija objekta ugniježđene klase
    MemorijskiAlokator::Alokacija aloc, aloc1;
    aloc.slijed = NULL;
    aloc.pret = &aloc1;
    // ...
    return 0;
}
```

Nakon definicije objekta `aloc` gdje je klasa referencirana pomoæu operatora za razluèivanje podruèja, moguæe je posve legalno pristupati njegovim javnim èlanovima.

Puni naziv ugniježðenih klasa obuhvaæa naziv okolne klase odvojen operatorom `::` od naziva klase.

8.11. Lokalne klase

Klasa se takoðer može deklarirati u sklopu lokalnog podruèja funkcije ili funkcijskog èlana. Njena je definicija vidljiva samo unutar bloka u kojem je definirana te u blokovima unutar tog bloka. Za razliku od globalnih i ugniježðenih klasa, lokalna klasa mora biti potpuno definirana unutar deklaracije. To znaèi da svi funkcijski èlanovi moraju biti umetnuti. Time je donekle ogranièena upotreba lokalnih klasa na klase èiji funkcijski èlanovi obavljaju posao od nekoliko linija kôda. Lokalna klasa ne može deklarirati statièke èlanove.

Funkcija u sklopu koje je klasa definirana nema nikakvih posebnih prava pristupa privatnim i zaštićenim članovima lokalne klase. Pravo pristupa se može dodijeliti tako da se funkcija deklarira prijateljem klase. No skrivanje informacija u slučaju lokalnih klasa nema previše smisla: lokalna klasa je vidljiva samo unutar funkcije gdje je definirana tako da niti jedna druga funkcija ne može pristupiti članovima lokalne klase. Zato su najčešće svi članovi lokalne klase javni. Lokalna klasa može pristupiti samo statičkim varijablama, tipovima i pobrojenjima definiranim u sklopu funkcije. Klasa ne može pristupati nestatičkim automatskim varijablama. Evo primjera za deklaraciju lokalne klase:

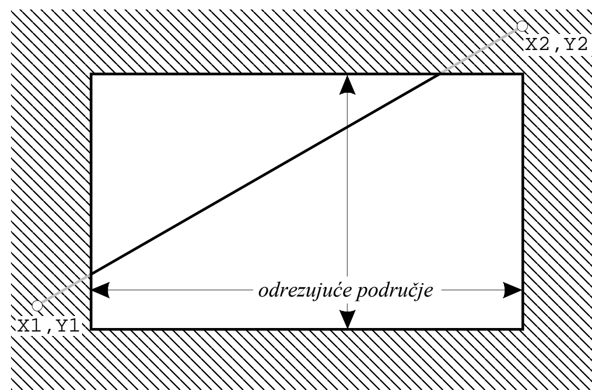
```
void SaLokalnomKlasom(int duljina) {
    static int sirina = 8;
    typedef void (*pokFunc)(int, int);
    class Lokalna {
    public:
        pokFunc pf;          // OK: pokFunc je lokalni tip
        int broj;
```

```
        void Clan() {
            broj = sirina;   // OK: širina je statička
                             // varijabla
            broj += duljina; // pogreška: duljina je
                             // automatska varijabla
        }
    };
    Lokalna lokVar;
    // ... tijelo funkcije
}
```

8.12. Pokazivaèi na èlanove klase

Jezik C++ je opæenamjenski jezik što znaèi da su njegova sintaksa i semantika dizajnirane tako da zadovolje veæinu zahtjeva koji se mogu postaviti prilikom rješavanja praktiènih problema. Važan cilj kojemu se težilo je opæenitost kôda. To znaèi da se jednom napisani kôd može koristiti za razlièite svrhe. Mehanizam koji omoguæava veæi stupanj opæenitosti nude *pokazivaèi na èlanove klasa* (engl. *class member pointers*). To je novi tip podataka koji se konceptualno razlikuje od obiènih pokazivaèa te æe njihova primjena biti objašnjena na sljedeæem primjeru.

Zamislimo da razvijamo programski paket za vektorsko crtanje. Jedna od funkcija takvog programa je rezanje svih linija izvan određenog pravokutnog područja (engl. *clipping*). Kako se crtež pretežno sastoji od niza linija, razvit ćemo klasu koja će objedinjavati podatke o liniji. Klasa će se, prirodno, zvati `Linija`. Podaci o položaju linije na ekranu će se pohranjivati pomoću četiri cjelobrojna člana (`X1`, `Y1`, `X2` i `Y2`) koji će pamtiti koordinate početne i krajnje točke linije u pravokutnom koordinatnom sustavu. Klasa će sadržavati funkcijski član `Odrezi()` koji će za parametar dobiti koordinate gornjeg lijevog i donjeg desnog kuta pravokutnika te će njegova zadaća biti podešavanje koordinata točaka linije tako da linija bude posve unutar zadanog



Slika 8.4. Dijelovi linije koji padaju izvan područja odrezivanja se uklanjaju

pravokutnika. Na slici 8.4 je prikazan taj problem.
Ako se malo pomnije udubimo u naš problem, primjeæujemo da se zadatak svoðenja linije unutar pravokutnika sastoji od dva suštinski ista dijela: prvo je potrebno odrezati koordinate poèetne, a zatim koordinate završne toèke. Postupak koji se primjenjuje u oba sluèaja je isti, jedina je razlika što se jednom u proraèunu koriste èlanovi `X1` i `Y1`, a u drugom èlanovi `X2` i `Y2`. Kako je sam postupak proraèunavanja koordinata toèaka dovoljno složen, svaki ozbiljniji programer æe težiti tome da riješi proraèun opæenito, a zatim ga primjeni jednom na poèetnu, a drugi put na završnu toèku. Jedno od rješenja koje se samo od sebe nameæe je smještanje algoritma za proraèun jedne toèke u zasebnu funkciju kojoj se prosljeðuju pokazivaèi na cjelobrojne varijable s koordinatama toèke za koju se proraèun obavlja. Ovakvo rješenje, iako dosta razborito, nije primjenjivo u našem sluèaju.

Problem leži u tome što je adresa pojedinih članova za svaki objekt drukčija. To je samo po sebi jasno, jer je svaki objekt smješten u zaseban memorijski prostor pa su i adrese pojedinih članova unutar objekta različite. Zato prilikom pisanja klase ne možemo unaprijed znati memorijsku adresu pojedinih članova. Ono što možemo znati su relativne udaljenosti pojedinih članova od početka objekta, a one su za pojedini tip objekata uvijek iste. Na primjer, podatkovni članovi objekata klase `Vektor` su u svim objektima smješteni jedni iza drugih po istom redoslijedu. Član `ax` se nalazi na početku objekta, dok je član `ay` smješten dva bajta od početka objekta. Stoga bismo funkciji koja obavlja proračun mogli proslijediti kao parametre udaljenosti članova.

Pokazivači na elemente klasa omogućavaju upravo to. Općenito, oni mogu pokazivati na podatkovne i na funkcijske članove.

8.12.1. Pokazivaèi na podatkovne èlanove

Neka je klasa `Linija` definirana na sljedeæi naèin:

```
class Linija {
public:
    int X1, Y1, X2, Y2;
    // ... Ovdje ćemo kasnije umetnuti
    // potrebne funkcijske članove
};
```

Podatkovnim èlanovima koji pamte koordinate dali smo javni pristup kako bismo demonstrirali upotrebu pokazivaèa na njih. Pokazivaè na cjelobrojni èlan klase `Linija` ima sljedeæi tip:

```
int Linija::*
```

U deklaraciji je navedeno da pokazivaè pokazuje na cjelobrojni èlan, a èlan pripada klasi `Linija`. Deklaracija varijable `pokKoord` koja pokazuje na èlan klase `Linija` izgleda ovako:

```
int Linija::*pokKoord;
```

Važno je uoèiti suštinsku razliku između pokazivaèa na cjelobrojnu varijablu i na cjelobrojni èlan klase. Neispravno je primijeniti operator za uzimanje adrese na èlan klase i dodijeliti rezultat obiènom pokazivaèu

```
int *pokClan = &Linija::X1;             // neispravno
```

zato jer `X1` nije smješten negdje u memoriji raèunala. To ne treba miješati s dohvaæanjem adrese èlana konkretnog objekta:

```
Linija objLinija;
int *pokClanObjekta = &objLinija.X1;    // ispravno
```

U ovom sluèaju uzima se adresa èlana koji je dio objekta. Kako je objekt stvoren na toèno određenom mjestu u memoriji, adresa èlana je u potpunosti poznata te je njeno uzimanje sasvim legalno.

Adresu člana klase možemo dohvatiti pomoću operatora `&` te pridružiti pokazivaču na član klase:

```
int Linija::*pokKoord = &Linija::X1;    // ispravno
```

Sada smo varijabli `pokKoord` pridružili podatak koji identificira pojedini èlan klase `Linija`. Valja primijetiti da se adresa realnog èlana klase ne može pridružiti pokazivaèu na cjelobrojni èlan. Također, nije moguæe pridružiti adresu èlana jedne klase pokazivaèu na èlan neke druge klase:

```
class Elipsa {
public:
    int cx, cy, velikaPoluos;
    float ekscentricitet;
};

class Linija { /* ... */ };

// ...

int Elipsa::*pokNaCjelobrojni = &Elipsa::ekscentricitet;
// neispravno: pokazivač pokazuje na cjelobrojni član,
// a pokušano mu je pridružiti adresu realnog člana

int Linija::*pokNaClanLinije = &Elipsa::cx;
// neispravno: pokazivač pokazuje na član klase Linija,
// a pridružena mu je adresa člana klase Elipsa

float Elipsa::*pokNaClanElipse = &Elipsa::ekscentricitet;
// ispravno: pokazivači se slažu po tipu i po klasi
```

Svrha pokazivaèa na èlan klase je identifikacija pojedinog èlana. Najèešæe se pokazivaèi na èlanove implementiraju kao brojevi koji pokazuju udaljenost èlana od poèetka objekta.

Pokazivaè na èlan klase ne sadrži adresu nekog konkretnog objekta u memoriji, veæ njegova vrijednost jednoznaèno identificira neki èlan klase.

Pokazivaèi na èlanove se uvijek koriste u kontekstu nekog objekta. Pri tome postoje dva specifièna C++ operatora koji omoguæavaju pristup èlanovima preko pokazivaèa, a to su `.*` (toèka i zvjezdica) te `->*` (minus, veæe od i zvjezdica). Operator `.*` se koristi tako da se s lijeve strane nalazi objekt kojemu se želi pristupiti, a s desne strane pokazivaè na èlan. Operatoru `->*` se s lijeve strane navede pokazivaè na objekt kojemu se želi pristupiti dok mu se s desne strane navede pokazivaè na èlan. Demonstrirajmo upotrebu operatora na sljedeæem primjeru:

```
Linija objLinija, *pokLinija = &objLinija;

int main() {
    int Linija::*pokCjelobrojni = &Linija::X1;

    // pristup preko operatora .*
    objLinija.*pokCjelobrojni = 7;

    pokCjelobrojni = &Linija::Y1;

    // pristup preko operatora ->*
    pokLinija->*pokCjelobrojni = 9;

    cout << objLinija.X1 << endl << objLinija.Y1 << endl;
    return 0;
}
```

Nakon prevoðenja i izvoðenja gornjeg primjera dobit æe se ispis

```
7
9
```

Pristup podacima preko pokazivaèa na èlanove se obavlja pomoæu operatora `->*` (minus, veæe od i zvjezdica) i `.*` (toèka i zvjezdica).

Za razliku od operacija dozvoljenih nad obiènim pokazivaèima, skup dozvoljenih operacija nad pokazivaèima na èlanove je sužen. Kako pokazivaèi na èlanove ne pokazuju na stvarno mjesto u memoriji raèunala, nije ih moguæe implicitno konvertirati

u tip `void *`. Također nema implicitne konverzije u aritmetièki tip te nije podržana pokazivaèka aritmetika. To znaèi da naredbom

```
if (pokNaClan)      // neispravno: nema konverzije
```

nije moguæe provjeriti je li pokazivaè nul-pokazivaè.

Moguće je definirati polje pokazivača na član. Postavljanje svih članova klase `Linija` na nulu može se obaviti pomoću polja pokazivača:

```
void PostaviNaNulu(Linija &refLinija) {
    int Linija::*pokLinija[] = {&Linija::X1,
                                &Linija::Y1,
                                &Linija::X2,
                                &Linija::Y2 };
    for (int i = 0; i < 4; i++)
        refLinija.*pokLinija[i] = 0;
}
```

Iako je ovakav naèin brisanja elemenata u najmanju ruku bizaran te se jednostavnije može obaviti pomoæu èetiri pridruživanja, ima prednosti u sluèaju kada je broj èlanova klase velik.

Pokazivače na članove moguće je prosljeđivati kao parametre funkcijama. Prikažimo rješenje problema svođenja linije u pravokutnik pomoću pokazivača na članove.

```
class Linija {
private:
    int X1, Y1, X2, Y2;
    void OdreziTocku(int Linija::*pok1,
                     int Linija::*pok2,
                     int v1, int v2, int p, int tip);

public:
    void Odrezi(int px1, int py1, int px2, int py2);
};

void Linija::Odrezi(int px1, int py1, int px2, int py2) {
    OdreziTocku(&Linija::X1, &Linija::Y1,
                X2 - X1, Y2 - Y1, px1, 1);
    OdreziTocku(&Linija::X1, &Linija::Y1,
                X2 - X1, Y2 - Y1, px2, 0);
    OdreziTocku(&Linija::Y1, &Linija::X1,
                Y2 - Y1, X2 - X1, py1, 1);
    OdreziTocku(&Linija::Y1, &Linija::X1,
                Y2 - Y1, X2 - X1, py2, 0);
    OdreziTocku(&Linija::X2, &Linija::Y2,
                X1 - X2, Y1 - Y2, px1, 1);
    OdreziTocku(&Linija::X2, &Linija::Y2,
```

```
                X1 - X2, Y1 - Y2, px2, 0);
    OdreziTocku(&Linija::Y2, &Linija::X2,
                Y1 - Y2, X1 - X2, py1, 1);
    OdreziTocku(&Linija::Y2, &Linija::X2,
                Y1 - Y2, X1 - X2, py2, 0);
}

void Linija::OdreziTocku(int Linija::*pok1,
                         int Linija::*pok2,
                         int v1, int v2, int p, int tip) {
    if ((this->*pok1 < p && tip) ||
        (this->*pok1 > p && !tip)) {
        float param = (p - this->*pok1)/((float)v1);
        if ((param > 0 && tip) || (param < 1 && !tip)) {
            this->*pok2 = this->*pok2 + param * v2 + 0.5;
            this->*pok1 = p;
        }
        else
            this->*pok1 = this->*pok2 = -1;
    }
}
```

Objasnimo naèin rada ovog programa. Funkcijski èlan `Odrezi()` je zadužen za svoðenje linije unutar zadanog pravokutnika pri èemu parametri `px1`, `py1`, `px2` i `py2` odreðuju koordinate gornjeg lijevog i donjeg desnog ugla pravokutnika. Funkcijski èlan poziva `OdreziTocku()` osam puta.

Funkcijski član `OdreziTocku()` ima za parametre dvije koordinate točke te dvije cjelobrojne varijable `v1` i `v2`. Točka i par (`v1`, `v2`) zajedno određuju parametarsku jednadžbu pravca koji ima vektor smjera određen pomoću `v1` i `v2`, a prolazi kroz zadanu točku. Parametar `p` određuje koordinatu pravca u odnosu na koji se promatra položaj zadane točke. Parametar `tip` vrijednošću 1 određuje da točka mora imati prvu koordinatu veću od koordinate pravca, dok vrijednošću 0 određuje da točka mora imati prvu koordinatu manju od koordinate pravca. Ako se testom ustanovi da se točka nalazi s pogrešne strane promatranog pravca, izračuna se vrijednost parametra u kojoj točka točno leži na pravcu te se pomoću njega izračuna vrijednost druge koordinate. Radi pravilnog zaokruženja prilikom konverzije realne vrijednosti u cjelobrojnu, dodaje se vrijednost 0.5.

Algoritam podešavanja koordinate na pravac je podjednak kada promatramo x koordinatu početne točke te pravce x = `px1` i x = `px2` kao i kada promatramo y koordinatu završne točke te pravce y = `py1` i y = `py2`. Zato on rješava općeniti slučaj. Funkcijski član `OdreziTocku()` provjerava odnose između člana na koji pokazuje `pok1` i pravca određenog s `p`, te pomoću njih izračunava vrijednost člana na koji pokazuje `pok2`. Funkcijski član `Odrezi()` je zadužen da primjereno pozove općeniti algoritam, prosljeđujući jednom pokazivač na član `X1` kao prvi parametar i pokazivač na član `Y1` kao drugi parametar, a drugi put pokazivač na `Y1` kao prvi parametar i pokazivač na `X1` kao drugi parametar.

Važno je primijetiti da se prilikom pristupanja članovima preko pokazivača koristi operator `->*` kojem se s lijeve strane nalazi ključna riječ `this`. To znači da se pristupa članovima objekta za koji je pozvan funkcijski član `OdreziTocku()`.

8.12.2. Pokazivaèi na funkcijske èlanove

Analogno pokazivaèima na funkcije, moguæe je definirati pokazivaèe na funkcijske èlanove. Pretpostavimo da želimo kontrolirati poziciju nekog grafièkog objekta na ekranu raèunala, pri èemu je svaki objekt prikazan jednim primjerkom klase `GrafObjekt`. Ta klasa sadrži funkcijski èlan `Crtaj()` potreban za crtanje objekta te èlanove `Gore()`, `Dolje()`, `Lijevo()` i `Desno()` koji pomièu objekt u zadanom smjeru za neki unaprijed odreðen korak. Èlanovi imaju cjelobrojnu povratnu vrijednost koja, postavljena na jedinicu, oznaèava da je objekt pomaknut do ruba ekrana. Inaèe je povratna vrijednost nula. Također pretpostavimo da postoji funkcijski èlan `PostaviVelicinu()` s dva parametra koji definiraju željenu velièinu i širinu objekta.

Željeli bismo dodati član `PonoviPomak()` koji bi ponavljao pomak u željenom smjeru određeni broj puta. Kada bi algoritmi za pomak bili smješteni u funkcijama te ne bi bili članovi klasa, funkcija `PonoviPomak()` bi kao parametar uzimala pokazivač na funkciju pomaka i broj ponavljanja. No takvo rješenje nije moguće, jer radimo s funkcijskim članovima klase, a ne s običnim funkcijama. Ne možemo, primjerice, odrediti pokazivač na funkciju `Gore()` – potrebno je koristiti pokazivače na funkcijske članove. Slično pokazivačima na podatkovne članove, pokazivači na funkcijske članove ne određuju direktno adresu funkcije na koju pokazuju, već samo identificiraju određeni funkcijski član klase. Da bismo pozvali funkcijski član pomoću pokazivača na njega, potrebno je navesti objekt ili pokazivač na objekt za koji se funkcijski član poziva kao u sljedećem primjeru:

```
class GrafObjekt {

// ... detalji implementacije su nebitni

public:

    void Crtaj();
    int Gore();
    int Dolje();
    int Lijevo();
    int Desno();
    void PostaviVelicinu(int sirina, int visina);

    // funkcija PonoviPomak ima parametar smjer koji je tipa
    // pokazivač na funkcijski član klase GrafObjekt koji ne
    // uzima parametre i vraća cijeli broj:
    void PonoviPomak(int (GrafObjekt::*smjer)(), int puta);

};
```

Pokazivaè na funkcijski èlan se deklarira tako da se navede tip povratne vrijednosti, klasa funkcije i njeni parametri; na primjer deklaracija pokazivaèa `pokNaFClan` na funkcijski èlan klase `GrafObjekt` koji vraæa cjelobrojnu vrijednost i nema parametara izgleda ovako:

```
int (GrafObjekt::*pokNaFClan)();
```

U zagradama se navode parametri funkcijskog èlana. Ako se želi definirati pokazivaè na èlan koji uzima dva cjelobrojna parametra i ne vraæa vrijednost piše se:

```
void (GrafObjekt::*pokNaDrugiClan)(int, int);
```

Vrijednost pokazivaèu se pridodjeljuje na sljedeæi naèin:

```
pokNaFClan = GrafObjekt::Gore;
```

Kao i kod pokazivaèa na funkcije, nije potrebno navoditi operator za uzimanje adrese prije navođenja funkcijskog èlana. Tako je zapis pokazivaèa `&GrafObjekt::Gore` i `GrafObjekt::Gore` ekvivalentan.

Prilikom pozivanja funkcijskog člana pomoću pokazivača na njega koriste se operatori `.*` i `->*` pri čemu se s lijeve strane operatora navodi objekt za operator `.*` odnosno pokazivač na objekt za operator `->*`, a s desne strane pokazivač na funkcijski član. Iza njega se u zagradama navode mogući parametri funkcijskog člana. Evo primjera:

```
GrafObjekt grObj, *pokNaGrObj = &grObj;

// uzimanje adrese funkcijskog člana
int (GrafObjekt::*bezparam)() = GrafObjekt::Gore;

// poziv funkcijskog člana preko pokazivača
(grObj.*bezparam)();

void (GrafObjekt::*dvaparam)(int, int);
dvaparam = GrafObjekt::PostaviVelicinu;
(pokNaGrObj->*dvaparam)(5, 6);  // poziv preko pokazivača
```

Upotreba zagrada oko naziva funkcije i objekta je obavezna zato jer operator `()` za poziv funkcije ima veæi prioritet izvođenja. U gore navedenom pozivu funkcijskog èlana na koji pokazuje `bezparam` redoslijed izvođenja operatora je sljedeæi: najprije se pomoæu

```
grObj.*bezparam
```

pokazivaè na èlan pretvori u pokazivaè na funkciju, na koju se zatim primijeni operator za poziv. Naprotiv, naredba

```
grObj.*bezparam();
```

bi se interpretirala kao

```
grObj.*(bezparam());
```

što ima znaèenje “pozovi funkciju `bezparam()` i njenu povratnu vrijednost veži za operator `.*` te pristupi èlanu”. Takav poziv rezultira pogreškom prilikom prevođenja jer `bezparam` nije pokazivaè na funkciju.

Pokazivači na funkcijske članove su, baš kao i pokazivači na funkcije, ovisni o potpisu funkcije. To znači da je pokazivaču moguće dodjeljivati samo pokazivače na članove koji imaju istu povratnu vrijednost i iste parametre te pripadaju istoj klasi:

```
bezparam = GrafObjekt::PostaviVelicinu;
// pogrešno: PostaviVelicinu ima dva parametra, a
// bezparam je deklariran kao pokazivač na član
// koji nema parametara

bezparam = GrafObjekt::Crtaj;
// pogrešno: Crtaj ne vraća vrijednost, a
// bezparam je deklariran kao pokazivač na član
// koji vraća cjelobrojnu vrijednost

bezparam = GrafObjekt::Gore;   // OK
bezparam = GrafObjekt::Lijevo; // OK

class DrugaKlasa {
public:
    int FunkClan();
};

bezparam = DrugaKlasa::FunkClan;
// pogrešno: bezparam je definiran kao pokazivač na
// funkcijski član klase GrafObjekt, a ne klase DrugaKlasa
```

Pokazivaèi na funkcijske èlanove jednoznaèno su određeni tipom povratne vrijednosti i potpisom funkcijskom èlana te klasom kojoj pripadaju.

Pokazivaèi na funkcijske èlanove imaju isti skup dozvoljenih operacija kao i pokazivaèi na podatkovne èlanove.

Promotrimo rješenje našeg problema pomicanja objekta. Dan je funkcijski član `PonoviPomak()` koji ponavlja pomak u željenom smjeru `puta` puta:

```
void GrafObjekt::PonoviPomak(int (GrafObjekt::*smjer)(),
                             int puta) {
```

```
        for (int i = 0; i < puta; i++)
            if ((this->*smjer)()) break;
}
```

Parametar `smjer` je pokazivaè na funkcijski èlan koji odreðuje smjer pomaka, te æe pokazivati na `Gore()`, `Dolje()`, `Lijevo()` ili `Desno()`. Broj pomaka je sadržan u parametru `puta`. Ako bismo željeli pomaknuti neki objekt prema dolje èetiri puta, to bismo mogli uèiniti na sljedeæi naèin:

```
GrafObjekt nekiObjekt;

nekiObjekt.PonoviPomak(GrafObjekt::Dolje, 4);
```

Zadatak. *Evo jednog primjera iz života (!). Načinite klasu `Tijelo` koje će opisivati ljudsko tijelo. Pri tome sami izaberite skup organa koje želite uključiti u klasu. Svaki organ neka je predstavljen jednim podatkovnim članom tipa `bool` koji će pokazivati da li dotični organ radi ispravno ili ne. Evo primjera:*

```
class Tijelo {
private:
    bool srce;
    bool lijeviBubreg;
    bool desniBubreg;
    bool jetra;
    bool slezena;
};
```

Napišite funkciju `DoktorePomozite()` koja će kao parametar dobiti pokazivač na objekt klase `Tijelo` te pokazivač na neki član klase koji pokazuje na član koji uzrokuje probleme. Funkcija mora prvo provjeriti je li organ uopće bolestan (tako da provjeri da li je u članu na koji pokazivač pokazuje upisano `true`). Ako nije, potrebno je ispisati poruku o tome da je pacijent hipohondar, a u suprotnom je potrebno "izliječiti" dotični organ tako da se u njega upiše `true`. Obratite pažnju da ta funkcija mora imati pristup unutrašnjosti pacijenta (u punom smislu riječi), pa zbog toga mora biti deklarirana prijateljem klase `Tijelo` (u stvarnosti je to već diskutabilno).

8.13. Privremeni objekti

Objekti kojima smo rukovali do sada su bili uvijek eksplicitno stvoreni: ili su nastali kao posljedica deklaracije ili je memorija za njih dodijeljena pomoæu operatora `new`. Osim takvim izrièitim naredbama, prevoditelj može ponekad sam stvoriti *neimenovani privremeni objekt* (engl. *unnamed temporary*) koji služi za privremeno pohranjivanje vrijednosti. Ako prevoditelj na odreðenom mjestu stvori privremeni objekt, on je odgovoran za njegovo pravovremeno uništavanje. Toèna mjesta na kojima se uvode privremeni objekti, njihovo uništavanje i kopiranje nisu strogo definirana standardom

C++ jezika i u velikoj mjeri ovise o implementaciji prevoditelja. Svi primjeri navedeni u ovom poglavlju su prevedeni pomoæu Borland C++ 4.5 prevoditelja.

8.13.1. Eksplicitno stvoreni privremeni objekti

Privremeni objekt se može eksplicitno stvoriti tako da se navede naziv klase i u okruglim zagradama parametri konstruktora. To svojstvo se èesto koristi kod objekata koji mogu graditi matematièke izraze.

Vratimo se za trenutak natrag na problem vektorskog računa. Zamislimo da želimo napisati funkciju koja zbraja dva vektora. U odsječku 8.2.4 je dan primjer funkcijskog člana `ZbrojiSa()` koji zbraja vektore, no ono što sada želimo postići je nešto sasvim drugo. Član `ZbrojiSa()` se poziva za objekt klase `Vektor` i kao parametre ima *x* i *y* komponente vektora s kojim se vektor zbraja. Sada želimo ostvariti funkciju (ne funkcijski član!) koja će kao parametre imati tri objekta klase `Vektor`. Funkcija će zbrojiti prva dva operanda i rezultat smjestiti u objekt proslijeđen kao treći argument. Ponovimo deklaraciju klase i navedimo kôd funkcije `ZbrojiVektore()`:

```
class Vektor {
friend void ZbrojiVektore(Vektor &a, Vektor &b,
                          Vektor &rez);
private:
    float ax, ay;
public:
    Vektor(float x = 0, float y = 0) : ax(x), ay(y) {}
    float DajX() { return ax; }
    float DajY() { return ay; }
    void PostaviXY(float x, float y) { ax = x; ay = y; }
};

void ZbrojiVektore(Vektor &a, Vektor &b, Vektor &rez) {
    rez.ax = a.ax + b.ax;
    rez.ay = a.ay + b.ay;
}
```

Zbrajanje dvaju vektora možemo obaviti tako da deklariramo tri objekta klase `Vektor` te pozovemo funkciju `ZbrojiVektore()`:

```
Vektor a(10.0, 2.8);
Vektor b(-2.0, 5.0);
Vektor c;

ZbrojiVektore(a, b, c);
```

Ovakav naèin rada je sasvim ispravan i regularan. Objekt `c` æe na kraju sadržavati zbroj vektora `a` i `b`. Jedina je mana što vektor `a`, iako nikada kasnije u programu ne koristimo, moramo deklarirati, i što je još ozbiljnije, za njega odvojiti memorijski prostor koji ostaje zauzet do kraja bloka unutar kojeg je deklariran. Mnogo praktiènije je provesti

raèunsku operaciju tako da stvorimo privremeni objekt klase `Vektor` koji æe “živjeti” samo za vrijeme raèunanja zbroja, a nakon toga æe biti uništen. Trebamo, dakle, naèin da u kôd upišemo “vektorsku konstantu”, ekvivalentnu cjelobrojnim konstantama koje pišemo u klasiènim izrazima.

Privremeni objekt možemo stvoriti tako da navedemo naziv klase i u okruglim zagradama stavimo parametre konstruktoru (ili samo otvorenu i zatvorenu zagradu ako parametre izostavljamo) . Prema tome bismo isti rezultat kao i u gornjem primjeru mogli postići sljedećim pozivom:

```
Vektor c;

ZbrojiVektore(Vektor(10.0, 2.8), Vektor(-2.0, 5.0), c);
```

Iako dovitljivi èitatelj može primijetiti da bi bilo daleko jednostavnije i efikasnije odmah deklarirati vektor `c` i dodijeliti mu vrijednost izraèunatu pomoæu džepnog raèunala, ovaj primjer pokazuje kako se mogu stvoriti privremeni objekti. Prije ulaska u funkciju `ZbrojiVektore()` prevoditelj æe stvoriti dva privremena objekta, a zatim pozvati funkciju – adrese objekata æe biti proslijeðene preko referenci. Kada funkcija završi, prevoditelj æe privremene objekte uništiti.

Život privremenih objekata je ograničen na izraz u kojem se pojavljuju. Na kraju naredbe (kod točke-zarez) svi objekti stvoreni u izrazu se uništavaju. Da bi se mogao pratiti tok stvaranja i uništavanja objekata, možemo proširiti konstruktore i destruktore naredbama koje ispisuju poruke o tome da je objekt stvoren, odnosno uništen. Klasi `Vektor` ćemo također dodati statički član `brojac` koji će brojati stvorene objekte. Svaki objekt će zapamtiti broj pod kojim je stvoren u članu `redbr`. Vrijednost tog člana će se ispisati prilikom stvaranja i uništavanja objekta, te ćemo na taj način moći identificirati objekte.

```
#include <iostream.h>

class Vektor {
friend void ZbrojiVektore(Vektor &a, Vektor &b,
                          Vektor &rez);
private:
    static int brojac;
    int redbr;
    float ax, ay;
public:
    Vektor(float x = 0, float y = 0);
    ~Vektor();
    float DajX() { return ax; }
    float DajY() { return ay; }
    void PostaviXY(float x, float y) { ax = x; ay = y; }
};

int Vektor::brojac = 0;
```

```
Vektor::Vektor(float x, float y) :
        ax(x), ay(y), redbr(++brojac) {
    cout << "Stvoren vektor pod brojem " << redbr << endl;
    cout << "X: " << ax << "    Y: " << ay << endl;
}

Vektor::~Vektor() {
    cout << "Uništen vektor pod brojem " << redbr << endl;
    cout << "X: " << ax << "    Y: " << ay << endl;
}

void ZbrojiVektore(Vektor &a, Vektor &b, Vektor &rez) {
    cout << "Zbrajam" << endl;
    rez.ax = a.ax + b.ax;
    rez.ay = a.ay + b.ay;
    cout << "Zbrojio sam" << endl;
}
```

Prije nego što pogledamo kako se stvaraju i uništavaju privremeni objekti, potrebno je imati na umu da stvaranje i uništavanje tih objekata nije strogo definirano standardom te ovisi o implementaciji prevoditelja. Tako primjerice Borland C++ 2.0 uništava sve objekte na kraju bloka umjesto na kraju izraza. Navedeni ispisi se odnose na Borland C++ 4.5. Štoviše, prilikom manipuliranja privremenim objektima mnogi prevoditelji koriste razne trikove kojima optimiraju izvršenje programa i izbjegavaju nepotrebna kopiranja. Radi toga dobiveni ispis može ovisiti o stupnju optimizacije koju prevoditelj provodi.

U sljedećem programskom odsječku imamo dva eksplicitna poziva konstruktora klase `Vektor` kojima se stvaraju privremeni objekti:

```
int main() {
    cout << "Ulazak u main" << endl;
    Vektor c;
    cout << "Pozivam ZbrojiVektore" << endl;
    ZbrojiVektore(Vektor(10.0, 2.8), Vektor(-2.0, 5.0), c);
    cout << "Završavam" << endl;
    return 0;
}
```

Nakon izvođenja se dobiva sljedeæi ispis:

```
Ulazak u main
Stvoren vektor pod brojem 1
X: 0    Y: 0
Pozivam ZbrojiVektore
Stvoren vektor pod brojem 2
X: -2    Y: 5
Stvoren vektor pod brojem 3
X: 10    Y: 2.8
Zbrajam
```

```
Zbrojio sam
Uništen vektor pod brojem 3
X: 10    Y: 2.8
Uništen vektor pod brojem 2
X: -2    Y: 5
Završavam
Uništen vektor pod brojem 1
X: 8    Y: 7.8
```

Vektor pod brojem 1 je vektor `c` deklariran kao lokalni objekt. Prije ulaska u funkciju se stvaraju privremeni vektori koji se uništavaju odmah nakon povratka iz nje. Program zatim završava te se na samom kraju uništava i objekt 1.

Moguće je pozvati funkcijski član privremenog objekta ili pristupati njegovim podatkovnim članovima. Dodajmo funkcijski član `IspisiVektor()` za ispis vektora te promotrimo rezultat sljedećeg kôda:

```
class Vektor {
// ... ovdje ide deklaracija klase
public:
    void IspisiVektor();
};

void Vektor::IspisiVektor() {
    cout << "Ispis {" << ax << "," << ay << "}" << endl;
}

int main() {
    cout << "Prije izraza" << endl;
    Vektor(12.0, 3.0).IspisiVektor();
    cout << "Nakon izraza" << endl;
    return 0;
}
```

Dobiveni ispis je ovakav:

```
Prije izraza
Stvoren vektor pod brojem 1
X: 12    Y: 3
Ispis {12,3}
Uništen vektor pod brojem 1
X: 12    Y: 3
Nakon izraza
```

8.13.2. Privremeni objekti kod prijenosa parametara u funkciju

U gornjem primjeru parametri funkcije za zbrajanje vektora su bile reference na objekte klase `Vektor`. To znaèi da funkcija nije dobivala kopiju objekta. Ako bi funkcija

mijenjala sadržaj parametara, mijenjao bi se i sadržaj objekta navedenog u parametarskoj listi. To se svojstvo koristi kod parametra `rez`: rezultat izraèunavanja se smješta u objekt na koji pokazuje referenca te tako funkcija vraæa vrijednost.

Česte su situacije kada funkcija mora dobiti kopiju objekta nad kojim obavlja razne proračune. U tom se slučaju izmjene objekta ne odražavaju na objekt koji je naveden kao stvarni parametar u pozivu funkcije. Kopija koju funkcija dobiva stvara se konstruktorom kopije. Dodajmo konstruktor kopije i funkciju koja kao argument dobiva vektor prenesen po vrijednosti te ga postavlja na nul-vektor (korisnost te funkcije je upitna, no poslužit će za demonstraciju).

```
class Vektor {
// ... ovdje idu deklaracije
public:
    Vektor(const Vektor &ref);
};

Vektor::Vektor(const Vektor &ref) :
            ax(ref.ax), ay(ref.ay), redbr(++brojac) {
    cout << "Stvoren vektor pomoću konstruktora kopije "
         << "pod brojem " << redbr << endl
         << "X: " << ax << "    Y: " << ay << endl;
}

void NaNulu(Vektor v) {
    cout << "Ušao u NaNulu" << endl;
    v.PostaviXY(0, 0);
    cout << "Postavio sam" << endl;
}

int main() {
    cout << "Ušao sam u main" << endl;
    Vektor c(12.0, 3.0);
    cout << "Pozivam NaNulu" << endl;
    NaNulu(c);
    cout << "Završavam" << endl;
    return 0;
}
```

Ispis je sljedeæi:

```
Ušao sam u main
Stvoren vektor pod brojem 1
X: 12    Y: 3
Pozivam NaNulu
Stvoren vektor pomoću konstruktora kopije pod brojem 2
X: 12    Y: 3
Ušao u NaNulu
Postavio sam
Uništen vektor pod brojem 2
```

```
X: 0    Y: 0
Završavam
Uništen vektor pod brojem 1
X: 12   Y: 3
```

Najprije je stvoren vektor `c` i inicijaliziran na vrijednosti 12 i 3. Zatim je pozvana funkcija `NaNulu()`. Kako funkcija za parametar ima objekt koji se prenosi po vrijednosti, dobit æe privremenu kopiju vektora `c` koja se stvara pomoæu konstruktora kopije. Ako konstruktor kopije nije definiran, koristi se automatski generirani konstruktor kopije. Zatim se izvodi tijelo funkcije `NaNulu()`. Funkcija postavlja vrijednost privremenog vektora na nulu i završava.

Privremeni objekt nastao kao posljedica prijenosa po vrijednosti se uništava uvijek nakon izlaska iz funkcije i to tako da se poziva destruktor. Iz ispisa se vidi da destruktor uništava objekt koji je postavljen na (0, 0). Zatim završava i glavni program. Uništava se vektor `c` koji i dalje ima vrijednosti 12 i 3 – vidimo da objekt sadržava svoje početne vrijednosti.

Konstruktor kopije mora biti dostupan u dijelu kôda u kojem se parametar prosljeđuje. To u ovom slučaju znači da on mora imati javni pristup. U suprotnom će prevoditelj dojaviti pogrešku prilikom prevođenja da konstruktor kopije nije dostupan.

Prenošenje objekata po vrijednost je očito sporije nego prenošenje po referenci jer se uvijek stvara dodatna kopija objekta nad kojom funkcija radi. Prenošenje objekata po referenci je učinkovitije. No pri tome je potrebno biti vrlo pažljiv, jer pri tome može doći do vrlo neugodnih pogrešaka.

Promotrimo situaciju u kojoj se adresa lokalnog objekta dodjeljuje globalnom pokazivaču. Kako se privremeni objekt uništava nakon završetka funkcije, nakon završetka funkcije pokazivač će pokazivati na memoriju koja više ne sadržava objekt. Takvi *viseći pokazivači* (engl. *dangling pointers*) predstavljaju veliku opasnost po program:

```
Vektor *pok;

void funkcija(Vektor a) {
    pok = &a;
    // ... radi nešto s a
}

int main() {
    Vektor a;
    funkcija(a);
    // pok visi - pokazuje na memoriju koja više
    // nije objekt
    return 0;
}
```

U funkciji se uzima adresa parametra i dodjeljuje globalnom pokazivaèu. Nakon završetka funkcije privremeni objekt se uništava, no `pok` živi i dalje te pokazuje na

memoriju gdje je bio privremeni objekt. I dok pristup podatkovnom èlanu pomoæu tog pokazivaèa vjerojatno neæe srušiti sistem te æe samo vratiti neku nedefiniranu vrijednost, pristup funkcijskom èlanu æe vrlo vjerojatno uzrokovati ispad programa. Također æe pokušaj upisivanja vrijednosti u objekt vrlo vjerojatno rezultirati prepisivanjem preko drugih podataka, što za aplikaciju može biti pogubno. Slièni efekti se mogu postiæi i tako da se reference inicijaliziraju objektima koji se zatim unište.

Pouka ovog primjera je da su viseći pokazivači i reference vrlo opasni po integritet programa. Po mnogim studijama oni su jedni od najčešćih uzroka nepredviđenog kraha mnogih komercijalnih programa te valja biti vrlo oprezan da se ovakve situacije izbjegnu.

Ponekad optimizacije izvedbenog koda koje prevoditelj provodi mogu dovesti do toga da postupak prenošenja objekata odstupa od gore izloženog. Promotrimo sljedeći primjer:

```
int main() {
    cout << "Ušao sam u main" << endl;
    cout << "Pozivam NaNulu" << endl;
    NaNulu(Vektor(12, 3));
    cout << "Završavam" << endl;
    return 0;
}
```

Funkcija `NaNulu()` se u ovom sluèaju poziva tako da joj se za parametar navodi privremeni objekt. Prema gore iznesenim pravilima može se oèekivati da æe taj objekt biti kopiran u privremeni objekt koji æe biti proslijeđen funkciji. Oba privremena objekta bit æe uništena nakon izlaska iz funkcije. No s Borland C++ 4.5 prevoditeljem dobivamo sljedeæi ispis:

```
Ušao sam u main
Pozivam NaNulu
Stvoren vektor pod brojem 1
X: 12    Y: 3
Ušao u NaNulu
Postavio sam
Uništen vektor pod brojem 1
X: 0    Y: 0
Završavam
```

Prevoditelj je izbjegao stvaranje jedne kopije tako što je privremeni objekt direktno proslijedio funkciji. Prevoditelj èak neæe provjeravati dostupnost konstruktora kopije jer ga uopæe neæe pozivati.

Slično se dešava i prilikom inicijalizacije objekata. U poglavlju o konstruktoru kopije rečeno je da se inicijalizacija

```
Vektor a, b = a;
```

interpretira kao “Stvori vektor `a` i zatim stvori vektor `b` tako da konstruktorom kopije kopiraš `a` u `b`”. Inicijalizacija

```
int main() {
    cout << "Ušao sam u main" << endl;
    Vektor a = Vektor(12.0, 3.0);
    // ...
    return 0;
}
```

bi se mogla provesti tako da se stvori privremeni objekt koji se zatim konstruktorom kopije preslika u `a` te se privremeni objekt uništi. No ispis koji se dobiva je sljedeæi:

```
Ušao sam u main
Stvoren vektor pod brojem 1
X: 12    Y: 3
Uništen vektor pod brojem 1
X: 12    Y: 3
```

Vidi se da je privremeni objekt stvoren odmah na mjestu predviðenom za vektor `a`. Time je izbjegnuto nepotrebno kopiranje. U ovom sluèaju, iako se konstruktor kopije ne poziva, potrebno je da konstruktor kopije bude dostupan (to jest da ima javni pristup).

8.13.3. Privremeni objekti kod vraæanja vrijednosti

Èesto je neophodno vratiti objekt kao rezultat funkcije. Dobar kandidat za tako nešto je funkcija `ZbrojiVektore()`. Prepravimo funkciju tako da uzima za parametre dva operanda te vraæa zbroj kao rezultat. Pri tome joj je potrebno dodijeliti prava pristupa privatnim èlanovima klase `Vektor`.

```
Vektor ZbrojiVektore(Vektor &a, Vektor &b) {
    cout << "Ušao u ZbrojiVektore" << endl;
    Vektor rez(a.ax + b.ax, a.ay + b.ay);
    return rez;
}
```

Kada se vraæa vrijednost iz funkcije, pomoæu konstruktora kopije stvara se privremeni objekt koji se inicijalizira objektom navedenim u naredbi `return`. Stoga konstruktor kopije mora biti dostupan. Vraæeni objekt se automatski uništava nakon što se izraèuna izraz koji je pozvao funkciju. Prilikom pozivanja funkcije sada možemo rezultirajuæu vrijednost funkcije pridružiti nekom objektu klase `Vektor`, kao u sljedeæem primjeru:

```
int main() {
    cout << "Ušao u main" << endl;
    Vektor a(12.0, 3.0), b(-2.0, -6.0), c;
    cout << "Ulazim u ZbrojiVektore" << endl;
    c = ZbrojiVektore(a, b);
    cout << "Završavam" << endl;
```

```
    return 0;
}
```

Ako prevedemo i izvedemo gornji primjer, dobit æemo sljedeæi ispis:

```
Ušao u main
Stvoren vektor pod brojem 1
X: 12    Y: 3
Stvoren vektor pod brojem 2
X: -2    Y: -6
Stvoren vektor pod brojem 3
X: 0    Y: 0
Ulazim u ZbrojiVektore
Ušao u ZbrojiVektore
Stvoren vektor pod brojem 4
X: 10    Y: -3
Stvoren vektor pomoću konstruktora kopije pod brojem 5
X: 10    Y: -3
Uništen vektor pod brojem 4
X: 10    Y: -3
Uništen vektor pod brojem 5
X: 10    Y: -3
Završavam
Uništen vektor pod brojem 5
X: 10    Y: -3
Uništen vektor pod brojem 2
X: -2    Y: -6
Uništen vektor pod brojem 1
X: 12    Y: 3
```

Na poèetku se stvaraju tri vektora `a`, `b` i `c`, a zatim se poziva funkcija `ZbrojiVektore()`. U njoj se stvara lokalni objekt pod brojem 4 koji privremeno nosi rezultat. Kada prevoditelj naiðe na naredbu `return`, pomoæu konstruktora kopije stvori objekt pod brojem 5 koji vraæa vrijednost, a lokalni objekt pod brojem 4 se uništava. Vrijednost rezultata se preslikava u objekt `c`. Buduæi da operacija dodjele nije drukèije definirana, dodjela se obavlja tako da se vrijednost svakog èlana privremenog objekta dodijeli svakom èlanu objekta `c`. Tako i objekt `c` ima broj 5. Zatim se uništava privremeni objekt pod brojem 5. Završetkom programa se brišu lokalni objekti `a`, `b` i `c`. Iako se možda na prvi pogled èini nelogiènim da se objekt 5 briše dva puta, treba se sjetiti da je zbog toga što operacija dodjele nije posebno definirana, objektu `c` dodijeljen takoðer broj 5. Prvi put se briše privremeni objekt, dok se drugi put briše lokalni objekt `c`.

Ovakav redoslijed kopiranja je podložan optimizaciji. Umjesto da se stvara lokalni objekt `rez` koji se zatim smješta u rezultat funkcije, moguće je stvoriti privremeni objekt koji čuva rezultat te se nakon završetka funkcije jednostavno proglašava rezultatom. Prepravimo funkciju `ZbrojiVektore()` i u tom smislu:

```
Vektor ZbrojiVektore(Vektor &a, Vektor &b) {
    return Vektor(a.ax + b.ax, a.ay + b.ay);
}
```

Sada se ne stvara lokalni objekt koji se zatim kopira u rezultat, nego se stvara privremeni objekt koji se nakon završetka proglašava rezultatom funkcije. Nakon izvoðenja program daje sljedeæe rezultate:

```
Ušao u main
Stvoren vektor pod brojem 1
X: 12    Y: 3
Stvoren vektor pod brojem 2
X: -2    Y: -6
Stvoren vektor pod brojem 3
X: 0    Y: 0
Ulazim u ZbrojiVektore
Ušao u ZbrojiVektore
Stvoren vektor pod brojem 4
X: 10    Y: -3
Uništen vektor pod brojem 4
X: 10    Y: -3
Završavam
Uništen vektor pod brojem 4
X: 10    Y: -3
Uništen vektor pod brojem 2
X: -2    Y: -6
Uništen vektor pod brojem 1
X: 12    Y: 3
```

Funkcija takoðer može vratiti referencu na objekt te se tada objekt ne kopira. Tipièan primjer takve funkcije je funkcijski èlan koji vraæa referencu na neki od èlanova objekta. Klasa `Tablica` koju smo uveli u poglavlju 8.4.1 ima èlanove za ubacivanje elemenata u tablicu te njeno poveæanje i smanjenje. No nigdje nismo naveli funkcijski èlan kojim se može pristupiti pojedinom elementu tablice. Stoga æemo dodati èlan `Element` koji æe kao parametar imati redni broj elementa tablice koji želimo dohvatiti te æe vratiti referencu na željeni èlan. Evo kako to izgleda:

```
class Tablica {
private:
    int *Elementi;
    int BrojElem, Duljina;
public:
    Tablica();
    Tablica(int BrElem);
    void PovecajNa(int NovaDulj);
    void DodajElem(int Elt);
    void BrisiElem(int Poz);
    int &Element(int indeks);
};
```

```
// definicije konstruktora i funkcijskih članova
// se ne mijenjaju

int &Tablica::Element(int indeks) {
    return Elementi[indeks];
}
```

Pažljiv èitatelj sigurno se pita zašto se vraæa referenca na element umjesto da se vraæa sam element. Razlog je u tome što se prilikom vraæanja elementa stvara privremeni objekt koji živi do kraja izraza u kojem je stvoren te nema nikakve veze s elementom tablice. U sluèaju vraæanja reference nema stvaranja privremenog objekta. Referenca nije ništa drugo nego adresa koja pokazuje gdje se u memoriji nalazi objekt i može se naæi s lijeve strane operatora pridruživanja. Moguæe je napisati sljedeæe:

```
int main() {
    Tablica t;
    // inicijaliziraj tablicu na 5 elemenata
    for (int i = 0; i < 5; i++) t.DodajElem(i * 8);
    cout << t.Element(3);   // pristup vrijednosti trećeg
                            // elementa
    t.Element(2) = 45;      // izmjena vrijednosti
                            // drugog elementa
    return 0;
}
```

Nije ispravno vratiti referencu na lokalni objekt zato jer se lokalni objekt uništava nakon što funkcija završi. Vraæena referenca æe uvijek pokazivati na podruèje u memoriji koje ne sadržava objekt, na što æe mnogi prevoditelji upozoriti prilikom prevođenja.

9. Strukture i unije

Da sam bio prisutan prilikom Stvaranja,
dao bih nekoliko korisnih savjeta
za bolji ustroj svemira.

Alfonso X, "Mudri" (1221–1284)

Strukture su tipovi podataka naslijeđeni iz jezika C i obogaæeni objektno orijentiranim dodacima. U C++ varijanti strukture se u svojoj biti ne razlikuju znaèajno od klasa. U sljedeæem poglavlju bit æe objašnjene razlike između struktura i klasa s posebnim osvrtom na razlike u deklaracijama u odnosu na C jezik.

Također, bit će obrađeni tipovi podataka koji omogućavaju efikasnije korištenje memorije: unije i polja bitova. Unije su posebni tipovi podataka koji omogućavaju smještaj više različitih tipova podataka na isto mjesto u memoriji. Polja bitova omogućavaju jednostavan pristup pojedinim bitovima nekog podatka umjesto korištenja složenih bitovnih podataka.

9.1. Struktura ili klasa?

U svojoj C++ varijanti *struktura* (engl. *structure*) je tip podataka koji ima ista svojstva kao klasa. Naèin deklaracije je identièan s tom razlikom što se koristi kljuèna rijeè `struct`. Struktura može sadržavati podatkovne i funkcijske èlanove, konstruktore, destruktore i kljuène rijeèi za dodjelu prava pristupa. Mala, no bitna razlika u odnosu na klase, je u tome što ukoliko se ne navede pravo pristupa elementi imaju podrazumijevan javni pristup (za razliku od klasa koje imaju podrazumijevan privatni pristup). Evo primjera strukture koja èuva podatke zaposlenog radnika u poduzeæu:

```
struct Zaposleni {

    char *ime;
    int brojGodina;
    char spol;
    char *odjel;
    int brojUzdrzavaneNejaci;

    unsigned short IzracunajPlacu();

};
```

Jezik C++ uvodi novinu u odnosu na C prilikom deklaracije objekata èiji je tip definiran strukturom. U C jeziku prilikom deklaracije strukturnih varijabli potrebno je bilo navesti

kljuènu rijeè `struct` iza koje se navodio naziv strukture te naziv varijable, kao na primjer:

```
struct Par {
    float a, b;
};

struct Par parBrojeva;       // deklaracija u C stilu
```

U C++ jeziku prilikom deklaracije varijabli nije potrebno navoditi poèetnu rijeè `struct`, nego se jednostavno samo navede naziv strukture iza kojeg se navedu nazivi objekata koje se želi deklarirati, baš kao da se radi o klasama:

```
Par parBrojeva;              // deklaracija u C++ stilu
```

Strukture se od klasa razlikuju i po tome što se, u sluèaju da se ne navede tip nasljeðivanja, podrazumijeva javno nasljeðivanje (kod klasa se podrazumijeva privatno nasljeðivanje). Inaèe strukture imaju svojstva identièna klasama: mogu sadržavati statièke èlanove, reference, druge objekte i sl.

Strukture su vrste klasa kod kojih se podrazumijeva javni pristup èlanovima, te javno nasljeðivanje, ukoliko drukèije nije naznaèeno.

Razlika između struktura i klasa je više filozofska, ali je mnogi programeri poštuju. Naime, klasa definira objekt te oznaèava da novi tip poštuje koncepte objektnog programiranja. To znaèi da æe klasa osim podatkovnih, definirati i funkcijske èlanove koji opisuju operacije na objektu. Struktura, pak, predstavlja jednostavno složeni tip poznat još iz C jezika. To je samo nakupina podataka umotanih u zajednièku ovojnicu. Ova razlika nije definirana jezikom te je na vama da odluèite želite li se toga pridržavati.

Strukture su u C++ jeziku ostavljene primarno radi kompatibilnosti s jezikom C. Kako pojam klase sam po sebi označava podatkovni skup o kojem je potrebno razmišljati na drukčiji način nego o strukturama, uvedena je nova ključna riječ `class`. Njome se ističe da navedeni tip ima dodatna svojstva u odnosu na obične strukture. Ključna riječ `struct` je ostavljena zato da se olakša prijelaz na C++ dotadašnjim C korisnicima, te da se omogući prevođenje postojećih programa bez većih izmjena.

9.2. Unije

Zamislimo da želimo napraviti program koji simulira kalkulator. Radi jednostavnosti pretpostavimo da korisnik najprije unosi prvi operand koji je broj, zatim operator, zatim drugi operand te na kraju znak jednakosti. Pri tome želimo unos korisnika prikazati jednim objektom koji æe se dalje obraðivati u programu za izraèunavanje rezultata. Vidimo da nam se unos sastoji od dva suštinski razlièita tipa podataka: brojeva, u sluèaju da je korisnik unio broj, te niza znakova, u sluèaju da je korisnik unio operator

(uzimamo niz znakova jer je korisnik mogao unijeti primjerice 'sqrt' kao operator za kvadratni korijen). Unos korisnika bismo mogli opisati sljedeæom klasom:

```
enum VrstaUnosa { unosOperator, unosBroj };
class Unos {
public:
    VrstaUnosa tip;
    int broj;
    char *nizOperator;
};
```

Klasa sadrži podatkovni èlan `tip` koji svojom vrijednošæu odreðuje da li objekt predstavlja broj ili operator. Èlanovi `broj` i `nizOperator` sadrže podatke o broju odnosno operatoru koji je unesen.

Važno je primijetiti da nikada nije potrebno pamtiti podatke i o broju i o operatoru: prisutan je samo jedan ili drugi podatak. Upravo zbog toga gornje rješenje nije efikasno sa stajališta zauzeća memorijskog prostora. Svaki objekt klase sadrži oba člana, bez obzira na to što je u jednom trenutku potreban samo jedan od njih. Bilo bi vrlo pogodno kada bi članovi `broj` i `nizOperator` mogli dijeliti memorijski prostor pa bi u svakom objektu postojao samo jedan ili samo drugi član, ovisno o potrebi.

Upravo to je moguće postići upotrebom posebnih tipova podataka nazvanih *unijama* (engl. *union*). Unije se deklariraju slično klasama i strukturama s tom razlikom što se koristi ključna riječ `union`. One mogu sadržavati podatkovne i funkcijske članove, ugniježđene klase i riječi za dodjelu prava pristupa. Bitna razlika u odnosu na klase je u tome što svi podatkovni članovi unije dijele isti memorijski prostor. To znači da se prilikom pridruživanja vrijednosti jednom podatkovnom članu prepisuje vrijednost podatkovnog člana koji je bio ranije upisan. Objekt `Unos` napisan pomoću unija izgledao bi ovako:

```
union Unos {
    int broj;
    char *nizOperator;
};
```

`Unos` može sadržavati ili `broj` ili `nizOperator`, ali nikako oba odjednom. Moguæe je definirati objekte tipa `Unos`. To se radi na isti naèin kao i prilikom definiranja objekata klase, tako da se iza naziva unije navedu identifikatori objekata:

```
Unos saTipkovnice;
```

Unije su postojale i u C jeziku gdje se prije definiranja unije ispred naziva morala stavljati kljuèna rijeè `union`:

```
union Unos saTipkovnice;        // deklaracija à la jezik C
```

U C++ jeziku to više nije potrebno (no ako se koristi nije pogrešno).

Kako svi članovi dijele isti memorijski prostor, veličina unije je jednaka veličini njenog najvećeg člana. Budući da se različiti tipovi podataka u memoriju pohranjuju prema različitim bitovnim predlošcima, vrlo je važno biti oprezan prilikom korištenja unija kako bi se uvijek pristupalo podatkovnom članu čiju vrijednost unija u tom trenutku sadržava. Ako pridružimo neku vrijednost članu `broj` unije `Unos`, nije sintaktički pogrešno čitati vrijednost člana `nizOperator`. No program koji to radi zasigurno neće funkcionirati ispravno. Vrijednost koja će se u takvom slučaju pročitati iz člana `nizOperator` je u principu slučajna i ovisi o načinu na koji računalo pohranjuje cjelobrojni i pokazivački tip. Na primjer, kôd

```
#include <iostream.h>

int main() {
    union Unos saTipkovnice;
    saTipkovnice.nizOperator = "+";
    cout << saTipkovnice.broj << endl;
    return 0;
}
```

æe ispisati broj èija æe vrijednost ovisiti o mjestu na kojem se nalazi znakovni niz `"+"`. Kako se i cijeli broj i pokazivaè na znak pohranjuju u isti memorijski prostor, prilikom èitanja èlana `broj` raèunalo æe jednostavno pristupiti memoriji i proèitati vrijednost koja je tamo zapisana. Ta vrijednost ovisi o adresi znakovnog niza te o naèinu njene pohrane.

Unija može sadržavati objekte klasa pod uvjetom da oni nemaju definiran niti konstruktor niti destruktor. Sama unija, naprotiv, može sadržavati i više konstruktora i destruktor. Statički članovi unija nisu dozvoljeni. Pojedinim članovima se može dodijeliti proizvoljno pravo pristupa. Ako se pravo pristupa ne navede, podrazumijeva se javni pristup. Unije mogu sadržavati funkcijske članove. Prilikom pisanja funkcijskih članova također valja imati na umu da unija odjednom sadrži samo jedan objekt.

Kako bi se vodila evidencija o tome koji je podatak upisan u uniju, često se unija navodi kao podatkovni član neke klase. Jedan član klase obično prati koji je podatkovni član unije trenutno aktivan. Taj član se zove *diskriminanta unije* (engl. *union discriminant*). Ako se prilikom deklaracije unije odmah deklarira varijabla ili podatkovni član klase tipa te unije, a unija se više nigdje ne koristi, njeno se ime može izostaviti. Demonstrirajmo to primjerom:

```
class Unos {
    enum { unosOperator, unosBroj} tip;
    union {             // unija nema ime
        int broj;
        char *nizOperator;
    } vrijednost;
};
```

Klasa `Unos` se sada sastoji iz podatkovnog èlana `tip` koji odreðuje vrstu unosa te iz èlana `vrijednost` koji je definiran kao *bezimena unija* (engl. *nameless union*) i koji odreðuje vrijednost unosa. Sada je moguæe napisati funkcijski èlan za ispis vrijednosti s tipkovnice na sljedeæi naèin:

```
class Unos {
    // Ovdje treba umetnuti gornju definiciju
    void Ispis();
};

void Unos::Ispis() {
    switch (tip) {
        case unosOperator:
            cout << vrijednost.nizOperator << endl;
            break;
        case unosBroj:
            cout << vrijednost.broj << endl;
            break;
    }
}
```

U gornjoj definiciji uveden je podatkovni èlan `vrijednost` koji sadrži vrijednost unosa te ga je potrebno navoditi svaki put prilikom pristupa èlanovima `broj` i `nizOpertator`. To je dosta nepraktièno i može se izbjeæi korištenjem *anonimnih unija* (engl. *anonymous unions*). Anonimna unija nema niti naziv unije niti naziv varijable koja se deklarira. Elementi unije tada pripadaju podruèju u kojem je unija definirana te im se pristupa direktno:

```
class Unos {
    enum { unosOperator, unosBroj} tip;
    union {                 // anonimna unija
        int broj;
        char *nizOperator;
    };
    void Ispis();
};

void Unos::Ispis() {
    switch (tip) {
        case unosOperator:
            cout << nizOperator << endl;
            break;
        case unosBroj:
            cout << broj << endl;
            break;
    }
}
```

U ovoj završnoj verziji klase `Unos` èlanovi `broj` i `nizOperator` pripadaju podruèju klase `Unos` te njihov naziv mora biti jedinstven unutar klase.

Zadatak. *Napišite univerzalni tip koji će moći sadržavati cijeli broj, realni broj ili pokazivač na znakovni niz. Opći oblik je ovakav:*

```
class MultiPraktik {
    char tip;
    union {
        int cj_vrij;
        double d_vrij;
        char *zn_vrij;
    };
};
```

Definirajte funkcijske članove `CjVrij()`, `DVrij()` i `ZnVrij()` za pristup vrijednostima. Članovi moraju ispisati poruku o pogreški ako se pokuša pristupiti krivoj vrijednosti.

9.3. Polja bitova

Ako se u klasi pamti više brojèanih podataka od kojih se svaki može smjestiti unutar nekoliko bitova, neracionalno je koristiti posebni cjelobrojni èlan za svaki od njih. Pogodnije je upakirati više podatkovnih èlanova u memorijski prostor kojeg zauzima jedna cjelobrojna varijabla i time uštedjeti na memorijskom prostoru. Nadalje, èesto je prilikom pisanja programa koji direktno kontroliraju sklopove raèunala potrebno omoguæiti kontrolu pojedinih bitova u jednoj cjelobrojnoj rijeèi. Slièan problem bio je razraðen u odsjeèku 2.4.11. Sa stanovišta programera elegantnije rješenje pružaju *polja bitova* (engl. *bit-fields*).

Polja bitova su posebni podatkovni članovi klase koji su svi upakirani u što manji memorijski prostor. Jedno polje se definira tako da se iza naziva cjelobrojnog člana doda znak `:` (dvotočka) iza kojeg se konstantnim izrazom navede broj bitova koliko ih član zauzima. Uzastopno navedena polja bitova će biti upakirana u jednu cjelobrojnu varijablu. Na primjer:

```
class Prekid {
public:
    unsigned short dozvoljen : 1;
    unsigned short prioritet : 3;
    unsigned short maska : 2;
};
```

Klasa `Prekid` ima tri polja bitova te æe biti upakirana u memorijski prostor koji zauzima tip `unsigned short`. Svakom od èlanova moguæe je nezavisno pristupati, a prevoditelju se prepušta da se brine o tome da modificira samo odreðene bitove nekog podatka. Pristup poljima bitova obavlja se standardnom C++ sintaksom:

```
Prekid int1;
// ...
int1.dozvoljen = 0;
if (int1.prioritet == 2) int1.maska = 1;
```

Prilikom pridruživanja je potrebno voditi raèuna o opsegu vrijednosti koje pojedini èlan može pohraniti. Na primjer, èlan `dozvoljen` je duljine samo jednog bita, tako da može poprimiti samo vrijednosti 1 ili 0. Èlan `prioritet` je duljine tri bita, pa može poprimiti vrijednosti od 0 do 7 ukljuèivo.

Operator za uzimanje adrese `&` se ne može primijeniti na polje bitova pa tako niti pokazivač na član na polje bitova nema smisla. Također, polje bitova ne može sadržavati statički član.

Zadatak. *Promijenite program iz odsječka 2.4.11 za definiranje parametara serijske komunikacije tako da se umjesto direktnog pristupa pojedinim bitovima cjelobrojne varijable koriste polja bitova.*

10. Preoptereæenje operatora

Napredak civilizacije se zasniva na proširenju broja važnih operacija koje možemo obaviti, a da ne razmišljamo o njima.

Alfred North Whitehead (1861-1947), "Introduction to Mathematics" (1911)

Osnovno svojstvo C++ jezika je enkapsulacija – pojam koji oznaèava objedinjavanje podataka i operacija. Operatori nisu ništa drugo nego operacije definirane za neki tip podataka, pa bi bilo vrlo praktièno definirati operatore za korisnièke tipove. Na primjer, zbrajanje objekata klase `Vektor` se najprirodnije provodi tako da se na dva vektora primjeni operator zbrajanja `+`. Jezik C++ to omoguæava, a postupak definiranja operatora za razlièite operande se zove *preoptereæenje operatora* (engl. *operator overloading*).

Osim redefiniranja već postojećih operacija, C++ jezik omogućava i definiranje korisničkih konverzija. Time klase postaju još sličnije ugrađenim tipovima podataka jer se korisnički definirane konverzije automatski primjenjuju kada se određeni objekt nađe na mjestu na kojem se očekuje neki drugi tip.

10.1. Korisnièki definirane konverzije

Kako bi korisnièki definirani tipovi bili što slièniji ugraðenim tipovima, moguæe je definirati operacije koje automatski konvertiraju objekt neke klase u objekt neke druge klase ili neki ugraðeni tip i obrnuto. Konverzija u opæem sluèaju, sa stanovišta klase, može biti dvosmjerna: može se konvertirati objekt klase u neki drugi tip i može se bilo koji drugi tip konvertirati u objekt klase.

Konverzije se najčešće primjenjuju prilikom prosljeđivanja objekata kao parametara funkcijama, slično kao kad se cjelobrojni podatak prosljeđuje kao parametar funkciji koja očekuje `float` broj. Prevoditelj će prilikom prevođenja stvoriti privremeni realni broj koji će imati vrijednost cjelobrojnog podatka te će se taj privremeni realni broj proslijediti funkciji.

Problem konverzija promatrat ćemo na primjeru klase `ZnakovniNiz`. Česti su prigovori na sistem manipulacije znakovnim nizovima C++ jezika. Operacije kao što su nadovezivanje i pridruživanje nizova su dosta komplicirane te zahtijevaju intenzivno korištenje sustava za alokaciju memorije. Zgodno je imati tip podataka kojim bi se to pojednostavnilo.

Zbog toga ćemo uvesti tip `ZnakovniNiz` koji će sadržavati pokazivač na niz znakova te niz funkcijskih članova, operatora i operatora konverzije potrebnih za manipulaciju nizom. Evo deklaracije klase:

```
#include <iostream.h>
#include <string.h>

class ZnakovniNiz {
private:
    char *pokNiz;
public:
    // implementaciju klase ćemo dodavati postupno
};
```

Nameæe se pitanje koje su sve konverzije prikladne za klasu `ZnakovniNiz`. Kako postoji veæ niz ugrađenih funkcija koje manipuliraju znakovnim nizovima, jedna od konverzija je svakako konverzija objekta klase `ZnakovniNiz` u pokazivaè na prvi znak. Drugim rijeèima, na mjestima gdje se oèekuje `char *`, ako se nađe objekt `ZnakovniNiz` potrebno je objekt zamijeniti vrijednošæu pokazivaèa `pokNiz`. Druga moguæa konverzija je obrnutog smjera, naime, konverzija pokazivaèa na znakovni niz u objekt klase `ZnakovniNiz`.

10.1.1. Konverzija konstruktorom

Konstruktor koji ima jedan parametar obavlja konverziju podatka iz tipa parametra u objekt klase. Dodajmo klasi `ZnakovniNiz` konstruktor, te destruktor koji æe osloboditi zauzetu memoriju. Kako bismo toèno pokazali kako se provodi proces konverzije, umetnut æemo u konstruktor i destruktor naredbe koje ispisuju poruke prilikom stvaranja i uništavanja objekta. Odmah æemo dodati i konstruktor kopije koji æe nam kasnije biti potreban.

Konstruktor s jednim parametrom obavlja konverziju iz tipa parametra u tip klase. Konstruktor kopije nije operator konverzije, jer bi on konvertirao tip u samoga sebe.

```
class ZnakovniNiz {
private:
    char *pokNiz;
public:
    // konstruktor koji će obaviti i konvenrziju
    // u tip ZnakovniNiz
    ZnakovniNiz(char *niz = "");
    ZnakovniNiz(const ZnakovniNiz &ref);
    ~ZnakovniNiz();
    char *DajPokazivac() { return pokNiz; }
};
```

```
ZnakovniNiz::ZnakovniNiz(char *niz) :
                pokNiz(new char[strlen(niz) + 1]) {
    strcpy(pokNiz,niz);
    cout << "Stvoren niz: " << niz << endl;
}

ZnakovniNiz::ZnakovniNiz(const ZnakovniNiz &ref) :
                pokNiz(new char[strlen(ref.pokNiz) + 1]) {
    strcpy(pokNiz, ref.pokNiz);
}

ZnakovniNiz::~ZnakovniNiz() {
    cout << "Uništen niz: " << pokNiz << endl;
    delete [] pokNiz;
}
```

Konverzije se koriste najèešæe prilikom deklaracije objekata te prilikom prosljeðivanja funkcijama:

```
void Funkcija(ZnakovniNiz niz) {
    cout << "Pozvana funkcija s parametrom: ";
    cout << niz.DajPokazivac() << endl;
}

int main() {
    ZnakovniNiz a = "Niz a";    // konverzija prilikom
                                // inicijalizacije
    Funkcija("parametar");      // konverzija prilikom
                                // prenošenja parametara
    return 0;
}
```

Nakon izvoðenja gornjeg programa dobiva se slijedeæi ispis:

```
Stvoren niz: Niz a
Stvoren niz: parametar
Pozvana funkcija s parametrom: parametar
Uništen niz: parametar
Uništen niz: Niz a
```

Prilikom deklaracije objekta `a` obavljena je konverzija niza s desne strane znaka `=` u objekt klase `ZnakovniNiz`. Slièno se dešava i prilikom prosljeðivanja parametra funkciji `Funkcija()`. Prije ulaska u funkciju stvara se privremeni objekt pomoæu konstruktora s jednim parametrom. Objekt se prosljeðuje funkciji te se obraðuje, a kada funkcija završi, objekt se uništava. Na kraju se uništava i objekt `a`.

Ako je potrebno, standardne konverzije se primjenjuju prije poziva konstruktora. Konverzija konstruktorom se primjenjuje samo ako nikakva druga konverzija nije

moguća. To znači da će se najprije pokušati standardna konverzija, a zatim će se pozvati konverzija konstruktorom.

10.1.2. Eksplicitni konstruktori

Ponekad može biti potrebno deklarirati eksplicitni konstruktor, odnosno konstruktor koji se neæe pozivati implicitno kao operator konverzije. To se može uèiniti tako da se ispred deklaracije konstruktora navede kljuèna rijeè `explicit`:

```
class Razlomak {
private:
    float brojnik, nazivnik;
public:
    // eksplicitni konstruktor
    explicit Razlomak(float raz) : brojnik(raz),
                                   nazivnik(1.0) {}
};
```

U gornjem kôdu je definiran konstruktor koji obavlja konverziju realnog broja u razlomak. No kako je definiran kao eksplicitan, on se neæe pozivati osim u sluèajevima kada to programer eksplicitno zahtijeva, na primjer, eksplicitnom dodjelom tipa ili izravnim pozivom konstruktora:

```
Razlomak r1(10.5);           // OK
Razlomak r2 = r1;            // OK
void AplusB(Razlomak raz1, Razlomak raz2);
AplusB(r1, 5.0);             // pogreška: 5.0 se prevodi u
                             // razlomak implicitnom
                             // konstrukcijom
AplusB(r1, Razlomak(5.0));  // OK: konstruktor je pozvan
                             // ekplicitno
```

Konstruktori deklarirani pomoæu kljuène rijeèi `explicit` se neæe koristiti kao operatori konverzije.

10.1.3. Operatori konverzije

Pomoæu operatora konverzije moguæe je definirati pravila pretvorbe podataka u neki drugi tip. Konverzija se obavlja pomoæu posebnog funkcijskog èlana, èija je deklaracija sljedeæeg oblika:

```
operator tip();
```

Pri tome je potrebno `tip` zamijeniti nazivom tipa u koji se pretvorba obavlja. Operator za konverziju ne smije imati naveden rezultirajuæi tip niti parametre.

Dodajmo klasi `ZnakovniNiz` operator koji konvertira objekt u pokazivač na prvi član:

```
class ZnakovniNiz {
// ...
public:
    operator char*() { return pokNiz; }
};
```

Sada je moguæe koristiti objekt klase `ZnakovniNiz` svugdje gdje se može pojaviti pokazivaè na znak. Na primjer:

```
#include <stdio.h>

int main() {
    ZnakovniNiz a = "abrakadabra";
    puts(a);// poziva konverziju objekta a u char *
    return 0;
}
```

Jedna klasa može definirati više operatora konverzije u razlièite tipove. Možemo dodati operator koji konvertira `ZnakovniNiz` u cijeli broj jednak duljini niza:

```
class ZnakovniNiz {
// ...
public:
    operator int() { return strlen(pokNiz); }
};
```

Kada klasa ima više operatora konverzije, vrlo èesto dolazi do situacije da prevoditelj ne može razluèiti koju konverziju treba primijeniti:

```
void Ispisi(char *pokZnak) {
    cout << "Ispisujem niz: " << pokZnak << endl;
}

void Ispisi(int broj) {
    cout << "Ispisujem broj: " << broj << endl;
}

int main() {
    ZnakovniNiz a = "U što ćeš me pretvoriti?";
    Ispisi(a);      // pogreška: koja konverzija?
    return 0;
}
```

U ovom sluèaju nije jasno treba li objekt `a` pretvoriti u cijeli broj ili u pokazivaè na znak te se zbog toga dobiva pogreška prilikom prevođenja. Programer mora eksplicitno naznaèiti koju konverziju želi pomoæu klasiène sintakse za dodjelu tipa. U okruglim zagradama se navede željeni tip, a iza zatvorene zagrade se navede objekt. Eksplicitna pretvorba u pokazivaè na znak izgleda ovako:

```
Ispisi((char *)a);
```

Moguæe je koristiti i oblik konverzije koji slièi funkcijskom pozivu:

```
Ispisi(int(a));     // isto kao i (int)a
```

Ako se ciljni tip ne može direktno dostiæi jednom korisnièki definiranom pretvorbom, prevoditelj æe pokušati pronaæi korisnièki definiranu konverziju koja æe podatak pretvoriti u međutip, koji se zatim svodi na traženi tip standardnom konverzijom. Korisnièki definirana konverzija se ne obavlja ako je na rezultat konverzije potrebno primijeniti još jednu korisnièki definiranu konverziju. Na primjer:

```
void PisiLong(long broj) {
    cout << "Long broj: " << broj << endl;
}

int main() {
    ZnakovniNiz a = "Dugi niz";
    PisiLong(a);// poziva se konverzija u int, a int
                 // se zatim standardnom konverzijom
                 // pretvara u long
    return 0;
}
```

Radi daljnje demonstracije svojstava operatora konverzije, dodat æemo još jednu klasu `Identifikator` koja može biti od koristi prilikom izgradnje prevoditeljskog programa. Klasa modelira svojstva identifikatora te sadrži objekt `naziv` klase `ZnakovniNiz` koji pamti naziv identifikatora, te cjelobrojni èlan `indeks` koji pokazuje indeks identifikatora u tablici identifikatora koju prevoditelj generira prilikom prevođenja. Klasu æemo opremiti s dva operatora konverzije, i to konverzijom u tip `ZnakovniNiz` koji æe vraæati vrijednost èlana `naziv`, te u `int` koji æe vraæati vrijednost èlana `indeks`.

```
class Identifikator {
private:
    int indeks;
    ZnakovniNiz naziv;

public:
    Identifikator(int ind, const ZnakovniNiz &ref) :
        indeks(ind), naziv(ref) { }
```

```
    const ZnakovniNiz &DajNiz() const { return naziv; }
    operator int() { return indeks; }
    operator ZnakovniNiz() { return naziv; }
};
```

Prevoditelj nikada neæe provesti korisnièki definiranu konverziju na rezultatu konverzije. Iako je moguæe `Identifikator` pretvoriti u `ZnakovniNiz`, a taj u `char *`, to se neæe provesti, te je slijedeæi poziv funkcije neispravan:

```
void PisiZnakovniNiz(char *niz) {
    cout << niz;
}

int main() {
    Identifikator a(5, ZnakovniNiz("Nema konverzije"));
    PisiZnakovniNiz(a);     // pogreška
    return 0;
}
```

Ako bismo željeli da nam ovaj kôd proradi, morali bismo umetnuti u klasu `Identifikator` operator za konverziju u `char *`:

```
class Identifikator {
public:
    // ...
    operator char*() { return naziv; }
    // implicitno se poziva konverzija
    // klase ZnakovniNiz u char*
};
```

Sada je gornji poziv ispravan. Pri tome se u naredbi `return` provodi implicitna konverzija iz tipa `ZnakovniNiz` u `char *` pomoæu korisnièki definirane konverzije iz klase `ZnakovniNiz`. Opisana konverzija u `char *` može imati neugodne popratne pojave:

```
int main() {
    ZnakovniNiz pobjednikNaLutriji = "Janko";
    char *pokNiz = pobjednikNaLutriji;
    // gornja naredba provodi implicitnu konverziju tako da
    // vraća pokazivač na implementaciju
    *pokNiz = 'R'; // pobjednik postaje Ranko
    return 0;
}
```

U ovom primjeru pretvorba tipa omoguæava pristup implementaciji objekta èime se narušava integritet objekta. To se može sprijeèiti tako da se umjesto konverzije u

`char *` uvede konverzija u `const char *`. Time se onemoguæava pristupanje pokazivanom znaku.

Nesmotreno napisane definicije operatora konverzije mogu vanjskom programu omogućiti pristup implementaciji objekta – korisnik objekta može narušiti njegov integritet, te time osnovni koncepti objektnom programiranja padaju u vodu.

Problem dvosmislenosti se èesto može pojaviti u sluèaju kada dvije klase definiraju operatore konverzije između sebe. Proširimo klasu `ZnakovniNiz` konstruktorom koji konvertira objekt klase `Identifikator` u `ZnakovniNiz`:

```
class ZnakovniNiz {
    // ...
public:
    ZnakovniNiz(const Identifikator &ref);
};

ZnakovniNiz:: ZnakovniNiz(const Identifikator &ref) :
    pokNiz(new char[strlen(ref.DajNiz().pokNiz)]) {
    strcmp(pokNiz, ref.DajNiz().pokNiz);
}
```

Sljedeæi poziv je neispravan jer nije jasno da li se `Identifikator` treba pretvoriti u `ZnakovniNiz` pomoæu operatora klase `Identifikator` ili pomoæu konstruktora klase `ZnakovniNiz`:

```
void NekaFunkcija(ZnakovniNiz);

int main() {
    Identifikator a(5, ZnakovniNiz("znn"));
    NekaFunkcija(a);
    return 0;
}
```

Programer mora eksplicitno naznaèiti željenu konverziju eksplicitnom pretvorbom tipa tako da se operator pretvorbe pozove sintaksom za poziv funkcijskih èlanova:

```
NekaFunkcija(a.operator ZnakovniNiz());
```

Zadatak. *Dodajte klasi `ZnakonvniNiz` konstruktor koji će kao parametar uzimati cijeli broj. Pri tome će se znakovni niz inicijalizirati tako da sadrži zadani broj. Na primjer, deklaracija*

```
ZnakovniNiz godina(1997);
```

će inicijalizirati objekt `godina` *na vrijednost* `"1997"`.

Zadatak. *Promijenite operator konverzije u cijeli broj klase* `ZnakovniNiz` *tako da rezultat bude broj sadržan u nizu, ili 0 ako niz ne sadrži broj. Na primjer, niz* `"-35"` *se mora pretvoriti u cijeli broj* `-35`*, dok niz* `"1a4d"` *ne sadrži ispravan broj te se mora pretvoriti u broj 0.*

Zadatak. *Zadane su klase* `Tocka`*,* `Poligon` *i* `Pravokutnik` *koje opisuju geometrijske likove u ravnini:*

```
class Tocka {
public:
    int x, y;
};

class Pravokutnik {
private:
    Tocka gornjaDesna, donjaLijeva;
};

class Poligon {
private:
    int brojVrhova;
    Tocka *vrhovi;
};
```

`Tocka` *je opisana pomoću svojih x i y koordinata.* `Pravokutnik` *opisuju dvije točke: gornja desna i donja lijeva.* `Poligon` *se sastoji od člana koji pokazuje broj vrhova poligona, te od pokazivača na niz točaka koje daju koordinate vrhova. Potrebno je realizirati pretvorbu iz klase* `Pravokutnik` *u klasu* `Poligon`*. Pri tome konverziju riješite na dva načina: jednom kao funkcijski član klase* `Pravokutnik`*, a drugi put kao član klase* `Poligon`*.*

Zadatak. *Konstruktor klase* `Razlomak` *promijenite tako da realni broj svede na najbliži razlomak (pri tome razlomak mora imati cijeli brojnik i prirodni nazivnik).*

10.2. Osnove preoptereæenja operatora

Ne mogu se svi operatori C++ jezika preopteretiti. Pet operatora navedenih u tablici 10.1, se ne mogu preopteretiti, dok su dozvoljeni operatori navedeni u tablici 10.2.

Tablica 10.1. Operatori koji se ne mogu preopteretiti

`.`	`.*`	`::`	`?:`	`sizeof`

Također, nije moguæe uvesti novi operator. Na primjer, jezik FORTRAN posjeduje operator `**` za potenciranje koji bi vrlo èesto bio koristan i u C++ programima. No

nažalost, nije moguæe dodati novi operatorski simbol, veæ samo proširiti znaèenje veæ postojeæih simbola.

Svaki operator zadržava broj argumenata koji ima po standardnoj C++ sintaksi. To znači da zato što je operator `<<` definiran kao binarni, mora biti i preopterećen kao binarni. Redoslijed obavljanja računskih operacija također ostaje nepromijenjen.

Tablica 10.2. Operatori koji se mogu preopteretiti

<table>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>^</td><td>&</td><td>|</td></tr>
<tr><td>~</td><td>!</td><td>,</td><td>=</td><td><</td><td>></td><td><=</td><td>>=</td></tr>
<tr><td>++</td><td>--</td><td><<</td><td>>></td><td>==</td><td>!=</td><td>&&</td><td>||</td></tr>
<tr><td>+=</td><td>-=</td><td>/=</td><td>%=</td><td>^=</td><td>&=</td><td>|=</td><td>*=</td></tr>
<tr><td><<=</td><td>>>=</td><td>[]</td><td>()</td><td>-></td><td>->*</td><td>new</td><td>delete</td></tr>
</table>

Operator se definira kao funkcija ili *funkcijski* èlan koji za naziv ima kljuènu rijeè `operator` iza koje se stavlja simbol operatora. Operator se može, a i ne mora odvojiti razmakom od kljuène rijeèi `operator` i liste parametara. Operatori

```
+        -        &        *
```

postoje u unarnoj i binarnoj varijanti (dakle, mogu uzimati jedan ili dva parametra), pa se mogu i preopteretiti kao unarni ili binarni.

Kako ima smisla zbrajati vektore, možemo u klasu `Vektor` ubaciti operator za zbrajanje:

```
class Vektor {
    friend Vektor operator+(Vektor &a, Vektor &b);
private:
    float ax, ay;
public:
    Vektor(float x = 0, float y = 0) { ax = x; ay = y; }
    void PostaviXY(float x, float y) { ax = x; ay = y; }
    float DajX() const { return ax; }
    float DajY() const { return ay; }
    void MnoziSkalarom(float skalar);
};

inline
void Vektor::MnoziSkalarom(float skalar) {
    ax *= skalar;
    ay *= skalar;
}

Vektor operator+(Vektor &a, Vektor &b) {
    return Vektor(a.ax + b.ax, a.ay + b.ay);
}
```

Definirani operator + se može koristiti na slijedeæi naèin:

```
#include <iostream.h>

int main() {
    Vektor a(12.0, 3.0), b(-3.0, -6.0), c;
    c = a + b;  // poziva se operator za zbrajanje vektora
    cout << c.DajX() << "    " << c.DajY() << endl;
    return 0;
}
```

Ovako definirani operator za zbrajanje se ni po èemu ne razlikuje od operatora za zbrajanje brojèanih tipova, primjerice cijelih brojeva. Izraz

```
c = a + b;
```

se interpretira kao

```
c = operator+(a, b);
```

Ovakav direktan poziv preoptereæenog operatora je posve legalan, iako je daleko praktiènije koristiti operatore unutar izraza (zbog toga je uopæe preopterećenje operatora i uvedeno).

Prilikom preopterećenja operatora ograničeni smo na operatore definirane za naše klase – nije moguće preopteretiti operatore za ugrađene tipove.

Time su zlobni i tašti programeri uskraæeni za zadovoljstvo pisanja potpuno neèitljivog koda u kojem bi operator + za cijele brojeve imao znaèenje dijeljenja.

10.3. Definicija operatorske funkcije

Svaki preoptereæeni operator je definiran svojom operatorskom funkcijom. Ona potpuno poštuje sintaksu obiènih funkcija te ima svoj naziv, argumente i povratnu vrijednost. Operatorske funkcije mogu biti preoptereæene, pod uvjetom da se mogu razlikovati po svom potpisu.

Kako je definiranje operatora dozvoljeno samo za korisnički definirane klase, barem jedan od argumenata operatorske funkcije uvijek mora biti objekt, pokazivač na objekt ili referenca na objekt neke klase. Broj argumenata operatora se ne može mijenjati pa niti jedan od parametara ne može imati podrazumijevanu vrijednost.

Operatorska funkcija može biti definirana kao funkcijski član klase, a može biti definirana i izvan klase kao obična funkcija. Za neke operatore nemamo mogućnost izbora – operatorske funkcije za operatore

```
=         []        ()        ->
```

mogu iskljuèivo biti definirane kao funkcijski èlanovi. Ako se operatorska funkcija definira kao funkcijski èlan, broj parametara je za jedan manji nego u sluèaju kada je funkcija definirana izvan klase.

Definirajmo klasu `X` te operatore `+` i `-`. Binarni operatori čije funkcije su definirane kao članovi imaju jedan parametar, dok unarni nemaju parametara. Također ćemo definirati objekte `a` i `b` klase `X`:

```
class X {
public:
    X operator-(const X &desno);
};

X operator+(const X &lijevo, const X &desno);

X a, b;
```

Poziv

```
a + b;
```

interpretira se kao

```
operator+(a, b);
```

dok se poziv

```
a - b;
```

interpretira kao

```
a.operator-(b);
```

Dakle, ako je operator definiran izvan klase, lijevi i desni objekt se prosljeðuju operatorskoj funkciji kroz prvi odnosno drugi parametar. Kada je operatorska funkcija definirana kao funkcijski èlan, poziva se funkcijski èlan za objekt lijevo od operatora, dok se objekt desno od operatora prosljeðuje kroz parametar.

Operatori se mogu preopteretiti samo za korisnički definirane tipove. U slučaju da se operatorska funkcija definira izvan klase, barem jedan od parametara mora biti klasa.

Kako operatori `+`, `-`, `*` i `&` imaju svoju unarnu i binarnu varijantu, obje varijante mogu biti preoptereæene neovisno, kao u sljedeæem primjeru:

```
class Y {
public:
    // definicija unutar klase
    void operator*();            // unarni *
    Y operator*(const Y &ref);   // binarni *
};

// definicija izvan klase
void operator&(const Y &ref);                // unarni &
Y operator&(const Y &lijevo, const Y &desno); // binarni &
```

Pojedine varijante operatora koriste se na slijedeæi naèin:

```
Y a, b;

int main() {
    *a;     // a.operator*();
    a * b;  // a.operator*(b);
    &a;     // operator&(a);
    a & b;  // operator&(a, b);
    return 0;
}
```

Iako se èini da je svejedno je li operatorska funkcija definirana unutar ili izvan klase, postoji suštinska razlika u jednom i drugom pristupu.

Ako je operator definiran kao funkcijski èlan, na lijevi argument se ne primjenjuju korisnièki definirane konverzije.

Na primjer:

```
class Kompleksni {
friend Kompleksni operator-(const Kompleksni &l,
                            const Kompleksni &d);
private:
    float real, imag;
public:
    Kompleksni(float r = 0, float i = 0) :
            real(r), imag(i) {}
    Kompleksni operator+(const Kompleksni &d);
};

Kompleksni Kompleksni::operator+(const Kompleksni &d) {
    return Kompleksni(real + d.real, imag + d.imag);
}

Kompleksni operator-(const Kompleksni &l,
                     const Kompleksni &d) {
```

```
        return Kompleksni(l.real + d.real, l.imag + d.imag);
    }
```

Kako je bilo koji realni broj ujedno i kompleksni (njegov imaginarni dio je nula), klasa `Kompleksni` ima konstruktor koji može svaki realni broj konvertirati u kompleksni. Operator `+` je definiran kao funkcijski èlan, dok je operator `-` definiran izvan klase. Matematièki je sasvim ispravno zbrajati ili oduzimati realan i kompleksan broj, ali æe pokušaj zbrajanja u sljedeæem primjeru izazvati nevolje:

```
int main() {
    Kompleksni a, b;
    b = 5 + a; // pogreška: 5 se ne konvertira u Kompleksni
    b = 5 - a; // OK: 5 se konvertira u Kompleksni
    return 0;
}
```

Pokušaj zbrajanja realnog i kompleksnog broja rezultira pogreškom prilikom prevođenja jer se poziv

```
5 + a
```

interpretira kao

```
5.operator+(a)      // pogreška
```

što je oèigledna glupost, jer je objekt s lijeve strane ugrađeni tip za koji se ne mogu uopæe pozivati funkcijski èlanovi. Naprotiv, poziv

```
5 - a;
```

se interpretira kao

```
operator-(5, a);    // OK
```

Sada se na oba parametra mogu primijeniti pravila konverzije podataka, te je poziv ispravan. Simetrièni operatori se obièno definiraju izvan klase – upotrebom konverzije i definicijom izvan klase izbjegnuta je potreba da se definiraju posebno varijante

```
Kompleksni operator-(float l, const Kompleksni &d);
Kompleksni operator-(const Kompleksni &l,
                     const Kompleksni &d);
Kompleksni operator-(const Kompleksni &l, float d);
```

Nije moguæe definirati operator zbrajanja koji u svojim parametrima ne spominje niti jedan objekt klase, jer bi se time promijenilo znaèenje ugrađenog operatora.

Tako nije moguæe, na primjer, definirati novi operator za zbrajanje realnih brojeva, jer se time zapravo ne preopтereæuje operator za novi tip, nego se mijenja znaèenje ugraðenog operatora za veæ postojeæe tipove:

```
Kompleksni operator+(float l, float d); // pogreška
```

Operatori redovito moraju pristupati implementacijskim detaljima klase. Kako se simetrièni operatori najèešæe deklariraju izvan klase, oni pod normalnim uvjetima nemaju pristup privatnim i zaštiæenim èlanovima. Tada ih je kljuènom rijeèi `friend` moguæe uèiniti prijateljem klase i time im omoguæiti pristup implementaciji.

Zadatak. *Implementirajte operatore `+=`, `-=`, `*=` i `/=` za objekte klase `Kompleksni`. Operatorske funkcije neka budu deklarirane izvan klase, kako bi se konverzija mogla primjenjivati na objekte s obje strane znaka. Obratite pažnju na tipove argumenata. Uputa: poželjno je izbjeći prečesta kopiranja objekta. Također, potrebno je omogućiti izraz tipa*

```
a = b += c;
```

koji je sasvim korektan za ugrađene tipove.

Zadatak. *Omogućite uspoređivanje kompleksnih brojeva operatorom `==`. Razmislite o tome da li će se operator definirati unutar ili izvan klase.*

10.3.1. Operator =

Iako je moguæe operator pridruživanja redefinirati za sasvim egzotiène primjene koje nemaju veze s dodjelom (na primjer redefinirati ga tako da ima znaèenje provjere jednakosti), to se ne preporuèa. Naime, u tom sluèaju neæe biti moguæe dodjeljivati vrijednosti jednog objekta drugom. Pakostan programer koji želi "zaštititi" svoj izvorni kôd može i tome doskoèiti, tako da pridruživanje smjesti, primjerice, u operator `%` i time si definitivno osigurati prvu nagradu na natjeèaju za najluđeg *hackera* u Jankomiru.

Operator pridruživanja mora biti definiran kao funkcijski član. Ako klasa ne definira operator pridruživanja, prevoditelj će sam generirati podrazumijevani operator koji će primijeniti operator pridruživanja na svaki član klase.

Razlozi zbog kojih to ponekad nije prihvatljivo su isti kao i razlozi zbog kojih podrazumijevani konstruktor kopije nije ispravan. Ako klasa koristi dinamièku alokaciju memorije, potrebno je prilikom pridruživanja osloboditi staru memoriju i alocirati novu. To æemo pokazati na primjeru klase `ZnakovniNiz`. Naime, ako napišemo

```
ZnakovniNiz a, b("Nešto je trulo u našem C++ kôdu.");
a = b;
```

podrazumijevani operator = æe jednostavno èlanu pokNiz objekta a dodijeliti vrijednost èlana pokNiz objekta b, te æe zapravo oba objekta pokazivati na isti komad memorije. Zbog toga æemo definirati vlastiti operator pridruživanja:

```
class ZnakovniNiz {
    // ...
public:
    ZnakovniNiz& operator=(const ZnakovniNiz &desno);
};

ZnakovniNiz& ZnakovniNiz::operator=(const ZnakovniNiz
                                    &desno) {
    if (&desno != this) {
        delete [] pokNiz;
        pokNiz = new char[strlen(desno.pokNiz) + 1];
        strcpy(pokNiz, desno.pokNiz);
    }
    return *this;
}
```

U gornjem kôdu se prije dodjele oslobaða memorija na koju pokazuje pokNiz prije pridruživanja te se alocira novi memorijski prostor za smještaj niza s desne strane. Važno je primijetiti da je potpuno ispravno pridodijeliti objekt samome sebi. Zato se prvo ispituje je li adresa desnog operanda razlièita od adrese objekta kojemu se vrijednost pridružuje. Kada tog uvjeta ne bi bilo, naredbom

```
a = a;
```

bi se memorija na koju pokazuje pokNiz oslobodila prije dodjele, èime bi se uništio i sadržaj desnog operanda jer je to isti objekt. Operator pridruživanja može imati proizvoljnu povratnu vrijednost, no najèešæe se vraæa referenca na objekt kojem se vrijednost dodjeljuje. Time se omoguæava uzastopno dodjeljivanje. Naredba

```
ZnakovniNiz a, b;
b = a = "copycat";
```

se interpretira kao

```
b.operator=(a.operator=(ZnakovniNiz("copycat")));
```

te se vrijednost dodjeljuje i objektu a i objektu b. Time se postiže da korisnièki definiran operator pridruživanja ima isto ponašanje kao i ugraðeni operator.

Promotrimo još način na koji su deklarirani parametri u operatorskim funkcijama. Mnogi operatori kao operande uzimaju reference na objekte, čime se postiže brži rad programa jer nema nepotrebnog kopiranja pomoću konstruktora kopije. Parametri se definiraju kao konstantni, što omogućava pozivanje operatora i za objekte deklarirane konstantnima. Povratna vrijednost se kod operatora pridruživanja i operatora

obnavljajućeg pridruživanja (engl. *update assignment*) najčešće definira kao referenca na objekt kojem se vrijednost pridružuje. Kod operatora kao što su zbrajanje i oduzimanje povratna vrijednost se definira kao objekt, a ne referenca na objekt, jer operator izračunava rezultat koji se smješta u privremeni objekt te se uništava na kraju izraza u kojem je nastao.

10.3.2. Operator `[]`

Operator `[]` prvenstveno se koristi za dohvaæanje èlanova polja te se zove *operator za indeksiranje* (engl. *indexing operator*). On mora uvijek biti definiran kao funkcijski èlan klase. Izraz

```
x[y];
```

se interpretira kao

```
x.operator[](y);
```

Klasi `ZnakovniNiz` æemo dodati operator `[]` za pristup proizvoljnom znaku niza. Operator æe imati cjelobrojni parametar `n` te æe vratiti referencu na `n`-ti znak niza. Vraæanje reference omoguæava da se operator može navesti i s lijeve i s desne strane operatora za pridruživanje.

```
class ZnakovniNiz {
    // ...
public:
    char& operator[](int n) { return pokNiz[n]; }
    // ...
};
```

Ovim je naèinom omoguæen pristup pojedinim znakovima znakovnog niza na isti naèin kao i u sluèaju obiènih znakovnih nizova (`char *`), na primjer

```
ZnakovniNiz a("Ivica i Marica");
cout << a[0] << a[6] << a[8];
```

Takav pristup èlanovima znakovnog niza æe posebno cijeniti prelaznici s jezika PASCAL, s obzirom da je upravo takva sintaksa ugrađena u PASCAL.

Operator `[]` se može preopteretiti sa samo jednim parametrom – nije moguće definirati indeksiranje po dvije dimenzije.

Jedan od velikih nedostataka matrica ugrađenih u C++ jezik je to što dimenzije matrice moraju biti konstante poznate prilikom prevođenja. Program mora alocirati memorijski prostor za najveæu oèekivanu matricu, što rezultira neefikasnim korištenjem memorije.

Zbog toga se kao rješenje nameæe razviti klasu `Matrica` kojom bi se taj nedostatak izbjegao. Takva klasa mora biti indeksirana po dvije dimenzije, jer bi u suprotnom dohvaæanje pojedinih èlanova bilo vrlo neèitko. Zato æemo se morati poslužiti raznim trikovima. Osnovna ideja je da se izraz

```
obj[m][n]
```

interpretira kao

```
(obj[m])[n]
```

Prvo se operator `[]` primjeni na objekt `obj`. Operator vraæa privremeni objekt u kojem je pohranjen podatak o prvoj dimenziji. Na taj se objekt zatim primijeni operator `[]` koji sada odreðuje drugu dimenziju. Privremeni objekt zna sada i prvu i drugu dimenziju te može obaviti indeksiranje matrice. Evo rješenja problema:

```
class Matrica {
private:
    float *mat;
    int redaka, stupaca;

    class Pristupnik {
    public:
        int prvadim, stupaca;
        float *mat;
        Pristupnik(int pd, int st, float *mt) :
            prvadim(pd), stupaca(st), mat(mt) {}
        float& operator[](int drugadim) {
            return mat[prvadim * stupaca + drugadim];
        }
    };

public:
    Matrica(int red, int stu);
    ~Matrica() { delete [] mat; }
    Pristupnik operator[](int prvadim) {
        return Pristupnik(prvadim, stupaca, mat);
    }
};

Matrica::Matrica(int red, int stu) : redaka(red),
        stupaca(stu), mat(new float[red * stu]) {}
```

U klasi `Matrica` èlanovi `redaka` i `stupaca` pamte broj redaka i stupaca matrice te se oni moraju navesti u konstruktoru, èime se matrica postavlja na neku poèetnu dimenziju. Pokazivaè `mat` pokazuje na podruèje memorije velièine `redaka * stupaca`, te se u njega smještaju èlanovi matrice. Podruèje je alocirano kao jednodimenzionalan niz zato jer prilikom korištenja operatora `new []` za alokaciju niza sve dimenzije osim prve

moraju biti poznate prilikom prevoðenja. U našem sluèaju su obje dimenzije nepoznate prilikom prevoðenja, te se pozicija elementa unutar podruèja izraèunava prilikom pristupa elementu.

Klasa ima preopterećen operator `[]` koji kao rezultat vraća privremeni objekt klase `Pristupnik`. Konstruktorom se objekt inicijalizira tako da se zapamti prva dimenzija, broj stupaca i pokazivač na početak matrice. Klasa `Pristupnik` ima također operator `[]` koji vraća referencu na element matrice.

Nedostatak ovakve simulacije dvodimenzionalnog indeksiranja jest taj da se svaki put prilikom pristupa matrici stvara privremeni objekt koji se uništava na kraju izraza, što može bitno usporiti rad programa. Ponekad to može biti jedino rješenje, pogotovo ako je potrebno provjeravanje je li indeks unutar dozvoljenog raspona, no naš problem možemo pokušati riješiti i drukčije. Ideja je da operator `[]` kao svoj rezultat vrati pokazivač na područje unutar matrice gdje počinje traženi redak. Na to se primjeni ugrađeni operator `[]` koji označava pomak od pokazivača te se njime pristupi traženom elementu matrice. Sada više nije moguće provjeriti da li je drugi indeks unutar dozvoljenog područja, no indeksiranje ide brže. Klasa `Pristupnik` više nije potrebna:

```
class Matrica {
private:
    float *mat;
    int redaka, stupaca;
public:
    Matrica(int red, int stu);
    ~Matrica() { delete [] mat; }
    float *operator[](int prvadim) {
        return mat + prvadim * stupaca;
    }
};
```

Ovakvo rješenje radi brže, no moguæe ga je primijeniti samo zato jer smo na prikladan naèin realizirali implementaciju objekta (matricu smo pamtili po recima). Veliki je nedostatak u tome što operator vraæa pokazivaè u implementaciju, èime se omoguæava programeru da mijenja unutrašnjost objekta na naèin zavisan od implementacije. To ugrožava integritet objekta jer ako se pokaže da je matricu potrebno pamtiti na drukèiji naèin, program koji koristi pokazivaèe unutar objekta pada u vodu. Kao sve stvari u životu, C++ programi su kompromisi između želja i moguænosti.

10.3.3. Operator `()`

Operator za poziv funkcije `()` mora biti definiran kao funkcijski èlan klase te može imati niti jedan ili proizvoljan broj parametara, a takoðer može pomoæu `...` (tri toèke) imati neodreðeni broj parametara. Izraz

```
x( parametri );
```

se interpretira kao

```
x.operator()( parametri );
```

Primjerice, možemo operator `()` preopteretiti tako da vraæa podniz niza znakova. Operator æe imati dva parametra: cjelobrojnu varijablu `start` koja æe pokazivati od kojeg znaka je potrebno poèeti uzimati podniz (nula oznaèava prvi znak), te varijablu `duljina` koja æe pokazivati koliko je znakova potrebno uzeti. Povratna vrijednost æe biti objekt `ZnakovniNiz` koji æe sadržavati traženi podniz:

```
class ZnakovniNiz {
    // ...
public:
    ZnakovniNiz operator()(int start, int duljina);
};

ZnakovniNiz ZnakovniNiz::operator()(int start,
                                    int duljina) {
    char pomocni[200], *pok1 = pomocni;
    char *pok2 = pokNiz + start;
    while (duljina-- && *pok2) *(pok1++) = *(pok2++);
    *pok1 = 0;
    return ZnakovniNiz(pomocni);
}
```

Sada je moguæe pozvati operator na sljedeæi naèin:

```
int main() {
    ZnakovniNiz a = "Kramer protiv Kramera";
    cout << (const char *)a(2, 17) << endl;
    return 0;
}
```

Program nakon izvoðenja ispisuje

```
amer protiv Krame
```

Primijetite u gornjem primjeru korisnièki definiranu konverziju privremenog objekta koji se vraæa iz operatora `()` u tip `const char *`. Važan je modifikator `const`, jer je operator `<<` za ispis pomoæu tokova definiran tako da u sluèaju ispisa znakovnih nizova za parametar uzima `const char *`.

10.3.4. Operator ->

Ovaj operator se uvijek preoptereæuje kao unarni i mora biti definiran kao funkcijski èlan. Kao rezultat on mora vratiti ili pokazivaè na objekt, referencu na objekt ili sam objekt neke klase. Izraz

```
x->m
```

interpretira se tako da se ponajprije promotri tip objekta `x`. Ako je `x` pokazivaè na objekt neke klase, izraz ima znaèenje klasiènog pristupa èlanu `m`. Ako je `x` objekt ili referenca na objekt, izraz se interpretira kao

```
(x.operator->())->m
```

Klasa objekta `x` se pretraži za preoptereæenu verziju operatora `->`. Ako operator ne postoji, prijavljuje se pogreška prilikom prevođenja. U suprotnom se poziva operatorska funkcija za objekt `x`. Operatorska funkcija mora vratiti pokazivaè, referencu ili sam objekt. Ako se vrati pokazivaè, tada se na njega primijeni klasièni operator `->` za pristup elementima klase. Ako se vrati objekt ili referenca, ponavlja se postupak, to jest traži se operator `->` u klasi objekta koji je vraæen te se izvršava. Pojasnimo to primjerom:

```
#include <iostream.h>

class Z {
public:
    int clan;
    Z(int mm) : clan(mm) {}
};


class Y {
public:
    Z &referencaNaZ;
    int clan;
    Y(Z &rz, int mm) : referencaNaZ(rz), clan(mm) {}
    Z *operator ->() {
        cout << "Pozvan oeprator -> od Y" << endl;
        return &referencaNaZ;
    }
};

class X {
public:
    Y &referencaNaY;
    int clan;
    X(Y &ry, int mm) : referencaNaY(ry), clan(mm) {}
    Y& operator ->() {
        cout << "Pozvan operator -> od X" << endl;
        return referencaNaY;
    }
};

int main() {
    Z objZ(1);
    Y objY(objZ, 2);
    X objX(objY, 3);
    cout << "član: " << objX->clan;
```

```
    return 0;
}
```

Nakon izvođenja se dobiva slijedeæi ispis:

```
Pozvan operator -> od X
Pozvan operator -> od Y
član: 1
```

Pojasnimo kako se izvodi navedeni primjer. Definirana su tri objekta klasa `X`, `Y` i `Z`. Objekt `objX` ima referencu na objekt `objY` koji ima referencu na `objZ`. Prilikom nailaska na `objX->clan`, prevoditelj ponajprije provjerava je li `objX` pokazivaè. Kako je to objekt, traži se operator `->` u klasi `X`. Operator postoji te se izvodi; ako klasa `X` ne bi sadržavala operator, prijavila bi se poruka o pogreški. Operator ispisuje prvu poruku te vraæa referencu na objekt `objY`. Na njega se ponovo primjenjuje operator `->`, kao da piše `objY->clan`. Kako je vraæena referenca, a ne pokazivaè, postupak se ponavlja. Traži se operator `->` u klasi `Y`. On postoji te se izvodi – ispisuje se druga poruka te se vraæa pokazivaè na `Z`. Kako je sada rezultat pokazivaè, preko njega se pristupa èlanu `clan`. Ispisuje se vrijednost `1`, što nam pokazuje da se pristupa èlanu objekta `objZ`, iako i `objX` i `objY` imaju èlan naziva `clan`.

Pomoću operatora `->` se mogu izgraditi klase koje služe kao *pametni pokazivači* (engl. *smart pointers*). "Pamet" pokazivača se sastoji u tome što se prilikom pristupa objektu mogu obaviti dodatne akcije. Zamislimo da želimo izgraditi napredan sustav upravljanja memorijom koji će omogućiti korištenje virtualne memorije na disku računala. Kako je osnovna memorija računala često nedovoljnog kapaciteta, često je potrebno nekorištene podatke prebaciti na diskovni sustav te ih pozivati po potrebi. Poseban sustav za *skupljanje smeća* (engl. *garbage collection*) može u određenim vremenskim intervalima dijelove memorije koji se ne koriste jednostavno snimiti na disk i osloboditi glavnu memoriju.

Upravljanje virtualnom memorijom može biti dosta složeno. Osnovna ideja je da se za svaku klasu `XX` napravi i klasa `PokXX` pokazivača na objekte klase `XX`. Klasa `PokXX` mora sadržavati operator `->` kojim se pristupa objektu na kojeg se pokazuje, te podatke o trenutnom smještaju samog objekta (je li on trenutno u glavnoj memoriji; ako nije, onda na kojem se disku nalazi, ili čak na kojem računalu u mreži). Svaki pristup objektu se obavezno mora obaviti preko odgovarajućeg pokazivača. Operator `->` klase pokazivača će biti zadužen za to da dovede objekt u memoriju te vrati pokazivač na njega. Pristup članovima objekta na koji se pokazuje na taj će način biti potpuno neovisan o načinu smještanja objekta. Programer ne mora voditi računa o virtualnoj memoriji, jedini ustupak je što se za pokazivače moraju koristiti objekti posebne klase. U nastavku će biti dana samo skica rješenja. Rješenje koje bi u potpunosti funkcioniralo je presloženo te je ovisno o operativnom sustavu na kojem se implementira. Napominjemo da je klasa pokazivača na objekte idealan kandidat da se napiše pomoću predložaka koji će biti objašnjeni u jednom od sljedećih poglavlja.

```
class XX {
    // klasa XX je sam objekt koji ima svoje članove
    // ovisno o samom problemu koji se rješava
};

class PokXX {
private:
    int uMemoriji;  // postavljen na 1 pokazuje da
                    // je objekt trenutno u memoriji
    XX *pokXX;      // pokazivač na objekt u memoriji
                    // preostali članovi ovise o načinu
                    // na koji se objekti smještaju na disk
public:
    XX* operator->();
};

XX* PokXX::operator->() {
    if (!uMemoriji) {
        // pozovi objekt s vanjske jedinice
        uMemoriji = 1;
    }
    return pokXX;
```

Operator `->*` za pristup èlanovima preko pokazivaèa na èlanove se preoptereæuje kao klasièan binarni operator te za njega ne vrijede ovdje iznesena pravila.

```
}
```

Dakle, izraz

```
a->*b
```

se interpretira kao

```
a.operator->*(b)
```

ili, ako je operator deklariran izvan klase, kao

```
operator->*(a, b)
```

10.3.5. Prefiks i postfiks operatori ++ i --

U C++ jeziku postoje dvije verzije operatora `++` i `--`. Prefiks verzija se piše ispred operanda, dok se postfiks verzija navodi iza njega. U konaènoj varijanti standarda jezika moguæe je posebno preopteretiti obje verzije operatora.

Prilikom preopterećenja je potrebno da svaka od preopterećenih funkcija ima jedinstven potpis kako bi se mogle razlikovati. Prefiks varijanta operatora ++ izgleda ovako:

```
class Primjer {
// ...
    void operator++();
};
```

Operatorska funkcija može biti èlan klase, a može biti i definirana izvan klase te tada uzima jedan parametar. Postfiks operator ++ ima isti potpis te se prilikom prevođenja ne može razaznati da li definicija operatorske funkcije pripada prefiks ili postfiks verziji. Kako bi se ipak omoguæilo preoptereæenje i jedne i druge varijante operatora, uvedena je konvencija po kojoj postfiks verzija ima još jedan dodatni parametar tipa `int`. Evo primjera definicija prefiks i postfiks verzija operatora:

```
class Primjer {
public:
    // deklaracije unutar klase
    void operator++();      // prefiks verzija
    void operator++(int);   // postfiks verzija
};

// deklaracije izvan klase

void operator--(Primjer &obj) {
    // prefiks verzija
}

void operator--(Primjer &obj, int) {
    // postfiks verzija
}
```

Prilikom pozivanja postfiks verzije operatora cjelobrojni parametar ima neku podrazumijevanu vrijednost koju određuje prevoditelj. Operator se može i pozvati navodeæi puno ime operatorske funkcije te se tada može i eksplicitno navesti cjelobrojni parametar, kao u sljedeæem primjeru:

```
int main() {
    Primjer obj;
    obj--;                  // prevoditelj daje svoju
                            // vrijednost parametra
    obj.operator++(76);     // eksplicitni poziv postfiksne
                            // funkcije s navedenim parametrom
    operator--(obj);    // prefiks operator
    operator--(obj, 32);    // postfiks operator
    return 0;
}
```

Programer ima potpunu slobodu u odabiru operacija koje æe se obavljati u prefiks i postfiks verzijama operatora. Izbor povratne vrijednosti je takoðer dan programeru na volju. Tako možemo primjerice preopteretiti operator `--` da ispisuje odreðene podatke klase. Takav stil programiranja nije preporuèljiv jer se time samo poveæava neèitkost programa, no važno je razumjeti da programer ima odriješene ruke prilikom odreðivanja što æe njegov operator raditi.

10.3.6. Operatori `new` i `delete`

Operatori `new` i `delete` su zaduženi za alokaciju i dealokaciju memorijskog prostora. Kao i veæina drugih operatora, i oni mogu biti preoptereæeni. C++ jezik definira globalne `new` i `delete` operatore koji su zaduženi za alokaciju i dealokaciju. Svaka klasa može definirati svoje operatore koji se tada koriste za alokaciju memorije objekata samo tih klasa.

U biti postoje po dvije varijante operatora `new` i `delete`: operator `new` za alociranje jednog objekta i `new []` za alokaciju polja objekata te operator `delete` za dealokaciju jednog i `delete []` za dealokaciju polja objekata. Posebno se mogu preopteretiti varijante operatora za jedan, a posebno za polje objekata.

Kada se operatori `new` i `delete` deklariraju unutar klase, oni su automatski statički, te to nije potrebno posebno naznačivati.

Razlog za to je posve jasan: objekt još ne postoji kada se poziva operator `new` te bi pokušaj pristupa èlanovima objekta rezultirao pristupom neinicijaliziranoj memoriji. Operator `delete` se poziva nakon što je destruktor uništio element, pa bi opet pokušaj pristupa èlanovima rezultirao pristupom neincijaliziranoj memoriji.

Prilikom alokacije objekta neke klase, prvo se pokuša pronaći preopterećena verzija operatora `new` u klasi koju se želi alocirati. Ako se želi alocirati jedan objekt, traži se operator `new`, a u slučaju da se alocira niz objekata traži se operator `new []`. Ako se operator ne nađe, poziva se pripadajući globalni operator.

Operatori `new` i `new []` mogu uzimati proizvoljan broj parametara, no prvi parametar mora biti tipa `size_t` (taj tip je definiran je u biblioteci `stddef.h`). Preostali parametri se mogu navesti eksplicitno prilikom pozivanja operatora u okruglim zagradama odmah iza ključne riječi `new`. Povratna vrijednost mora biti `void *`. Evo primjera:

```
void *operator new(size_t velicina, int br, char *naz);

// primjer poziva
Kompleksni *pok1 = new (67, "Naziv") Kompleksni(6.0, 7.0);
```

Prvi parametar daje velièinu bloka kojeg je potrebno alocirati i on je obavezan. Njega izraèunava sam prevoditelj prilikom poziva operatora te ga nije moguæe navesti

prilikom poziva. Preostali parametri se navode u okruglim zagradama prije naziva tipa kojeg je potrebno alocirati. Prilikom traženja prikladne verzije operatora poštuju se pravila preoptereæenja, što znaèi da se parametri moraju poklapati po tipovima i pozicijama. Ako operator `new` nema parametara, zagrade se ne navode. Operator mora vratiti pokazivaè na alociranu memoriju. Isto tako izgleda i deklaracija operatora `new[]`:

```
void *operator new[](size_t velicina, int poz);

// primjer poziva
Vektor *pok2 = new (49) Vektor[4];
```

Podjednako se deklariraju operatori kao funkcijski èlanovi.

Operatori `delete` i `delete []` moraju imati kao povratnu vrijednost `void`, dok im prvi parametar mora biti `void *`. Taj parametar nosi pokazivač na segment memorije koji se treba osloboditi. Ako se operator deklarira kao član klase, može imati drugi parametar tipa `size_t` koji označava veličinu bloka koji se treba osloboditi. Nije dozvoljeno deklarirati druge parametre tih operatora. Evo primjera deklaracije:

```
void operator delete(void *pok);
void operator delete[](void *pok);

// primjeri poziva
delete pok1;
delete [] pok2;
```

Iako klasa može imati preoptereæen operator `new`, ponekad je potrebno pozvati globalnu verziju operatora `new`. To se može uèiniti pomoæu operatora za odreðivanje podruèja tako da se navede `::new`. U suprotnom, ako se operator `::` izostavi, pozvat æe se operator `new` iz klase. Na primjer:

```
class Primjer {
public:
    void *operator new(size_t vel);
};

int main() {
    Primjer *pok1 = new Primjer;// poziva se new iz
                                 // klase Primjer
    Primjer *pok2 = ::new Primjer;  // poziva se globalna
                                 // verzija
    return 0;
}
```

Reèeno vrijedi i za operator `new[]`: ako želimo pozvati globalni operator, napisat æemo `::new[]`.

Praktična primjena preopterećenja operatora `new` i `delete` može biti poboljšanje sustava alokacije memorije koji je bio izložen u poglavlju o ugniježđenim klasama. Svaka alokacija će se bilježiti u posebnu listu koja će se ispisati na kraju programa. Alokacije će se pratiti u listi struktura koje će biti alocirane pomoću funkcije `malloc()` (definirane u standardnoj datoteci zaglavlja `stdlib.h`) jer se unutar operatora `new` ne može pozivati operator `new` (to bi rezultiralo rekurzijom). Memorijski blokovi za objekte će se također alocirati funkcijom `malloc()`, a oslobađati funkcijom `free()`. Operatori `new` i `new[]` će imati dva opcijska parametra koji će pamtiti liniju kôda i naziv datoteke u kojoj je alokacija obavljena.

```
#include <stddef.h>
#include <malloc.h>
#include <string.h>
#include <iostream.h>

struct Alokacija {
    Alokacija *sljeceda;
    void *mjesto;
    int linija;
    size_t velicina;
    char datoteka[80];
};

Alokacija *Prva = NULL, *Zadnja = NULL;

void DodajAlokaciju(int lin, size_t vel, char *dat,
                    void *mj) {
    Alokacija *pom = (Alokacija*)malloc(sizeof(Alokacija));
    pom->sljeceda = NULL;
    pom->linija = lin;
    pom->velicina = vel;
    pom->mjesto = mj;
    if (dat)
        strcpy(pom->datoteka, dat);
    else
        pom->datoteka[0] = 0;
    if (Zadnja)
        Zadnja->sljeceda = pom;
    else
        Prva = pom;
    Zadnja = pom;
}

void UkloniAlokaciju(void *mj) {
    Alokacija *pom = Prva, *pom1 = NULL;
    while (pom) {
        if (pom->mjesto == mj) {
            if (pom1)
                pom1->sljeceda = pom->sljeceda;
            else
```

```
                Prva = pom->sljeceda;
            if (!pom->sljeceda) Zadnja = pom1;
            free(pom);
            break;
        }
        pom1 = pom;
        pom = pom->sljeceda;
    }
}

void *operator new(size_t vel, int lin, char *dat) {
    void *alo = malloc(vel);
    DodajAlokaciju(lin, vel, dat, alo);
    return alo;
}

void *operator new[](size_t vel, int lin, char *dat) {
    void *alo = malloc(vel);
    DodajAlokaciju(lin, vel, dat, alo);
    return alo;
}

void operator delete(void *mjesto) {
    UkloniAlokaciju(mjesto);
}

void operator delete[](void *mjesto) {
    UkloniAlokaciju(mjesto);
}

void IspisiListu() {
    Alokacija *pom = Prva;
    while (pom) {
        cout << pom->linija << ":";
        cout << pom->datoteka << ":";
        cout << pom->velicina << endl;
        pom = pom->sljeceda;
    }
}
```

Svaka alokacija memorije se bilježi u listi. Prilikom izlaska iz programa lista se može ispisati te se može ustanoviti je li sva memorija ispravno vraæena. Operatori `new` i `new[]` imaju dva parametra pomoæu kojih se može locirati mjesto u programu na kojem je alokacija obavljena. Prvi parametar je broj linije kôda, a drugi je naziv datoteke u kojoj je naredba smještena. Pretprocesorski simbol `__LINE__` se prilikom prevođenja zamjenjuje brojem linije, dok se simbol `__FILE__` zamjenjuje nazivom datoteke. Evo primjera poziva preoptereæenog operatora `new`:

```
int main() {
    char *p = new (__LINE__, __FILE__) char[20];
```

```
    Vektor *v = new (__LINE__, __FILE__) Vektor(4.0, 9.0);
    // radi neki posao
    delete [] p;
    delete v;
    // provjeri ima li alokacija koje nisu oslobođene
    IspisiListu();
    return 0;
}
```

Zadatak. *Za opisivanje polinoma u nekom programu koristit ćemo klasu* `Polinom` *deklariranu na sljedeći način:*

```
class Polinom {
private:
    int stupanj;
    float *koef;
public:
    Polinom(int pocetniStupanj);
};
```

Napišite konstruktor koji će inicijalizirati polinom na stupanj zadan parametrom, alocirati polje za koeficijente polja (oprez: broj koeficijenata je `stupanj + 1`*), te postaviti sve koeficijente na nulu. Također, dodajte konstruktor kopije koji će ispravno obaviti kopiranje polinoma. Za ispravno uništavanje polinoma dodajte destruktor koji će osloboditi alociranu memoriju.*

Zadatak. *Omogućite pristup pojedinom koeficijentu polinoma tako da preopteretite operator* `[]`*. Pri tome nula kao parametar označava slobodni član polinoma. Ako korisnik traži član veći od stupnja polinoma, vraća se nula (jer su svi koeficijenti veći od stupnja polinoma uistinu nula).*

Zadatak. *Kako klasa* `Polinom` *koristi dinamičku alokaciju memorije, podrazumijevani operator dodjele neće biti ispravan. Napišite vlastiti operator dodjele koji će to ispraviti.*

Zadatak. *Polinome je moguće zbrajati, oduzimati i množiti. Dodajte operatore* `+`, `-` *i* `*` *koji će to omogućavati. Također, dodajte unarne* `+` *i* `-` *operatore za određivanje predznaka. Uputa: rezultat svih tih funkcija mora biti tipa* `Polinom`*, a ne* `Polinom &`*. Drugim riječima, želimo omogućiti pisanje izraza pomoću polinoma.*

Zadatak. *Zbog potpunosti, pravilo dobrog programiranja nalaže da ako za klasu postoji neki aritmetički operator, primjerice* `+`*, onda treba definirati i odgovarajući operator obnavljajućeg pridruživanja* `+=`*. Dodajte, stoga, klasi* `Polinom` *operatore* `+=`, `-=` *i* `*=`.

Zadatak. *Izračunavanje polinoma* `p` *u točki* `x` *u standardnoj matematičkoj notaciji ima oznaku* `p(x)`*. Omogućite takvu notaciju i u C++ programima tako da preopteretite operator* `()`*. Za izračunavanje polinoma koristite Hornerov algoritam:*

$$r_n = a_n$$
$$r_{n-1} = r_n \times x + a_{n-1}$$
$$\ldots$$
$$r_0 = r_1 \times x + a_0$$

Računaju se redom brojevi r_i po gornjim formulama. Pri tome x predstavlja točku u kojoj se izračunava vrijednost polinoma, a a_i predstavlja i-ti koeficijent polinoma.

10.4. Opæe napomene o preoptereæenju operatora

Preoptereæenje operatora je nesumnjivo vrlo moæan alat za razvoj aplikacija visokog stupnja složenosti no postoje neka pravila koje je dobro imati na umu. Programeri koji se po prvi put susreæu s preoptereæenjem operatora skloni su pretjerivanju u upotrebi tog svojstva jezika. Prvo što se je potrebno upitati prilikom razvoja klase jest kojim je operatorima prikladno opremiti klasu.

Za svaku klasu su definirani operatori `=` (pridruživanje) i `&` (uzimanje adrese). Svi ostali operatori dobivaju svoje značenje tek ako se definiraju za određenu klasu. Podrazumijevani operator pridruživanja radi tako da se svaki član objekta s desne strane jednostavno dodjeli istom članu objekta s lijeve strane. Prvenstveno je potrebno obratiti pažnju na to da li podrazumijevani operator pridruživanja obavlja operaciju koju mi želimo.

Prilikom određivanja skupa potrebnih operatora dobro je početi od analize javnog sučelja klase. Mnogi funkcijski članovi će biti daleko jednostavniji za upotrebu ako se definiraju kao operatori. Zato je dobro prvo odrediti skup potrebnih funkcijskih članova, a potom iz njega izvesti skup potrebnih operatora.

Iako je gotovo sve operatore moguće redefinirati za sasvim proizvoljne funkcije, svi operatori asociraju programera na neku određenu operaciju. Operator `=` ima smisao pridruživanja, dok operator `+` ima smisao zbrajanja. Iako je moguće zamijeniti uloge tih dvaju operatora, dobiveni kôd je potpuno nečitljiv i nerazumljiv, da ne spominjemo probleme koji nastaju u vezi s hijerarhijom operacija. Uputno je definirati `+` operator za klase `Vektor` i `Kompleksni` u svrhu zbrajanje dvaju vektora odnosno kompleksna broja te ne dovoditi čitatelja programa u nedoumicu. Prirodno značenje operatora `+` za klasu `ZnakovniNiz` je nadovezivanje desnog operanda na kraj lijevog. Ako operator nema prirodno asocijativno značenje za određenu klasu, bolje ga je uopće ne definirati.

Često se sljedeći funkcijski članovi, koji su dijelovi javnog sučelja klase, mogu jednostavnije realizirati operatorima:

- `JeLiPrazan()` postaje logièki operator negacije, `operator!`
- `JeLiJednak()` postaje operator jednakosti, `operator==`
- `Kopiraj()` postaje operator pridruživanja, `operator=`

Ako je definiran operator pridruživanja `=` i neki od binarnih operatora, primjerice `+`, tada je uputno definirati i operator obnovljenog pridruživanja `+=` kako se ne bi narušila intuitivnost jezika.

Upotreba operatora u izrazima može znatno degradirati performanse programa zbog niza akcija kopiranja koje se primjenjuju prilikom prosljeđivanja parametara i rezultata operatorskim funkcijama. Promotrimo što se sve događa u naoko bezazlenom dijelu programa (definicija operatora zbrajanja je dana u uvodnom dijelu ovog poglavlja):

```
Kompleksni a(5.0, 2.0), b(2.0, 6.0), c;
c = a + b;
```

Prilikom izvoðenja operacije zbrajanja prosljeðuju se reference na objekte `a` i `b` operatorskoj funkciji. Izraèunava se rezultat te se stvara privremeni objekt koji se vraæa. Zatim se taj privremeni objekt kopira u objekt `c` pomoæu operatora pridruživanja te se uništava.

U ovoj računskoj operaciji smo za prijenos rezultata koristili privremeni objekt koji smo na kraju morali kopirati. Mnogo bolje rezultate prilikom izvođenja postigli bismo da smo napisali funkciju kojoj prosljeđujemo reference na tri objekta, dva parametra i jedan objekt koji će čuvati rezultat. Rezultirajuću vrijednost mogli bismo direktno dodijeliti rezultirajućem objektu. Time bismo izbjegli potrebu za stvaranjem privremenog objekta, njegovim kopiranjem i uništavanjem.

U slučaju klase `Kompleksni` dobici na brzini ne moraju biti spektakularni jer objekt nije dugačak te se njegovo stvaranje, kopiranje i uništavanje može obaviti u razmjerno kratkom vremenu. No zamislimo da radimo s klasom `Matrica` koja ima 100×100 elemenata. Prikladno je definirati operatore za zbrajanje i pridruživanje objekata te klase. Sada potrebna kopiranja, alokacija i dealokacija memorije nisu zanemarivi. Štoviše, vrijeme potrebno za alokaciju, kopiranje i uništavanje matrice može nadmašiti vrijeme potrebno za izračunavanje zbroja. Zato je prikladnije zaobići upotrebu operatora te napraviti funkcijski član koji će kao treći parametar dobiti referencu na objekt u koji se smješta rezultat. Time se izbjegavaju nepotrebna kopiranja.

Pouka ovog razmatranja jest da su operatori vrlo moćan i praktičan alat, no prilikom korištenja potrebno je dobro odvagnuti prednosti i gubitke koji se pojavljuju njihovom primjenom.

11. Nasljeðivanje i hijerarhija klasa

Neka nepoznata gospođa obratila se George Bernard Shawu: 'Vi imate najpametniji mozak na svijetu, a ja imam najljepše tijelo; mi bismo stoga proizveli savršeno dijete.' Gospodin Shaw je odvratio: 'No što ako dijete naslijedi moje tijelo, a Vaš mozak?'

Hesketh Pearson: "Bernard Shaw" (1942)

Svaki ozbiljan objektno orijentirani jezik koji danas postoji ima implementiran barem nekakav rudimentaran mehanizam nasljeðivanja, koji znaèajno može skratiti vrijeme potrebno za razvoj složenih programa. Nasljeðivanje u jeziku C++ je vrlo razraðeno, što omoguæava brzo i efikasno stvaranje novih klasa iz veæ postojeæih.

Osnovna ideja nasljeđivanja jest da se prilikom razvoja identificiraju klase koje imaju sličnu funkcionalnost, te da se u izvedenim klasama samo redefiniraju specifična svojstva, dok se preostala svojstva nasljeđuju u nepromijenjenom obliku. Razumijevanje mehanizama nasljeđivanja i građenja hijerarhije klasa je od ključne važnosti za ispravnu primjenu koncepta objektno orijentiranog programiranja.

11.1. Ima li klasa bogatog strica u *Ameriki*?

Želimo li napraviti dobar program za vektorsko crtanje, bit æe potrebno omoguæiti unos razlièitih grafièkih elemenata na radnu plohu. Radi jednostavnosti izostavimo za poèetak vrlo korisne no pomalo ezoteriène objekte kao što su Bezieréove krivulje i ogranièimo skup objekata na linije, elipsine isjeèke, krugove, poligone i pravokutnike. Ako se detaljnije razmotri dani skup objekata, može se ustanoviti da svi oni imaju niz zajednièkih svojstava: svaki objekt se može nacrtati na željenom podruèju ekrana, može ga se translatirati za neki vektor te rotirati oko neke toèke za željeni kut. Takoðer, svaki objekt može biti nacrtan u odreðenoj boji koja se po želji može mijenjati.

No svaki od tih objekata ima i svoja specifična svojstva koja u potpunosti opisuju upravo njega. Primjerice, linija ima početnu i završnu točku. Elipsin isječak ima središte elipse, veliku i malu poluos te početni i završni kut isječka. Poligon ima niz točaka koje određuju njegove vrhove. Operacija crtanja će za svaki od tih objekata biti obavljena na različit način. Isto vrijedi i za operacije translacije i rotacije.

Prirodno se nameće implementacija navedenih objekata pomoću hijerarhijskog stabla. Osnovna klasa `Grafobjekt` definirat će zajednička svojstva svih grafičkih objekta bez navođenja kako se pojedina operacija mora obaviti. Takva klasa koja postoji samo zato da bi ju se moglo naslijediti zove se *apstraktna klasa* (engl. *abstract class*).

Preostale klase će naslijediti tu klasu te će uvesti nova svojstva potrebna za opisivanje objekata i redefinirati postojeća u skladu sa samim objektom.

Razmotrimo kako će se izgraditi takvo stablo te pokušajmo razlučiti koja svojstva svaka od klasa u hijerarhiji mora definirati. Klasa `GrafObjekt` mora definirati funkcijske članove `Crtaj()`, `Translatiraj()` i `Rotiraj()` za crtanje, translaciju odnosno rotaciju. Ti funkcijski članovi ne mogu biti definirani jer ta klasa opisuje opći grafički objekt i definira njegova opća svojstva. `Translatiraj()` ima dva cjelobrojna parametra koji će pokazivati komponente vektora za koji se translacija obavlja. `Rotiraj()` ima tri brojčana parametra; prva dva označavaju koordinate centra rotacije, dok treći daje kut rotacije. Također, `GrafObjekt` mora sadržavati podatkovni član `Boja` za pamćenje boje objekta. Pretpostavit ćemo da se boja objekta može zapamtiti pomoću cjelobrojnog podatkovnog člana. Kako bi se ostvarila konzistentnost javnog sučelja, taj član će biti privatan, a pristupat će mu se pomoću javnih funkcijskih članova `PostaviBoju()` i `CitajBoju()`. Ti članovi mogu biti u potpunosti definirani jer se boja postavlja i čita na način jednak za sve objekte. Pretpostavit ćemo da možemo sve koordinate pamtiti pomoću cjelobrojnog pravokutnog koordinatnog sustava. Evo deklaracije klase `GrafObjekt`:

```
class GrafObjekt {
private:
    int Boja;
public:
    void PostaviBoju(int nova) { Boja = nova; }
    int CitajBoju() { return Boja; }
    void Crtaj() {}
    void Translatiraj(int, int) {}
    void Rotiraj(int, int, int) {}
};
```

Kako funkcijski èlanovi `Translatiraj()` i `Rotiraj()` ne mogu biti definirani, nije specificirano njihovo tijelo. Parametri su namjerno navedeni bez imena, jer bi se u suprotnom prilikom prevođenja dobilo upozorenje o tome da parametri nisu iskorišteni unutar tijela èlana. Istina, kôd æe se ispravno prevesti, no bolje je ne ignorirati upozorenja i modificirati kôd tako da se upozorenje izbjegne. Ovako se prevoditelju eksplicitno daje na znanje da parametar neæe biti korišten.

Liniju će opisivati klasa `Linija` koja mora uvesti nova svojstva potrebna za ostvarenje njene funkcije. Ta nova svojstva su koordinate početne i završne točke. Zajednička svojstva, kao što su boja i rutine za njeno čitanje i postavljanje, se zadržavaju nepromijenjenima. Iako bi bilo moguće definirati klasu `Linija` tako da se zajednički elementi jednostavno ponove, to je vrlo nepraktično. Takvih elemenata može biti znatno više nego što je novih elemenata. Praktičnije je uzeti postojeći objekt i jednostavno ga obogatiti novim elementima.

Drugo moguće rješenje je u klasu `Linija` uključiti objekt klase `GrafObjekt` kao podatkovni član. Klasa bi tada izgledala ovako:

```
class Linija {
private:
    int x1, y1, x2, y2;
public:
    GrafObjekt osnovna;
    void Crtaj();
    void Translatiraj(int vx, int vy);
    void Rotiraj(int cx, int cy, int kut);
};
```

Kada bi se sada htjelo postaviti boju nekoj liniji, to bi se moralo uèiniti ovako:

```
#define CRVENA      14
Linija l;
l.osnovna.PostaviBoju(CRVENA);
```

Takav pristup ozbiljno narušava pravila skrivanja informacija. Programer mora poznavati implementaciju klase `Linija` kako bi ju direktno koristio. Sljedeæi je nedostatak što su neki èlanovi, primjerice `Crtaj()`, redefinirani u novoj klasi. Tako pozivi

```
l.Crtaj();
l.osnovna.Crtaj();
```

predstavljaju pozive razlièitih funkcija. Programer mora paziti da izmijenjene funkcije poziva direktno na samom objektu, dok se neizmijenjene funkcije moraju pozivati pomoæu objekta osnovne klase. Takvo maltretiranje programera æe sigurno prije ili kasnije rezultirati pogreškama. Moguæe je rješenje ponoviti sve nepromijenjene funkcije i u izvedenoj klasi kao u sljedeæem primjeru:

```
class Linija {
private:
    int x1, y1, x2, y2;
public:
    GrafObjekt osnovna;
    void PostaviBoju(int nb) { osnovna.PostaviBoju(nb); }
    int CitajBoju() { return osnovna.CitajBoju(); }
    void Crtaj();
    void Translatiraj(int vx, int vy);
    void Rotiraj(int cx, int cy, int kut);
};
```

Iako je ovakvo rješenje znatno prikladnije, pisanje kôda je izuzetno zamorno, a sam program je optereæen funkcijama koje služe samo pozivanju funkcija iz osnovne klase.

Jezik C++ na elegantan način rješava sve te probleme pomoću *nasljeđivanja* (engl. *inheritance*). Pri tome se definira nova *izvedena klasa* (engl. *derived class*) na osnovu već postojeće *osnovne klase* (engl. *base class*). Objekti izvedene klase sadrže sve funkcijske i podatkovne članove osnovne klase, te mogu dodati nova svojstva.

Nasljeđivanje se specificira tako da se u deklaraciji klase iza naziva klase navede lista osnovnih klasa, kao u sljedećem primjeru:

```
class Linija : public GrafObjekt {
    // ...
};
```

Lista osnovnih klasa navodi se tako da se iza naziva klase stavi dvotoèka te se navedu nazivi osnovnih klasa razdvojeni zarezom. Ispred svakog od naziva moguæe je staviti jednu od kljuènih rijeèi `public`, `private` ili `protected` èime se navodi tip nasljeðivanja. Ovime se jednim potezom specificira da æe objekt klase `Linija` sadržavati sve funkcijske i podatkovne èlanove klase `GrafObjekt`. Slièan bi se uèinak postigao da se u klasu `Linija` ukljuèio potpun sadržaj osnovne klase. Gornjom deklaracijom je klasa `Linija` izvedena te nasljeðuje svojstva osnovne klase `GrafObjekt`.

Podatkovni i funkcijski članovi osnovne klase u slučaju javnog nasljeđivanja (kada se koristi ključna riječ `public` u listi nasljeđivanja) zadržavaju svoje pravo pristupa. To znači da podatkovni član `Boja` ostaje privatan, te mu se ne može pristupiti niti unutar naslijeđene klase `Linija`, dok članovi `PostaviBoju()` i `CitajBoju()` imaju javni pristup te im se može pristupiti i izvan klase.

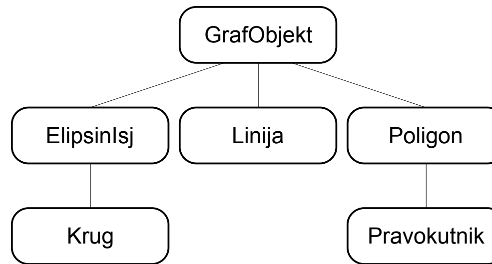
Klasa `Linija` dodaje nove podatkovne članove `x1`, `y1`, `x2` i `y2` koji će pamtiti koordinate početne i krajnje točke linije. U izvedenoj klasi je moguće zamijeniti željene funkcijske članove osnovne klase novim članovima. Taj postupak se zove *zaobilaženje* (engl. *overriding*). Tako klasa `Linija` definira svoj postupak za crtanje koje se razlikovati od postupka za crtanje navedenog u osnovnoj klasi.

Razmotrimo kako se dalje može graditi hijerarhijsko stablo grafičkih objekata. Neka klasa `ElipsinIsj` definira elipsin isječak. Ta klasa sadrži podatkovne članove `centarX` i `centarY` koji pamte koordinate centra elipse, zatim `poluosA` i `poluosB` koji pamte koordinate velike i male poluosi elipse te `pocetniKut` i `zavrsniKut` koji specificiraju početni i krajnji kut crtanja isječka. Kako bi se omogućilo postavljanje parametara objekta, potrebno je dodati funkcijske članove za postavljanje i čitanje svih podatkovnih članova. Na prikladan način se definiraju i funkcijski članovi za crtanje, rotaciju i translaciju.

Krug ćemo opisati klasom `Krug`. Krug je grafički objekt koji je zapravo jedna podvrsta elipsinog isječka, te je najbolji način za implementaciju kruga naslijediti klasu `ElipsinIsj`. Pri tome je potrebno dodati funkcijske članove koji će omogućiti postavljanje i čitanje radijusa. Važno je primijetiti da u ovom slučaju nije potrebno ponovo definirati funkcijski član za crtanje. Kako je krug elipsin isječak koji ima obje poluosi jednake te početni kut 0, a završni 360 stupnjeva, operacija crtanja elipsinog isječka će u ovom slučaju nacrtati krug.

Slična je situacija i s poligonom i pravokutnikom. Klasa `Poligon` opisuje općeniti objekt koji može imati proizvoljno mnogo vrhova. Klasa `Pravokutnik` nasljeđuje od klase `Poligon` te jedino na prikladan način postavlja podatkovne članove objekta. Operacije za translaciju, rotaciju i crtanje nije potrebno ponavljati jer one definirane u

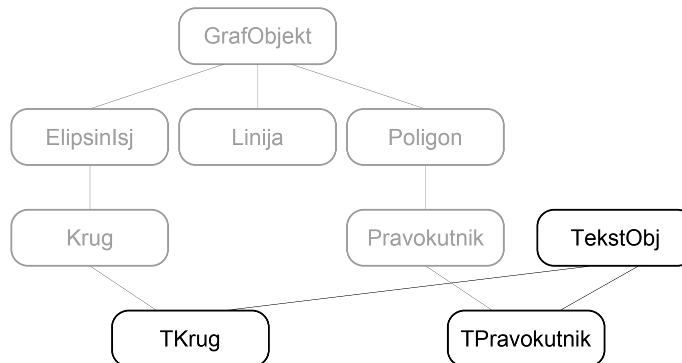
klasi `Poligon` u potpunosti odgovaraju. Cjelokupno hijerarhijsko stablo prikazano je na slici 11.1.



Slika 11.1. Hijerarhijsko stablo grafičkih elemenata

Klasa može naslijediti i nekoliko drugih klasa. U tom sluèaju æe svi podatkovni i funkcijski èlanovi svih osnovnih klasa biti ukljuèeni u novu klasu. Zamislimo da želimo stvoriti nove grafièke objekte koji æe sadržavati neki tekst. Sve postojeæe elemente æemo naslijediti kako bi se dodalo novo funkcijsko svojstvo.

Iako je moguće još jednom proći cijelo hijerarhijsko stablo, te kod svake klase gdje je to potrebno dodavati elemente potrebne za ispis teksta, jednostavnije je definirati zasebnu klasu `TekstObj` koja će definirati tekstovni objekt. Objekti te klase moraju pamtiti znakovni niz koji se ispisuje te poziciju niza na ekranu. Dodavanje teksta na bilo koji postojeći grafički element sada se jednostavno može napraviti nasljeđivanjem klase `TekstObj` i željene klase grafičkog objekta. Na primjer, klasa `TPravokutnik` koja opisuje pravokutnik u kojem je ispisan tekst, nasljeđuje klasu `TekstObj` i klasu `Pravokutnik`, dok se klasa `TKrug` koja opisuje krug s upisanim tekstom dobiva tako da se naslijedi klasa `TekstObj` i klasa `Krug`. Takvo hijerarhijsko stablo prikazano je na slici 11.2.



Slika 11.2. Hijerarhijsko stablo tekstualnih grafičkih elemenata

Ovakav tip nasljeðivanja se zove *višestruko nasljeðivanje* (engl. *multiple inheritance*). Jedna izvedena klasa sada ima više osnovnih klasa.

11.2. Specificiranje nasljeðivanja

Nasljeðivanje se specificira tako da se iza naziva klase umetne lista osnovnih klasa. Ta lista se sastoji od dvotoèke te od naziva osnovnih klasa. Ispred svakog naziva se može navesti jedna od kljuènih rijeèi `public`, `private` ili `protected` kojom æe se opisati tip nasljeðivanja. Evo primjera:

```
// osnovne klase
class GrafObjekt {};
class TekstObj {};

// izvedene klase
class Poligon : public GrafObjekt {};
class Pravokutnik : private Poligon {};
class TPravokutnik : Pravokutnik, public TekstObj {};
```

Klase `GrafObjekt` i `TekstObj` su u ovom primjeru osnovne klase, odnosno klase od kojih se nasljeðuje. Klase `Poligon`, `Pravokutnik` i `TPravokutnik` su izvedene klase, odnosno klase koje nasljeðuju. Kako je prilikom deklaracije klase `Poligon` navedeno javno nasljeðivanje, klasa `GrafObjekt` se naziva *javnom osnovnom klasom* klase `Poligon`. Naprotiv, klasa `Poligon` je *privatna osnovna klasa* za klasu `Pravokutnik`, što je navedeno privatnim nasljeðivanjem. Takoðer može postojati i *zaštiæena osnovna klasa* u sluèaju zaštiæenog nasljeðivanja.

Ako se prilikom nasljeðivanja klasa izostavi tip nasljeðivanja, podrazumijeva se privatno nasljeðivanje. Prilikom nasljeðivanja struktura podrazumijeva se javno nasljeðivanje.

Usprkos tome, dobra je navika i u sluèaju privatnog nasljeðivanja navesti kljuènu rijeè `private` kako bi se izbjegle moguæe zabune. Takav kôd je daleko pregledniji i manje podložan pogreškama.

Prilikom nasljeđivanja, jedna klasa se ne može navesti više nego jednom u listi nasljeđivanja. To je i logično, jer bi se u suprotnom unutar objekta izvedene klase nalazila primjerice dva objekta osnovne klase, što bi rezultiralo nedoumicom kojem se objektu od njih pristupa.

Svi članovi, funkcijski i podatkovni, osnovnih klasa automatski se prenose u izvedenu klasu. Tim članovima je moguće pristupati kao da su navedeni u izvedenoj klasi (s time da vrijede prava pristupa). Zapravo, objekt koji osnovna klasa definira je u cijelosti uključen kao podobjekt u izvedenu klasu. Time je izbjegnuta potreba za prepisivanjem sadržaja klase prilikom nasljeđivanja. Na primjer, neka su klase definirane ovako:

```
class GrafObjekt {                          // osnovna klasa
private:
    int Boja;
public:
    void PostaviBoju(int nova) { Boja = nova; }
    int CitajBoju() { return Boja; }
    void Crtaj() {}
    void Translatiraj(int, int) {}
    void Rotiraj(int, int, int) {}
};

class Linija : public GrafObjekt {          // izvedena klasa
private:
    int x1, y1, x2, y2;
public:
    void Crtaj();
    void Translatiraj(int vx, int vy);
    void Rotiraj(int cx, int cy, int kut);
};
```

Sada možemo deklarirati objekt `ln` klase `Linija` te pristupati èlanovima linije, no takoðer možemo pristupati i èlanovima klase `GrafObjekt`, i to na isti naèin kao da su ti èlanovi navedeni u klasi `Linija` (to je jasno ako se razumije da svaka izvedena klasa sadrži sve èlanove osnovnih klasa):

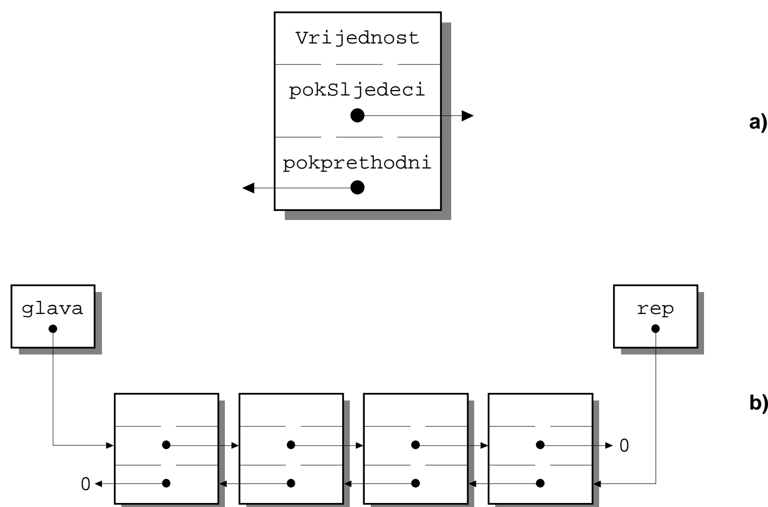
```
Linija ln;
ln.Rotiraj(10, 10, 10); // pristup članu klase Linija
ln.PostaviBoju(CRNA);   // pristup osnovnoj klasi
```

Kako bismo smisao nasljeðivanja pojasnili u praksi, opisat æemo jedan od moguæih naèina za realizaciju dvostruko vezane liste objekata. Vezana lista se èesto koristi u sluèajevima kada nije poznat ukupan broj objekata koji se žele zapamtiti, ili kad se taj broj mijenja tijekom izvoðenja programa. Naime, ako objekte želimo pamtiti u polju, potrebno je prilikom alokacije niza navesti broj objekata. Takoðer, ako želimo ubaciti novi objekt u polje na odreðeno mjesto, ili ako želimo izbrisati neki objekt iz polja, a da pri tome ne ostavimo "rupu", morat æemo premještati èlanove polja uzduž i poprijeko. *Vezana lista* (engl. *linked list*) je u tom sluèaju znatno efikasnije rješenje.

Prije nego što započnemo objašnjavati princip rada liste te damo programski kôd koji ju realizira, evo važne napomene: lista je kontejnerska klasa, a takve klase se u C++ jeziku daleko kvalitetnije rješavaju predlošcima. Tome je više razloga, a osnovni je taj što predlošci znaju točan tip objekta koji se smješta u kontejner. Naprotiv, rješenje koje je ovdje prikazano ne zna točan tip objekta, pa tako kontejner niti ne može pristupiti specifičnostima objekta. Ovakav pristup rezultira intenzivnim korištenjem operatora za dodjelu tipa kako bi se "izigrao" sistem statičkih tipova koji je duboko usađen u C++. Zbog toga će lista biti ponovo obrađena u poglavlju 11. No i ovakva realizacija liste ima svoje prednosti u slučaju kada u listi želimo čuvati objekte različitih klasa. Tada nam predlošci ne pomažu znatno, a korištenje dodjela tipova je neumitno.

Vezana lista se ostvaruje tako da svaki član liste osim vrijednosti sadrži i pokazivač na sljedeći element liste. U jačoj varijanti vezanih lista – dvostruko povezanoj listi – svaki član liste sadržava i pokazivač na prethodni član. U našem primjeru to će biti pokazivači `pokPrethodni` i `pokSljedeci`. Struktura svakog člana liste prikazana je na slici 11.3a. Kako nema člana ispred prvoga, njegov `pokPrethodni` bit će jednak nul-pokazivaču, čime se signalizira početak liste. Ista je stvar i s posljednjim članom liste: njegov `pokSljedeci` će biti jednak nul-pokazivaču čime se označava kraj liste.

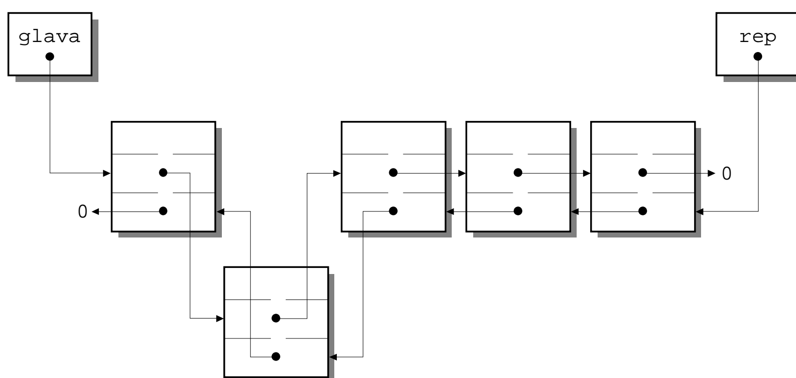
Samoj listi se pristupa preko dva pokazivača: `glava` i `rep`. Pokazivač `glava` pokazuje na prvi član liste, dok pokazivač `rep` pokazuje na posljednji član. Shematski je to prikazano na slici 11.3b. Pojedini član se dohvaća tako da se lista slijedno pretražuje, tako da se krene od glave ili od repa, te se uzastopno iščitavaju pokazivači na sljedeći odnosno prethodni član. Umetanje i brisanje člana iz liste je trivijalno: umjesto da se članovi premještaju po memoriji i time troši vrijeme, potrebno je samo promijeniti pokazivače prethodnog i sljedećeg člana. Ono što listama gubimo jest mogućnost izravnog dohvaćanja člana pomoću indeksa – da bismo dohvatili pojedini član uvijek je potrebno krenuti od glave ili od repa i postupno doći do željenog člana.



Slika 11.3. Struktura dvostruko vezane liste

Zamislimo sada da želimo u listu sa slike 11.3b umetnuti novi èlan između prvog i drugog èlana. Pri tome je samo potrebno preusmjeriti pokazivaè `pokSljedeci` prvog èlana i pokazivaè `pokPrethodni` drugog èlana na novi èlan. Pokazivaèe `pokPrethodni` i `pokSljedeci` novoga èlana potrebno je usmjeriti na prvi odnosno drugi èlan liste, kao na slici **11.4**.

Definirat ćemo klasu `Atom` koja će opisivati objekt koji se drži u listi. Ta klasa će sadržavati pokazivač na prethodni i sljedeći član u listi, koji će biti privatni kako bi im se spriječio pristup izvan klase. Također, klasa `Atom` će sadržavati funkcijske članove za



Slika 11.4. Umetanje novog člana u dvostruko vezanu listu

postavljanje i čitanje pokazivača. Ta klasa definira opća svojstva i mehanizme elementa u listi.

```
class Atom {
private:
    Atom *pokSljedeci, *pokPrethodni;
public:
    Atom *Sljedeci() { return pokSljedeci; }
    Atom *Prethodni() { return pokPrethodni; }
    void StaviSljedeci(Atom *pok) { pokSljedeci = pok; }
    void StaviPrethodni(Atom *pok) { pokPrethodni = pok; }
};
```

Evo i klase `Lista` koja æe sadržavati funkcijske èlanove za umetanje i brisanje èlanova. Pri tome klasa `Lista` nije odgovorna za alokaciju memorije za pojedini èlan liste. To mora uèiniti vanjski program pomoæu operatora `new`. Funkcijskim èlanovima klase `Lista` se tada samo prosljeðuje pokazivaè na taj èlan.

```
class Lista {
private:
    Atom *glava, *rep;
public:
    Lista() : glava(NULL), rep(NULL) {}
    Atom *AmoGlavu() { return glava; }
    Atom *AmoRep() { return rep; }
    void UgurajClan(Atom *pok, Atom *izaKojeg);
    void GoniClan(Atom *pok);
};

void Lista::UgurajClan(Atom *pok, Atom *izaKojeg) {
    // da li se dodaje na početak?
    if (izaKojeg != NULL) {
        // ne dodaje se na početak
```

```
        // da li se dodaje na kraj?
        if (izaKojeg->Sljedeci() != NULL)
            // ne dodaje se na kraj
            izaKojeg->Sljedeci()->StaviPrethodni(pok);
        else
            // dodaje se na kraj
            rep = pok;
        pok->StaviSljedeci(izaKojeg->Sljedeci());
        izaKojeg->StaviSljedeci(pok);
    }
    else {
        // dodaje se na početak
        pok->StaviSljedeci(glava);
        if (glava != NULL)
            // da li već ima članova u listi?
            glava->StaviPrethodni(pok);
        glava = pok;
    }
    pok->StaviPrethodni(izaKojeg);
}

void Lista::GoniClan(Atom *pok) {
    if (pok->Sljedeci() != NULL)
        pok->Sljedeci()->StaviPrethodni(pok->Prethodni());
    else
        rep = pok->Prethodni();
    if (pok->Prethodni() != NULL)
        pok->Prethodni()->StaviSljedeci(pok->Sljedeci());
    else
        glava = pok->Sljedeci();
}
```

Funkcijski èlan `UgurajClan()` æe umetnuti èlan u listu. Pokazivaè `pok` æe pokazivati na èlan koji se umeæe, dok æe pokazivaè `izaKojeg` pokazivati na èlan koji je veæ u listi iza kojeg želimo ugurati novi èlan. Ako èlan želimo dodati na poèetak, `izaKojeg` mora biti jednak `NULL`. Funkcijski èlan `GoniClan()` æe izbaciti èlan iz liste. Pri tome æe pokazivaè `pok` pokazivati na èlan kojeg želimo izbrisati.

Klase čije objekte želimo smjestiti u listu moraju naslijediti klasu `Atom`. Na primjer, klasa `LVektor` će definirati vektor koji se može smjestiti u listu:

```
class LVektor : public Atom {
private:
    float ax, ay;
public:
    LVektor(float a = 0, float b = 0) : ax(a), ay(b) {}
    void PostaviXY(float x, float y) { ax = x; ay = y; }
    float DajX() { return ax; }
```

```
    float DajY() { return ay; }
};
```

Kao posljedica nasljeðivanja, u svakom `LVektor` objektu postoji podobjekt `Atom` koji sadržava sve potrebno da bi se objekt mogao pohraniti u listu. Prisutni su svi podatkovni i funkcijski èlanovi. Javnim èlanovima je moguæe pristupiti i na klasièan naèin:

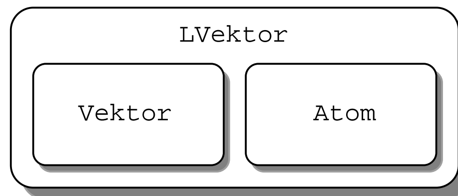
```
LVektor lv;
lv.StaviSljedeci(NULL);
if (lv.Prethodni()) {
    // ...
}
```

Podatkovnim èlanovima `pokSljedeci` i `pokPrethodni` se ne može pristupiti niti iz klase `LVektor` niti izvana zato jer su definirani kao privatni. Da su definirani kao javni, moguæe bi im bilo pristupiti na klasièan naèin.

Ako je klasa izvedena od više osnovnih klasa, govori se o višestrukom nasljeđivanju. Tada svaki od objekata iz osnovne klase postoji kao podobjekt u izvedenoj klasi. Klasu `LVektor` moguće je dobiti iz klasa `Atom` i `Vektor` pomoću višestrukog nasljeđivanja na sljedeći način:

```
class LVektor : public Atom, public Vektor {
public:
    LVektor(float a = 0, float b = 0) : Vektor(a, b) {}
};
```

Sada uopæe nije potrebno ponavljati funkcijske èlanove za postavljanje i èitanje realnog i imaginarnog dijela; oni su jednostavno naslijeðeni iz klase `Vektor`. Grafièki se objekt klase `LVektor` može prikazati kao na slici **11.5**.



Slika 11.5. Grafički prikaz klase `LVektor`

Evo kako se objekti mogu smještati u listu i èitati iz nje:

```
Lista lst;

// punjenje liste
lst.UgurajClan(new LVektor(10.0, 20.0));
lst.UgurajClan(new LVektor(-5.0, -6.0));
```

```
// ...

// čitanje liste
LVektor *pok = (LVektor *)lst.AmoGlavu();
while (pok) {
    // obrada vektora, na primjer ispis:
    cout << "(" << pok->DajX() << ","
         << pok->DajY() << ")" << endl;
    pok = (LVektor *)pok->Sljedeci();
}
```

Primijetite operatore za dodjelu tipa koji se koriste za dobivanje pokazivaèa na ispravan tip. Oni èine kôd neèitljivijim te su èest uzrok pogreškama, ali su u našem primjeru neophodni. Naime, klasa `Lista` vraæa pokazivaè na `Atom` – ona ne zna za tip `LVektor`. Kako bi se dobilo na sigurnosti programa, umjesto statièkih dodjela tipa bolje bi bilo koristiti dinamièke dodjele, koje æe biti opisane u poglavlju 13. Te dodjele bi se mogle eliminirati upotrebom predložaka. No ako bismo u listi htjeli držati objekte razlièitih klasa, ovo nam je jedino rješenje.

Zadatak. *Klasi `Lista` dodajte operacije za dodavanje člana na početak i kraj liste. Također, dodajte član `UbaciListu()` koji će kao parametar imati referencu na objekt klase `Lista`. Taj član će sve objekte iz liste navedene u parametru ubaciti u listu pridruženu objektu čiji je član pozvan.*

11.3. Pristup naslijeðenim èlanovima

Naslijeðenim èlanovima se može pristupati jednostavno navoðenjem njihovog imena. Iako se tako èini da dotièni èlanovi pripadaju izvedenim klasama, oni ostaju u podruèju osnovne klase. Svakom èlanu se može pristupiti i preko osnovne klase pomoæu operatora `::` za odreðivanje podruèja. Na primjer:

```
LVektor lv;
lv.Atom::StaviSljedeci(NULL);
lv.Vektor::PostaviXY(5, 8);
```

U mnogim sluèajevima je odreðivanje podruèja nepotrebno jer prevoditelj može jednoznaèno odrediti èlan bez dodatnog navoðenja. No u dva posebna sluèaja je potrebno dodatno odrediti èlan:

- kada se u izvedenoj klasi deklarira istoimeni član,
- kada u osnovnim klasama postoje dva ili više članova istog naziva.

Èlan u izvedenoj klasi skriva istoimeni èlan u osnovnoj klasi. No èlanu osnovne klase se može pristupiti tako da se navede podruèje kojemu èlan pripada pomoæu operatora za odreðivanje podruèja. Ta je situacija slièna onoj kada lokalni identifikator skriva globalni identifikator. U oba sluèaja navedeni identifikator oznaèava entitet iz neposrednog podruèja. Na primjer:

```
#include <iostream.h>

class Osnovna {
public:
    int i;
    void Var() { cout << "Osnovna::i " << i << endl; }
};

class Izvedena : public Osnovna {
public:
    int i;      // prekriva Osnovna::i !!!
    void Int() { cout << "Izvedena::i " << i << endl; }
};
```

Identifikator `i` u klasi `Izvedena` prekriva identifikator `i` u klasi `Osnovna`. Pridruživanje èlanu bez navoðenja podruèja ima smisao pridruživanja èlanu klase `Izvedena`. Èlanu iz klase `Osnovna` pristupa se pomoæu navoðenja podruèja. To ilustrira sljedeæi programski kôd:

```
int main() {
    Izvedena izv;
    izv.i = 9;           // pristupa se Izvedena::i
    izv.Osnovna::i = 20; // pristupa se Osnovna::i
    izv.Var();           // ispisuje 'Osnovna::i 20'
    izv.Int();           // ispisuje 'Izvedena::i 9'
    return 0;
}
```

Situacija je nešto složenija ako u osnovnim klasama postoje dva ili više ista identifikatora.

Ako dvije osnovne klase sadrže isti identifikator, njegovo navoðenje bez specificiranja pripadnog podruèja je neprecizno te rezultira pogreškom prilikom prevoðenja.

Upotreba operatora za odreðivanje podruèja je obavezna kako bi se jednoznaèno odredio èlan kojem se pristupa. Na primjer, za deklaracije

```
class A {
public:
    void Opis() { cout << "Ovo je klasa A" << endl; }
};

class B {
public:
    void Opis() { cout << "Ovo je klasa B" << endl; }
};

class D : public A, public B { };
```

sljedeæi poziv je dvosmislen:

```
D obj;
obj.Opis(); // pogreška: dvosmislenost
```

Objekt `D` u biti posjeduje dva funkcijska èlana `Opis`, jedan u `A` dijelu i jedan u `B` dijelu. Prilikom poziva neophodno je eksplicitno odrediti kojemu se pristupa:

```
D obj;
obj.A::Opis();  // ispisuje 'Ovo je klasa A'
obj.B::Opis();  // ispisuje 'Ovo je klasa B'
```

Ovakav naèin odreðivanja podruèja ima dva vrlo nezgodna nedostatka:

1. Onemogućen je mehanizam pozivanja virtualnih funkcijskih članova. Naime, kada se navede područje funkcijskog člana pomoću operatora za određivanje područja, onemogućava se virtualan poziv funkcije te se poziv veže statički (smisao i značenje virtualnih funkcija će biti kasnije objašnjen).
2. Navedena nejednoznačnost se proteže u daljnjim nasljeđivanjima. Pravilo dobrog objektno orijentiranog dizajna kaže da izvedene klase ne bi smjele uvesti dodatne implementacijske detalje izvan sučelja definiranog osnovnom klasom.

Èesto je prikladno stoga u izvedenoj klasi definirati istoimeni funkcijski èlan koji æe objediniti funkcionalnost oba podèlana. Na primjer:

```
class D : public A, public B {
public:
    void Opis() { cout << "Ovo je klasa D" << endl; }
};
```

Time se jednostavno uklanja svaka moguæa nejednoznaènost. Sada je poziv

```
D obj;
obj.Opis(); // ispisuje 'Ovo je klasa D'
```

potpuno ispravan. Naravno da je još je uvijek moguæe pristupiti èlanovima iz klase `A` i `B`, kao na primjer:

```
obj.A::Opis(); // ispisuje 'Ovo je klasa A'
```

11.4. Nasljeðivanje i prava pristupa

Veæ je objašnjeno da postoje tri tipa prava pristupa: javni, privatni i zaštiæeni. Èlanovima s javnim pristupom se može pristupiti i izvan klase, dok se elementima s privatnim i zaštiæenim pristupom može pristupiti iskljuèivo unutar klase. Dodjeljujuæi

prava pristupa pojedinim èlanovima klase definira se javno suèelje pomoæu kojeg objekt komunicira s okolinom.

Uvođenjem nasljeđivanja, osim javnog sučelja namijenjenog korisnicima klase, potrebno je uvesti i sučelje vidljivo programerima koji će izvoditi nove klase iz postojeće. Time je moguće sakriti osjetljive informacije u izvedenoj klasi što osigurava konzistentnost objekta.

Pravo pristupa članu u izvedenoj klasi određeno je pravom pristupa u osnovnoj klasi te vrstom nasljeđivanja. Kako postoje tri prava pristupa te tri moguća tipa nasljeđivanja, imamo ukupno devet mogućih kombinacija.

11.4.1. Zaštiæeno pravo pristupa

Osim veæ poznatih javnih i privatnih prava pristupa, postoji i *zaštiæeno* (engl. *protected*) pravo pristupa koje primarno služi stvaranju suèelja za nasljeðivanje. Ono se specificira pomoæu kljuène rijeèi `protected`:

```
class GrafObjekt {
protected:                  // zaštićeni članovi
    int Boja;
public:
    void PostaviBoju(int nova) { Boja = nova; }
    int CitajBoju() { return Boja; }
    void Crtaj() {}
    void Translatiraj(int, int) {}
    void Rotiraj(int, int, int) {}
    void PostaviBrojac(int br) { Brojac = br; }
};
```

U gornjem primjeru klasa `GrafObjekt` definira èlan `Boja` zaštiæenim. Njemu se i dalje ne može pristupiti izvan klase, na primjer, pomoæu objekta te klase, no moguæe ga je koristiti u funkcijskim èlanovima izvedenih klasa. Na primjer:

```
class Linija : public GrafObjekt {
private:
    int x1, y1, x2, y2;
public:
    void Crtaj();
    void Translatiraj(int vx, int vy);
    void Rotiraj(int cx, int cy, int kut);
};

void Linija::Crtaj() {
    // crni objekti se ne crtaju
    if (Boja == CRNA) return;
    // ...
}
```

Klasa `Linija` može potpuno regularno pristupati èlanu `Boja`. To nipošto ne treba shvatiti da je moguæe tom èlanu pristupiti pomoæu objekta klase `Linija`:

```
Linija ln;
ln.Boja = CRNA;     // pogreška
```

Èlan `Boja` je jednostavno dio suèelja za nasljeðivanje: time se odreðuje koji èlanovi su namijenjeni korištenju u izvedenim klasama. Privatni èlanovi nisu dostupni u izvedenim klasama, a javni èlanovi su dostupni i korisnicima objekta.

Zaštiæeni èlanovi nisu javno dostupni, ali su dostupni iz naslijeðenih klasa i èine suèelje za nasljeðivanje.

Na kraju je još potrebno napomenuti da pravo pristupa èlanovima osnovne klase u izvedenoj klasi ovisi o tipu nasljeðivanja, o èemu æe biti više rijeèi u narednim odjeljcima.

11.4.2. Javne osnovne klase

Odreðena klasa je *javna osnovna klasa* (engl. *public base class*) ako je u listi prilikom nasljeðivanja navedena pomoæu kljuène rijeèi `public`. Svi elementi osnovne klase bit æe ukljuèeni u izvedenu klasu u kojoj æe zadržati svoje originalno pravo pristupa. Privatni èlanovi neæe biti dostupni iz izvedenih klasa niti iz preostalog programa. Zaštiæenim èlanovima moæi æe se pristupiti iz izvedene klase, ali ne i iz glavnog programa. Javni èlanovi ostat æe dostupni i iz glavnog programa i iz izvedene klase. To se može pokazati sljedeæim primjerom:

```
class GrafObjekt {
private:
    int Brojac;
protected:
    int Boja;
public:
    void PostaviBoju(int nova) { Boja = nova; }
    int CitajBoju() { return Boja; }
    void Crtaj() {}
    void Translatiraj(int, int) {}
    void Rotiraj(int, int, int) {}
    void PostaviBrojac(int br) { Brojac = br; }
};

class Linija : public GrafObjekt {
private:
    int x1, y1, x2, y2;
public:
    void Crtaj();
    void Translatiraj(int vx, int vy);
```

```
    void Rotiraj(int cx, int cy, int kut);
};
```

Klasi `GrafObjekt` je dodan cjelobrojni èlan `Brojac` koji pamti redni broj objekta i inicijalizira se pomoæu javnog funkcijskog èlana `PostaviBrojac()`. Kako je èlan privatan, to znaèi da mu se ne može pristupiti iz klase `Linija`. Svaki objekt klase æe sadržavati taj èlan, no neæe mu se moæi pristupiti, na primjer:

```
void Linija::Crtaj() {
    if (Brojac) { // pogreška: član nije dostupan
        // ...
    }
}
```

Èlan `Brojac` je na taj naèin iskljuèen iz suèelja predviðenog za nasljeðivanje. Moguæe mu je pristupiti iz same klase `GrafObjekt` ili iz klase proglašene prijateljem klase `GrafObjekt`. Èlan `Boja` je, naprotiv, deklariran kao zaštiæeni èlan. Izvan klase mu nije moguæe pristupiti, no kako je `GrafObjekt` javna osnovna klasa za klasu `Linija`, moguæe mu je pristupiti iz klase `Linija`. Na primjer:

```
#define PLAVA       35
#define BIJELA      22

void Linija::Crtaj() {
    if (Boja == PLAVA) { // OK: Boja je zaštićeni član
        // ...
    }
}

int main() {
    Linija l;
    l.Boja = BIJELA;// pogreška: član Boja nije
                    // dostupan izvan klase
    return 0;
}
```

To znaèi da su zaštiæeni èlanovi iskljuèeni iz javnog suèelja klase, ali su ukljuèeni u suèelje namijenjeno nasljeðivanju. Javni èlanovi osnovne klase su dostupni i izvan programa i u izvedenoj klasi. Oni su dio i javnog suèelja i suèelja namijenjenog nasljeðivanju.

Javne osnovne klase se razlikuju od privatnih i zaštićenih osnovnih klasa po važnom svojstvu:

Moguæa je implicitna pretvorba pokazivaèa i referenci izvedene klase u njenu javnu osnovnu klasu.

Na primjer, pokazivaè na objekt `Linija` može biti implicitno pretvoren u pokazivaè na objekt `GrafObjekt`. To je dozvoljeno, zato jer svaki objekt klase `Linija` u sebi sadrži jedan podobjekt klase `GrafObjekt`. Prilikom pretvorbe æe se vrijednost pokazivaèa na objekt `Linija` promijeniti tako da se dobije pokazivaè na podobjekt `GrafObjekt`. Na primjer:

```
Linija *pokLinija = new Linija;
GrafObjekt *pokObjekt = pokLinija; // implicitna pretvorba
pokObjekt->PostaviBoju(PLAVA);
```

Može se postaviti pitanje zašto je implicitna pretvorba pokazivaèa i referenci ogranièena na javne osnovne klase. Da bi se to razjasnilo, potrebno je uoèiti da je prilikom javnog nasljeðivanja cjelokupno javno suèelje osnovne klase ukljuèeno u izvedenu klasu. To znaèi da prilikom pretvorbe pokazivaèa ne postoji opasnost da se time dozvoli pristup nekom èlanu koji se ne nalazi u javnom suèelju izvedene klase.

11.4.3. Privatne osnovne klase

Klasa je *privatna osnovna klasa* (engl. *private base class*) ako se u listi nasljeðivanja navede s kljuènom rijeèi `private`. Takoðer, ako se izostavi kljuèna rijeè koja odreðuje tip nasljeðivanja, podrazumijeva se privatno nasljeðivanje.

Prilikom privatnog nasljeđivanja, privatni članovi osnovne klase nisu dostupni izvedenoj klasi, dok javni i zaštićeni članovi osnovne klase postaju privatni članovi izvedene klase. Zamislimo da želimo na što jednostavniji način od klase `Vektor` načiniti klasu `Kompleksni`. Iako bi sa stanovišta principa ispravnog objektno orijentiranog dizajna bilo ispravnije napisati klasu `Kompleksni` neovisno od klase `Vektor`, na prvi pogled se čini da vektori i kompleksni brojevi imaju mnogo zajedničkih svojstava: i jedni i drugi imaju dva realna člana koji pamte *x* i *y* vrijednosti, odnosno realni i imaginarni dio. Na jednak se način zbrajaju i oduzimaju, pa čak i množe sa skalarom. Razlika je primjerice u tome što ne postoji skalarni produkt kompleksnih brojeva; produkt kompleksnih brojeva se računa po drugim pravilima. Zato se može činiti prikladnim da se klasa `Kompleksni` jednostavno dobije nasljeđivanjem klase `Vektor`. Pretpostavimo za početak da koristimo javno nasljeđivanje:

```
class Vektor {
private:
    float ax, ay;
public:
    Vektor(float a = 0, float b = 0) : ax(a), ay(b) {}
    void PostaviXY(float x, float y) { ax = x; ay = y; }
    float DajX() { return ax; }
    float DajY() { return ay; }
    float MnoziSkalarnoSa(Vektor &vekt);
};
```

```
class Kompleksni : public Vektor {
public:
    Kompleksni(float a = 0, float b = 0) : Vektor(a, b) {}
    void MnoziSa(Kompleksni &broj);
};
```

Nedostatak ovakve hijerarhije klasa je u tome što je cjelokupno javno suèelje klase `Vektor` ukljuèeno u javno suèelje klase `Kompleksni`. To znaèi da æe i funkcijski èlan `MnoziSkalarnoSa()` biti dostupan u klasi `Kompleksni`. Time je omoguæeno vektorsko množenje kompleksnih brojeva koje nema smisla i ne smije biti dozvoljeno:

```
Kompleksni z1, z2;
z1.MnoziSkalarnoSa(z2); // logička pogreška: operacija
                        // nema smisla za kompleksne
                        // brojeve
```

Bolje je rješenje prilikom izvoðenja klase `Kompleksni` koristiti privatno nasljeðivanje. Time svi javni i zaštiæeni èlanovi osnovne klase postaju privatni èlanovi izvedene klase. Javno suèelje osnovne klase ne ukljuèuje se u javno suèelje izvedene klase; izvedena klasa mora sama definirati svoje suèelje. Definicija klase `Kompleksni` u tom sluèaju izgledala bi ovako:

```
class Kompleksni : private Vektor {
public:
    Kompleksni(float a = 0, float b = 0) : Vektor(a, b) {}
    float Realni() { return DajX(); }
    float Imaginarni() { return DajY(); }
    void MnoziSa(Kompleksni &broj);
};
```

Sada nije više moguæe pozvati primjerice èlan `MnoziVektorskiSa()` na objektu klase `Kompleksni`.

Privatno nasljeđivanje se koristi u slučajevima kada izvedena klasa nije podvrsta osnovne klase nego osnovna i izvedena klasa samo dijele implementacijske detalje.

Nije moguæa implicitna konverzija iz izvedene klase u privatnu osnovnu klasu. Ako bi to bilo dozvoljeno, tada bi sljedeæi programski odsjeèak omoguæio pristup klasi na neprikladan naèin preko javnog suèelja osnovne klase:

```
Kompleksni *pokKompl = new Kompleksni(4.0, 6.0);

Vektor *pokVekt = pokKompl;     // ne smije biti dozvoljeno!

// pokVekt zapravo pokazuje na kompleksni broj koji se ne
```

```
// može množiti skalarno s vektorom
pokVekt->MnoziSkalarnoSa(Vektor(7.0, 8.0)); // besmislica
```

Ovako je omoguæeno korištenje objekata na nepredviðen naèin te je uèinjena ista pogreška kao i prilikom javnog nasljeðivanja. Zbog toga æe prevoditelj dojaviti pogrešku prilikom prevoðenja gornjeg programa.

Ako je apsolutno sigurno da je potrebna gornja konverzija, moguće ju je provesti tako da se navede eksplicitna pretvorba:

```
pokVekt = (Vektor *)pokKompl;
```

Sada programer sam odgovara za eventualne neželjene efekte prilikom korištenja konvertiranog pokazivaèa.

11.4.4. Zaštiæene osnovne klase

Zaštiæeno nasljeðivanje se specificira tako da se prije naziva osnovne klase navede kljuèna rijeè `protected`. Time se svi javni i zaštiæeni èlanovi *zaštiæene osnovne klase* (engl. *protected base class*) prenose kao zaštiæeni u izvedenu klasu. Cjelokupno javno suèelje osnovne klase se prenosi kao suèelje za nasljeðivanje u izvedenu klasu. Sve izvedene klase mogu imati pristup èlanovima osnovne klase, a ostatak programa tim èlanovima ne može pristupiti. Na primjer:

```
class Osnovna {
public:
    int i;
protected:
    int j;
private:
    int k;
};

class Izvedena : protected Osnovna {
public:
    void Pristupi();
};

void Izvedena::Pristupi() {
    i = 8;  // OK: član je preuzet kao protected
    j = 9;  // OK: član je preuzet kao protected
    k = 10; // pogreška: član je bio privatan
}

int main() {
    Izvedena obj;
    obj.i = 7;  // pogreška: član je zaštićen
    obj.j = 6;  // pogreška: član je također zaštićen
    obj.k = 5;  // pogreška: član je privatan za osnovnu
```

```
                // klasu
    return 0;
}
```

Ne postoji standardna konverzija pokazivaèa na izvedenu klasu u pokazivaè na osnovnu klasu, osim u podruèju izvedene klase. Na primjer:

```
int main() {
    // pogreška: nema ugrađene konverzije izvan klase
    Osnovna *pok = new Izvedena();
    return 0;
}
void Izvedena::Pristupi() {
    // OK: ugrađena konverzija je ispravna unutar klase
    Osnovna *pok = new Izvedena();
}
```

Razlozi za onemoguæavanje ugraðene pretvorbe izvan klase su identièni onima zbog kojih nije omoguæena pretvorba u privatnu osnovnu klasu: smisao zaštiæenog nasljeðivanja je skrivanje javnog suèelja osnovne klase od vanjskog programa. Uz dozvoljenu pretvorbu, javno suèelje bi prešutno postalo dostupno i vanjskom programu. Drukèije su okolnosti unutar klase: javno suèelje osnovne klase je regularno dostupno i unutar klase te zbog toga pretvorba ne može narušiti princip skrivanja informacija. Ako je izrièito potrebna pretvorba izvan klase, moguæe ju je postiæi pomoæu operatora za dodjelu tipa.

U tablici **Error! Reference source not found.** pregledno su dane sve moguće kombinacije prava pristupa prilikom nasljeđivanja. U gornjem redu tablice nalazi se pravo pristupa člana u osnovnoj klasi, a s lijeve strane popisani su tipovi navođenja.

Tablica 11.1. Pregled tipova nasljeđivanja i prava pristupa

Tip nasljeđivanja	Pravo pristupa u osnovnoj klasi		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

11.4.5. Posebne napomene o pravima pristupa

U prethodnim odsjeècima navedeno je da izvedena klasa ima u sluèaju javnog ili zaštiæenog nasljeðivanja pristup naslijeðenim javnim i zaštiæenim èlanovima. Na prvi

pogled može se pomisliti da time nasljeðivanje definira odnos slièan prijateljstvu klasa, no to nije tako. Vrlo je važno uoèiti bitnu razliku između nasljeðivanja i prijateljstva.

Prijateljska funkcija ili klasa ima potpuna prava pristupa svim članovima objekata neke klase. Uz modifikacije u deklaraciji klase `GrafObjekt`, funkcija `AnimirajBoju()` može pristupati svim članovima objekta:

```
class GrafObjekt {
friend void AnimirajBoju(); // prijateljska funkcija
private:
    int Brojac;
protected:
    int Boja;
public:
    void PostaviBoju(int nova) { Boja = nova; }
    int CitajBoju() { return Boja; }
    void Crtaj() {}
    void Translatiraj(int, int) {}
    void Rotiraj(int, int, int) {}
    void PostaviBrojac(int br) { Brojac = br; }
};

void AnimirajBoju() {
    GrafObjekt obj;
    obj.Brojac++;                   // dozvoljeno
    obj.Boja = (obj.Boja + 1) % 16; // također dozvoljeno
}
```

Klasa `Linija` dobivena je javnim nasljeðivanjem:

```
class Linija : public GrafObjekt {
private:
    int x1, y1, x2, y2;
public:
    void Crtaj();
    void Translatiraj(int vx, int vy);
    void Rotiraj(int cx, int cy, int kut);
};
```

Kako je svaka `Linija` ujedno i `GrafObjekt`, funkcija `AnimirajBoju()` imat æe dozvoljen pristup samo èlanovima klase `Linija` naslijeðenim od klase `GrafObjekt`. Na primjer:

```
void AnimirajBoju() {
    Linija l;
    l.Boja = BIJELA;// dozvoljeno
    l.x1 = 8;           // nije dozvoljeno
}
```

Ako bismo htjeli da funkcija ima pristup èlanu `x1`, potrebno je funkciju uèiniti prijateljem klase `Linija`.

Funkcijski članovi izvedenih klasa imaju mogućnost pristupa javim i zaštićenim naslijeđenim članovima osnovnih klasa, no nemaju nikakvo posebno pravo pristupa članovima objekata osnovnih klasa. Na primjer, funkcijski član `Crtaj()` klase `Linija` može pristupiti naslijeđenom članu `Boja`, no ne može mu pristupiti u nekom drugom objektu:

```
void Linija::Crtaj() {
    GrafObjekt go;
    Linija ln;
    Boja = CRVENA;  // OK: pristupa se naslijeđenom članu
    go.Boja = CRNA; // pogreška: nema prava pristupa
                    // objektima osnovnih klasa
    ln.Boja = CRNA; // OK: postoje sva prava pristupa za
                    // objekt iste klase
}
```

Funkcije prijatelji klasa imaju ista prava pristupa kao da su èlanovi te klase. To znaèi da u sluèaju da je prijateljstvo funkcije `AnimirajBoju()` pomaknuto u klasu `Linija`, sljedeæi pristupi bi bili moguæi:

```
class GrafObjekt {
private:
    int Brojac;
protected:
    int Boja;
public:

    // funkcijski članovi su nepromijenjeni
};

class Linija : public GrafObjekt {
friend void AnimirajBoju();
private:
    int x1, y1, x2, y2;
public:

    // funkcijski članovi su nepromijenjeni
};

void AnimirajBoju() {
    Linija ln;
    GrafObjekt go;
    ln.Boja = BIJELA;   // OK
    go.Boja = BIJELA;   // pogreška
    ln.Brojac = CRNA;   // pogreška
}
```

Pristup `ln.Boja` je ispravan jer svaki funkcijski èlan klase `Linija` (a time i svaki prijatelj klase) ima pristup naslijeðenim èlanovima osnovne klase. Pristup `go.Boja` nije ispravan jer funkcijski èlan klase `Linija` (a time i svaki prijatelj klase) nema posebna prava pristupa èlanovima objekata osnovnih klasa. Pristup `ln.Brojac` nije ispravan jer je `Brojac` èlan osnovne klase s privatnim pristupom.

11.4.6. Iskljuèivanje èlanova

Prilikom privatnog ili zaštiæenog nasljeðivanja ponekad može biti prikladno ostaviti originalno pravo pristupa pojedinom èlanu osnovne klase. Na primjer, prilikom privatnog izvoðenja klase `Kompleksni` iz klase `Vektor`, javni èlanovi `DajX()` i `DajY()` automatski postaju skriveni. No kompleksni brojevi imaju svoj realni i imaginarni dio kojima bi se takoðer moglo pristupiti pomoæu tih èlanova. Iako je moguæe u klasi `Kompleksni` ponoviti te èlanove tako da se oni jednostavno pozovu èlanove osnovne klase, to samo dodatno unosi probleme u stablo nasljeðivanja. Zbog toga je moguæe iskljuèiti pojedini èlan iz privatnog nasljeðivanja te mu ostaviti njegovo osnovno pravo pristupa:

```
class Vektor {
private:
    float ax, ay;
public:
    Vektor(float a = 0, float b = 0) : ax(a), ay(b) {}
    void PostaviXY(float x, float y) { ax = x; ay = y; }
    float DajX() { return ax; }
    float DajY() { return ay; }
    float MnoziSkalarnoSa(Vektor &vekt);
};

class Kompleksni : public Vektor {
public:
    Kompleksni(float a = 0, float b = 0) : Vektor(a, b) {}
    void MnoziSa(Kompleksni &broj);
    Vektor::DajX;   // isključuje DajX iz mehanizma
                    // privatnog nasljeđivanja
    Vektor::DajY;   // isključuje DajY iz mehanizma
                    // privatnog nasljeđivanja
};
```

Iskljuèivanje se provodi tako da se u odgovarajuæem dijelu izvedene klase navede identifikator èlana osnovne klase s punim imenom. Pri tome se ne navodi tip èlana ukoliko se radi o podatkovnom èlanu, odnosno ne navodi se povratna vrijednost i parametri ako se radi o funkcijskom èlanu.

Ovim načinom nije moguće promijeniti pravo pristupa člana iz osnovne klase, već samo zadržati osnovno pravo pristupa. To znači da nije bilo moguće isključivanje članova `DajX()` i `DajY()` provesti u zaštićenoj ili privatnoj sekciji klase `Kompleksni` jer bi se time mijenjalo pravo pristupa određeno u osnovnoj klasi.

U završnu varijantu jezika C++ uvedena je *deklaracija* `using` (engl. `using` *declaration*) koja omogućava bolje isključivanje pojedinih članova iz nasljeđivanja. Moguće je da će se isključivanje opisano u ovom odsječku nakon nekog vremena isključiti iz jezika.

Zadatak. *Implementirajte gore navedene klase grafičkih objekata s obzirom na grafičke mogućnosti vašeg računala. Dodajte novi grafički objekt* `PuniPravokutnik` *koji će opisivati pravokutnik ispunjen nekom bojom.*

Zadatak. *Pomoću klase* `Tablica` *definirane u poglavlju 6 o klasama, potrebno je ostvariti klasu* `Stog`*. Ta klasa je slična klasi* `Tablica` *utoliko što se članovi pamte u nizu. Razlika je u tome što se novi članovi mogu dodavati isključivo na vrh stoga, te se mogu skidati samo s vrha stoga. Klasa* `Stog` *mora imati članove* `Stavi()` *i* `Uzmi()` *za stavljanje broja na stog te za skidanje broja sa stoga. Kako* `Stog` *nije* `Tablica`*, javno sučelje tablice nije poželjno uključiti u javno sučelje stoga, pa zato koristite privatno nasljeđivanje.*

11.5. Nasljeđivanje i pripadnost

Važno je razumjeti osnovni smisao nasljeđivanja. Ono opisuje relaciju *biti* između objekata izvedene klase i objekata osnovne klase. Svaki `Pravokutnik` *jest* ujedno i `Poligon`, dok svaki `Poligon` *jest* ujedno i `GrafObjekt`. Stoga svaka operacija primijenjena na `GrafObjekt` ima smisla ako je primijenjena na `Pravokutnik` ili `Poligon`.

Nasljeđivanjem se svi članovi osnovne klase uključuju u objekt izvedene klase. Svaki objekt izvedene klase sadržava podobjekt osnovne klase. Zbog te činjenice moguće je nasljeđivanje upotrijebiti na krivi način tako da se tim mehanizmom pokuša opisati relacija *sadrži*.

Na primjer, poligon sadrži četiri linije. Neiskusan programer bi mogao ustanoviti da bi možda bilo pogodno da se klasa `Pravokutnik` realizira tako da se naslijedi klasa `Linija` čime se pojednostavljuje problem: jedna linija je prisutna kao podobjekt osnovne klase:

```
class Pravokutnik : public Linija {
    // ...
};
```

No to znatno narušava koncept objektno orijentiranog programiranja i nasljeđivanja. `Pravokutnik` nije `Linija`. Klasa `Linija` može primjerice sadržavati funkcijski èlan koji vraæa dužinu linije. Dužina je atribut koji nema smisla za poligon; tamo je moguæe uvesti funkciju koja izraèunava površinu pravokutnika. Time se narušava princip po kojem svaka operacija koja ima smisla za izvedenu klasu mora imati smisla i za osnovnu klasu.

Relacije tipa *sadrži* se opisuju pripadnošću: kako se poligon sastoji od četiri linije, ispravna implementacija klase `Pravokutnik` je ona koja nasljeđuje klasu `GrafObjekt` i sadrži četiri objekta klase `Linija`:

```
class Pravokutnik : public GrafObjekt {
private:
    Linija L1, L2, L3, L4;
public:
    // funkcijski članovi
};
```

11.6. Inicijalizacija i uništavanje izvedenih klasa

Uvoðenjem nasljeðivanja potrebno je proširiti mehanizam konstruktora kako bi se omoguæila ispravna inicijalizacija izvedenih klasa. Konstruktor izvedene klase zadužen je za inicijalizaciju iskljuèivo èlanova definiranih u pripadnoj klasi; èlanovi osnovnih klasa se moraju inicijalizirati u konstruktoru osnovne klase. Kako bi se odredio naèin na koji se inicijalizira objekt osnovne klase, konstruktor osnovne klase se može navesti u inicijalizacijskoj listi konstruktora izvedene klase ispisivanjem naziva osnovne klase i navoðenjem parametara konstruktora.

Modificirat ćemo klase koje definiraju grafičke objekte tako da im dodamo konstruktore. Klasa `GrafObjekt` sadržavat će konstruktor koji će postaviti boju na vrijednost zadanu parametrom. Konstruktor klase `Linija` imat će pet parametra: boju i četiri koordinate koje opisuju početnu i završnu točku linije. Evo kôda koji to opisuje:

```
class GrafObjekt {
private:
    int Brojac;
protected:
    int Boja;
public:

    GrafObjekt(int b = CRNA) : Boja(b) {}

    // ostali članovi su nepromijenjeni

};

class Linija : public GrafObjekt {
private:
    int x1, y1, x2, y2;
public:

    Linija(int b, int px, int py, int zx, int zy) :
        GrafObjekt(b), x1(px), y1(py), x2(zx), y2(zy) {}

    // ostali članovi su nepromijenjeni
```

```
};
```

Konstruktori svih osnovnih klasa se mogu navesti u inicijalizacijskoj listi izvedene klase. Ako se neka od osnovnih klasa ne navede u inicijalizacijskoj listi, prevoditelj æe automatski umetnuti kôd koji poziva podrazumijevani konstruktor osnovne klase. U sluèaju klase `Linija`, izostavljanje konstruktora `GrafObjekt` iz inicijalizacijske liste rezultiralo bi postavljanjem poèetne boje na crnu (jer upravo to radi podrazumijevani konstruktor klase).

Ako izvedena klasa nema konstruktor, prevoditelj će automatski generirati podrazumijevani konstruktor koji će pozivati podrazumijevane konstruktore osnovnih klasa (koji u tom slučaju moraju postojati).

Prilikom inicijalizacije objekta konstruktori osnovnih klasa izvode se prije nego što se poène s izvoðenjem konstruktora izvedene klase. Kako bi se jednoznaèno definirao naèin na koji se inicijaliziraju objekti klase, ustanovljen je obavezan redoslijed inicijalizacije klase:

1. Konstruktori osnovnih klasa izvode se prema redoslijedu navedenom u listi izvođenja klasa. Važno je uočiti da se ne poštuje redoslijed navođenja u inicijalizacijskoj listi (kao što se i podatkovni članovi inicijaliziraju po redoslijedu navođenja u deklaraciji klase, a ne po redoslijedu navođenja u inicijalizacijskoj listi).
2. Svaki se konstruktor člana izvodi u redoslijedu navođenja objekata (a ne u redoslijedu navođenja u inicijalizacijskoj listi).
3. Na kraju se izvodi konstruktor izvedene klase.

Redoslijed pozivanja konstruktora je uvijek ovakav te nije podložan implementacijskim zavisnostima.

Redoslijed pozivanja konstruktora osnovne klase definiran je redoslijedom navođenja osnovnih klasa u listi nasljeđivanja. Konstruktori objekata članova se izvode prema redoslijedu deklaracije objekata u klasi.

Za ispravno uništavanje klase koriste se destruktori. Prilikom uništavanja objekta poziva se destruktor klase koji uništava samo dio objekta koji je definiran izvedenom klasom. Automatski æe se pozvati destruktor osnovnih klasa èime æe se osloboditi dijelovi objekta koji su definirani osnovnim klasama. Redoslijed pozivanja destruktora je toèno obrnut od redoslijeda pozivanja konstruktora.

Zadatak. *Objasnite zašto navedene deklaracije klasa ne daju očekivane rezultate:*

```
class Prva {
public:
    Prva(int);
};

class Druga {
```

```
public:
    Druga(Prva &);
};

class Treca : public Druga, public Prva {
public:
    Treca(int a) : Prva(a), Druga(*this) {}
};
```

Zadatak. *Dodajte klasi `Stog` iz zadatka na stranici 354 konstruktor koji će kao parametar imati cijeli broj koji će definirati maksimalan broj elemenata u stogu.*

11.7. Standardne pretvorbe i nasljeðivanje

Postoje èetiri standardne pretvorbe koje se mogu provesti između izvedene klase i osnovnih klasa:

1. Objekt izvedene klase može se implicitno pretvoriti u objekt javne osnovne klase.
2. Referenca na objekt izvedene klase može se implicitno pretvoriti u referencu na objekt javne osnovne klase.
3. Pokazivač na objekt izvedene klase može se implicitno pretvoriti u pokazivač na objekt javne osnovne klase.
4. Pokazivač na član izvedene klase može se implicitno pretvoriti u pokazivač na član javne osnovne klase.

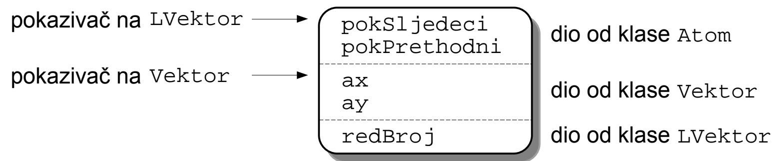
Dodatno, pokazivaè na bilo koju klasu može se implicitno pretvoriti u pokazivaè na tip `void *`. Pokazivaè na tip `void *` se može jedino eksplicitnom dodjelom tipa pretvoriti u pokazivaè na bilo koji drugi tip.

Svaki objekt izvedene klase sadržava po jedan podobjekt svake osnovne klase. Zbog toga je pretvorba izvedene klase u osnovnu sigurna. Obrnuto ne vrijedi. Uzmimo za primjer stablo nasljeđivanja koje je već bilo navedeno u prethodnim poglavljima (klasa `LVektor` proširena članom `redBroj` koji sadrži redni broj člana u listi):

```
class Atom {
private:
    Atom *pokSljedeci, *pokPrethodni;
public:
    // funkcijski članovi nisu bitni
};

class Vektor {
private:
    float ax, ay;
public:
    // funkcijski članovi nisu bitni
};

class LVektor : public Atom, public Vektor {
public:
```

```
        int redBroj;
        // funkcijski članovi nisu bitni
    };
```

Objekt klase `LVektor` se može prikazati kao na slici 11.6.



Slika 11.6. Struktura objekta `LVektor`

Pokazivaè na `LVektor` pokazivat æe na poèetak objekta. Prilikom pretvorbe u recimo pokazivaè na `Vektor`, bit æe ga potrebno uveæati tako da pokazuje na poèetak podobjekta `Vektor`.

Ako je potrebno pretvoriti pokazivač na objekt osnovne klase u pokazivač na objekt izvedene klase, potrebno je koristiti eksplicitnu pretvorbu. Razmotrimo okolnosti zbog kojih je to tako:

```
    // pv se inicijalizira tako da pokazuje u biti na objekt
    // klase LVektor
    Vektor *pv = new LVektor;

    // plv se inicijalizira tako da se pv eksplicitno pretvori
    // u pokazivač na LVektor; što je OK, jer pv pokazuje u
    // biti na objekt klase LVektor
    LVektor *plv = (LVektor *)pv;

    // dodjela je u redu, jer plv pokazuje na LVektor
    plv->redBroj = 5;
```

U gornjem primjeru pokazivaè `pv` na objekt klase `Vektor` se pretvara u pokazivaè `plv` na objekt klase `LVektor`. Takva konverzija se ne može obaviti automatski, jer svaki objekt klase `Vektor` ne sadržava podobjekt `LVektor`. Prevoditelj ne zna u trenutku prevođenja da li `pv` pokazuje na objekt `LVektor` te ima li stoga takva konverzija smisla. Zbog toga programer daje eksplicitnom dodjelom tipa prevoditelju na znanje da se dodjela smije obaviti te da sve eventualne nepoželjne posljedice pretvorbe snosi sam programer. U ovom sluèaju sve je u redu, no evo primjera gdje korištenje takve pretvorbe može prouzroèiti probleme:

```
    // pv se inicijalizira tako da pokazuje u biti na objekt
    // klase Vektor
    Vektor *pv = new Vektor;
```

```
// plv se inicijalizira tako da se pv eksplicitno pretvori
// u pokazivač na LVektor; što je pogrešno, jer pv pokazuje
// na objekt klase Vektor
LVektor *plv = (LVektor *)pv;

// dodjela će prouzročiti probleme, jer plv pokazuje na
// objekt Vektor, te će se broj 5 dodijeliti u nepoznato
// područje memorije
plv->redBroj = 5;
```

U ovom sluèaju programer je natjerao prevoditelja da pretvori pokazivaè na objekt `Vektor` u pokazivaè na objekt `LVektor`. Dodjela èlanu `redBroj` promijenit æe sada neko nepoznato memorijsko podruèje. Velika je vjerojatnost da æe se promijeniti neki èlan nekog sasvim drugog objekta koji je smješten u memoriji na tom mjestu. Zbog toga ovakav program ima velike šanse zablokirati raèunalo.

Pokazivači na članove klasa se ponašaju obrnuto od običnih pokazivača. Kako svaka izvedena klasa sadrži sve članove osnovne klase, dozvoljeno je provesti implicitnu konverziju pokazivača na član osnovne klase u pokazivač na član izvedene klase:

```
void Linija::Crtaj() {
    int Linija::*pokClan;

    // dozvoljena implicitna konverzija jer je Boja
    // prisutna i u osnovnoj i u izvedenoj klasi
    pokClan = &GrafObjekt::Boja;
}
```

Obrnuta konverzija, to jest konverzija pokazivaèa na èlan izvedene klase u pokazivaè na èlan osnovne klase nije dozvoljena, jer osnovna klasa ne sadržava sve èlanove koji su prisutni u izvedenoj klasi:

```
void Linija::Crtaj() {
    int GrafObjekt::*pokClan;

    // pogreška: za implicitnu konverziju je potrebna
    // eksplicitna dodjela tipa
    pokClan = &Linija::x1;

    // ovo je dozvoljeno, ali nije sigurno
    pokClan = (int GrafObjekt::*)&Linija::x1;

    // evo zašto je gornja dodjela opasna: pokClan sada
    // pokazuje na područje izvan objekta GrafObjekt te
    // će sljedeća dodjela članu objekta izazvati
    // upisivanje u memoriju izvan objekta
    GrafObjekt go;
    go.*pokClan = 8;
```

```
        // sljedeća dodjela je u redu, jer se provodi dodjela
        // objektu klase Linija
        this->*pokClan = 9;

}
```

U gornjem primjeru, nakon eksplicitne dodjele tipa pokazivaè `pokClan` se postavlja tako da pokazuje na èlan klase `Linija`. Kako taj èlan ne postoji u klasi `GrafObjekt`, `pokClan` æe pokazivati na èlan izvan objekta `GrafObjekt`. Pridruživanje broja 8 pomoæu varijable `pokClan` æe rezultirati pridruživanjem u memorijsko podruèje izvan objekta èime æe se vjerojatno uništiti neki drugi objekt u raèunalu. Naprotiv, pridruživanje broja 9 pomoæu pokazivaèa `this` nije opasna jer pokazivaè `this` pokazuje na objekt klase `Linija` u kojoj postoji èlan na koji pokazivaè pokazuje.

11.8. Podruèje klase i nasljeðivanje

Svaka izvedena klasa definira svoje podruèje identifikatora u koje se smještaju nazivi deklarirani unutar klase. Kako se svaka izvedena klasa bazira na nekoj postojeæoj osnovnoj klasi, potrebno je definirati odnose izmeðu podruèja izvedene i osnovne klase.

Nasljeđivanje rezultira ugnježđivanjem područja. Može se zamisliti da je područje izvedene klase okruženo područjem osnovne klase. Kada se pokušava pronaći neki identifikator, pretražuje se prvo područje klase kojoj objekt pripada. Ako se identifikator ne može pronaći u tom području, pretražuju se područja osnovnih klasa sve dok se ne dođe do datotečnog područja.

U slučaju višestrukog nasljeđivanja pretražuju se područja svih osnovnih klasa. Na primjer, neka je potrebno pronaći identifikator `pokSljedeci` u funkcijskom članu `DajSljedeci()`:

```
class Atom {
protected:
    Atom *pokSljedeci;
    // ...
};

class Vektor {
    // ...
};

class LVektor : public Atom, public Vektor {
    // ...
    Atom *DajSljedeci() { return pokSljedeci; }
};
```

Prvo se pretražuje lokalno podruèje funkcijskog èlana. Kako ne postoji identifikator s tim nazivom u tom podruèju, pretražuje se nadreðeno podruèje, a to je podruèje klase `LVektor`. Kako takav èlan ne postoji niti u podruèju klase kojoj funkcijski èlan pripada,

pretražuje se postoji li takav naslijeðeni èlan u bilo kojoj osnovnoj klasi. Ako takav èlan postoji u više osnovnih klasa, onda nije jednoznaèno jasno kojem se èlanu pristupa te prevoditelj javlja pogrešku. Zatim se po istom principu pretražuje podruèje osnovne klase sve dok se ne pronaðe traženi identifikator. Ako se identifikator ne pronaðe, javlja se pogreška. Tako je u gornjem primjeru èlan `pokSljedeci` u funkcijskom èlanu `DajSljedeci()` identificiran kao naslijeðeni èlan `Atom::pokSljedeci`.

11.8.1. Razlika nasljeðivanja i preoptereæenja

Valja dobro uoèiti i razumjeti razliku između preoptereæenja funkcijskih èlanova i nasljeðivanja. Promotrimo naèin na koji prevoditelj razlikuje èlanove po potpisu (na osnovu parametara navedenih u pozivu) u sluèaju kada su pojedini identifikatori navedeni u razlièitim podruèjima:

```
class Osnovna {
public:
    void funkcija();
};

class Izvedena : public Osnovna {
public:
    void funkcija(int);
};

int main() {
    Izvedena obj;
    // pogreška: nasuprot očekivanju da će donji poziv
    // pozvati Osnovna::funkcija() to ne funkcionira jer
    // član Izvedena::funkcija(int) skriva istoimeni član
    // osnovne klase
    obj.funkcija();
    return 0;
}
```

Nasljeðivanje nije vrsta preoptereæenja. Funkcijski èlan u izvedenoj klasi prekrit æe istoimeni èlan u osnovnoj klasi iako se oni razlikuju po potpisu. Sve funkcije nekog skupa preoptereæenih funkcija moraju biti definirane u istom podruèju. Ako bismo željeli zadržati varijantu èlana `funkcija()` s jednim parametrom i u klasi `Izvedena`, morali bismo ponoviti njegovu definiciju:

```
class Izvedena : public Osnovna {
public:
    void funkcija(int);
    void funkcija() { Osnovna::funkcija(); }
};
```

11.8.2. Ugniježðeni tipovi i nasljeðivanje

Na ugniježðene tipove se takoðer primjenjuju pravila pristupa i nasljeðivanja kao i za obiène podatkovne i funkcijske èlanove. Na primjer, pretpostavimo da želimo razviti klasu koja opisuje objekt koji pamti skup nizova znakova. Neka ta klasa podržava niz operacija kojima se može dodati element u skup, provjeriti da li neki element veæ postoji u skupu te po redu proæi kroz sve elemente skupa. Uzmimo da želimo realizirati skup pomoæu vezane liste kako se ne bi postavljalo ogranièenje na broj elemenata liste.

Za takvu realizaciju potrebne su nam dvije klase: `SkupNizova` i `Elem`. Prva klasa realizira sam skup, dok druga klasa realizira pojedini element skupa. Kako korištenje klase `Elem` nema smisla izvan konteksta klase `SkupNizova`, željeli bismo onemogućiti stvaranje objekata klase izvan samog skupa. To možemo postići primjerice tako da konstruktor klase `Elem` postavimo privatnim ili zaštićenim. Da bi se omogućilo stvaranje elemenata unutar klase `SkupNizova`, potrebno je tu klasu učiniti prijateljem klase `Elem` čime joj se stavlja na raspolaganje konstruktor klase. To izgleda ovako:

```
class SkupNizova {
    // ...
};

class Elem {
friend class SkupNizova;
protected:
    Elem(char *niz);
};
```

Ovakav pristup ima više nedostataka. Kao prvo, naziv `Elem` se pojavljuje u globalnom podruèju imena iako se ne može koristiti iz tog podruèja. Time se onemoguæava njegova upotreba za klasu koja æe imati smisla u globalnom podruèju. Nadalje, ako se kasnije želi proširiti skup novim operacijama, to se može izvesti tako da se naslijedi klasa `SkupNizova`. Problem je što je novu klasu potrebno navesti prijateljem klase `Elem` kako bi dotièna klasa imala pristup konstruktoru te mogla stvarati objekte klase. Konaèno, moguæe bi bilo naslijediti klasu `Elem` kako bi se primjerice osigurao rad dvostruke povezane liste. U tom sluèaju, naslijeðena klasa bi morala ponoviti sve klase kao prijatelje da bi im omoguæila stvaranje objekata klase. Takvo pisanje složenih programa je vrlo nepraktièno.

Ti problemi se mogu riješiti tako da se klasa `Elem` ugnijezdi u područje klase `SkupNizova`. Ako se postavi u privatni ili zaštićeni dio, glavni program neće imati pristup identifikatoru te će time automatski biti onemogućeno stvaranje objekata klase. Nadalje, naziv `Elem` bit će automatski uklonjen iz globalnog područja.

Ako se klasa `Elem` ugnijezdi u zaštićeni dio klase `SkupNizova`, sve klase koje nasljeđuju klasu `SkupNizova` automatski će moći koristiti klasu `Elem`. Kako nema više opasnosti od pristupa izvan klase `SkupNizova`, klasa `Elem` sada može imati i javni konstruktor, te više nema potrebe za davanjem posebnih prava pristupa klasama koje koriste klasu `Elem`. Opisane deklaracije klasa sada bi izgledale ovako:

```
class SkupNizova {
protected:

    class Elem {
        Elem(char *niz);
        // ...
    };

    // ...

};
```

Pristup ugniježðenoj klasi iz izvedene klase slijedi prava pristupa za obiène èlanove: moguænost pristupa ovisi o pravu pristupa u osnovnoj klasi i o tipu nasljeðivanja, a pravila koja to reguliraju opisana su u prethodnim odjeljcima.

Članovi ugniježđene klase se mogu definirati i izvan okolne klase. Tada se područja jednostavno nadovezuju operatorom `::` po sljedećem principu:

```
SkupNizova::Elem::Elem(char *niz) {
    // ...
}
```

Takoðer, ugniježðena klasa može poslužiti kao objekt nasljeðivanja, s time da u tom sluèaju ona mora biti javno dostupna:

```
class SkupNizova {
public:

    class Elem {
        Elem(char *niz);
    };
};

class NaslijediOd : public SkupNizova::Elem {
    // ...
};
```

11.9. Klase kao argumenti funkcija

Kako objekti mogu biti proslijeðeni funkcijama preko parametara, potrebno je dodatno proširiti mehanizme preoptereæenja funkcija èime bi se omoguæilo jednoznaèno pronalaženje željene funkcije.

Ako funkcija ima za parametre ugrađene tipove, a u pozivu nije naveden isti tip (na primjer parametar kaže da se očekuje `int`, a u pozivu je stavljen `long`), tada se koriste pravila standardnih konverzija kojima se navedeni parametar pretvara u očekivani. Slično se može dogoditi da skup preopterećenih funkcija očekuje jednu klasu, a poziv u

listi parametara navodi neku drugu klasu. Jezik C++ definira niz pravila kojima se tada navedeni tip može pokušati svesti na tip naveden u parametrima preopterećene funkcije i tako odrediti poziv. Ta pravila uključuju točno podudaranje tipova, primjenu standardne konverzije i primjenu korisnički definiranih konverzija.

11.9.1. Toèno podudaranje tipova

Ako je objekt naveden u pozivu funkcije primjerak klase koja je navedena u parametrima, onda se ta dva tipa toèno podudaraju te je poziv preoptereæene funkcije time odreðen. Ovo pravilo još ukljuèuje pretvorbu objekta u referencu na tip objekta pomoæu trivijalne konverzije. Na primjer:

```
void Rastegni(GrafObjekt &obj);
void Rastegni(Linija &obj);

Linija lin;

// zbog točnog podudaranja donji poziv će pozvati
// Rastegni(Linija&)
Rastegni(lin);
```

U gornjem primjeru se poziv funkcije preusmjerava na funkciju koja ima toèan tip kao parametar. Prilikom preoptereæenja funkcija važno je imati na umu da algoritam za odreðivanje argumenata ne može razlikovati objekt i referencu na taj objekt, te æe takva preoptereæenja prouzroèiti pogrešku prilikom preoptereæenja:

```
// pogreška: nije moguće razlikovati objekt i referencu na
// objekt
void Rastegni(Linija &obj);
void Rastegni(Linija obj);
```

11.9.2. Standardne konverzije

Ako ne postoji toèno podudaranje tipova, prevoditelj æe pokušati svesti tip stvarnog argumenta na tip naveden u parametrima funkcije pomoæu standardne konverzije.

Objekt izvedene klase, referenca ili pokazivač će se implicitno pretvoriti u odgovarajući tip javne osnovne klase. Na primjer:

```
void Deformiraj(GrafObjekt &obj);

Linija lin;
// OK: lin se konvertira u GrafObjekt&
Deformiraj(lin);
```

Pokazivaè na bilo koju klasu se implicitno konvertira u tip `void *`, na primjer:

```
void PopuniNulama(void *pok);

// OK: provodi se konverzija u void *
PopuniNulama(&lin);
```

Važno je uoèiti da ne postoji standardna konverzija u izvedenu klasu (razlozi za to su dani u odjeljku 11.7), pa æe prilikom prevođenja donjeg kôda prevoditelj javiti pogrešku:

```
void ProduljiLiniju(Linija &obj);

GrafObjekt grobj;
// pogreška: nema konverzije u izvedenu klasu
ProduljiLiniju(grobj);
```

Ako postoje dvije funkcije od kojih u parametrima svaka ima po jednu neposrednu javnu osnovnu klasu, poziv funkcije se smatra automatski nejasnim te se javlja pogreška prilikom prevođenja. Programer tada mora pomoæu eksplicitne dodjele tipa odrediti koja se varijanta funkcije poziva:

```
void Obradi(Atom &obj);
void Obradi(Vektor &obj);

LVektor lv;
//pogreška: postoje funkcije s obje javne osnovne klase
Obradi(lv);

// programer određuje poziv funkcije Obradi(Vektor&) pomoću
// eksplicitne dodjele tipa
Obradi((Vektor&)lv);
```

Ako postoji više osnovnih klasa, pri èemu je samo jedna bliža klasi navedenog objekta, provodi se pretvorba u taj tip te se poziva odgovarajuæa funkcija:

```
class A { };
class B : public A { };
class C : public B { };

void funkcija(A &obj);
void funkcija(B &obj);

int main() {
    C objc;
    // OK: poziva se funkcija(B &) jer je klasa B bliža
    // klasi C nego klasa A
    funkcija(objc);
    return 0;
}
```

Tip `void *` je najudaljeniji od bilo koje klase, odnosno neki pokazivaè æe biti pretvoren u taj tip samo ako niti jedna druga konverzija nije moguæa.

11.9.3. Korisnièki definirane pretvorbe

Ako prevoditelj ne uspije pomoæu prethodna dva pravila pronaæi traženu preoptereæenu funkciju, pokušat æe primijeniti korisnièki definiranu konverziju kako bi sveo navedeni tip na neki od ugraðenih tipova.

Ako primjerice radimo program koji računa s kompleksnim brojevima, razvit ćemo klasu `Kompleksni`. No svi realni brojevi su ujedno i kompleksni brojevi kojima je imaginarni dio nula te bi bilo pogodno da se, na mjestu gdje se očekuje kompleksni broj, može prihvatiti i realan broj koji se zatim interpretira kao kompleksan. Tada bismo mogli napraviti funkcije `Korijen1()` i `Korijen2()` koje izračunavaju prvu i drugu vrijednost korijena iz kompleksnog broja te bismo ih mogli koristiti i za obične realne brojeve. Konverziju realnog broja u kompleksni možemo obaviti konstruktorom:

```
class Kompleksni {
private:
    double cx, cy;
public:
    Kompleksni(double a = 0, double b = 0)
        : cx(a), cy(b) {}
    // ...
};
```

Funkcije `Korijen1()` i `Korijen2()` imale bi sljedeæu deklaraciju:

```
Kompleksni Korijen1(const Kompleksni &kompl) {
    // ...
}

Kompleksni Korijen2(const Kompleksni &kompl) {
    // ...
}
```

Ako bismo pozvali primjerice funkciju `Korijen1()` tako da joj kao parametar navedemo realan broj, prvo bi se pozvao konstruktor klase `Kompleksni` koji bi konvertirao realan broj u kompleksni tako što bi stvorio privremeni objekt. Taj objekt bi se zatim proslijedio pozvanoj funkciji. Na primjer:

```
Kompleksni kor = Korijen1(3.9);
```

Ovako korisnik klase tretira realne i kompleksne brojeve na isti naèin. No i cijeli brojevi su kompleksni. Na svu sreæu, nije potrebno posebno navoditi konstruktor koji æe konvertirati cijele brojeve u kompleksne zato jer postoji ugraðena konverzija između cijelih i realnih brojeva. Poziv

```
Kompleksni kor = Korijen1(12);
```

rezultirat æe proširivanjem cijelog broja `12` na tip `double` te æe se zatim pozvati odgovarajuæi konstruktor za konverziju. Konverzija tipa se provodi uvijek u jednom koraku, što znaèi da prevoditelj neæe uzastopno pozivati konverziju u neki tip koji æe se zatim konvertirati u neki drugi tip kako bi se postigao željeni tip prilikom poziva funkcije. Korisnièki definirane konverzije æe se koristiti samo ako se ne uspije pronaæi funkcija za poziv koje su dovoljne trivijalne i standardne pretvorbe. Mogli bismo funkcije preopteretiti i tako da brže rade za realne brojeve (njihovi korijeni se lakše izraèunavaju nego korijeni iz opæenitih kompleksnih brojeva):

```
Kompleksni Korijen1(double broj) {
    // ...
}

Kompleksni Korijen2(double broj) {
    // ...
}
```

Poziv koji navodi tip za koji se ne mora koristiti korisnièka konverzija

```
Kompleksni kompl = Korijen1(5.6);
```

rezultirat æe pozivom funkcije `Korijen1(double)`, a ne konverzijom u kompleksni broj i pozivom funkcije `Korijen1(Kompleksni &)`.

Prevoditelj će primijeniti standardnu konverziju na rezultat korisnički definirane konverzije ako se time može postići podudaranje parametara. To se često može koristiti u kombinaciji sa standardnim konverzijama u osnovnu klasu. Zamislimo da želimo u sklopu neke matematičke aplikacije prikazivati vektore kao linije koje izlaze iz ishodišta. U prethodnim poglavljima razvili smo klasu `Linija` koja sadrži sve potrebne podatkovne i funkcijske članove za opis linije. Svaki vektor se može pretvoriti u liniju tako da se prva koordinata linije postavi u ishodište, a druga se postavi u točku do koje vektor seže. Deklaracija klase `Vektor` s takvom konverzijom izgleda ovako:

```
class Vektor {
    // ...
    operator Linija();
};
```

Ako postoji funkcija koja kao parametar oèekuje objekt klase `GrafObjekt`, moguæe joj je kao parametar navesti objekt klase `Vektor`:

```
void NestoRadi(GrafObjekt &go) {
    // ...
}

Vektor v;
```

```
NestoRadi((Linija)v);
// rezultira konverzijom objekta klase Vektor u objekt
// klase Linija koja se ugrađenom konverzijom može
// svesti na klasu GrafObjekt
```

Operatori konverzije nasljeðuju se po identiènim pravilima po kojima se nasljeðuju obiène funkcije.

Postoje li dvije korisničke konverzije koje se mogu ravnopravno primijeniti, prevoditelj će prilikom prevođenja dojaviti pogrešku o nejasnom pozivu. Na primjer, ako bismo omogućili konverziju neke klase `A` i u klasu `Kompleksni` i u tip `double`, poziv funkciji `Korijen1()` s objektom klase `A` kao parametrom bio bi nejasan:

```
class A {
public:
    // ...
    operator double();
    operator Kompleksni();
};

A obj_a;

Korijen1(obj_a);    // pogreška prilikom prevođenja: nije
                    // jasno pretvara li se obj_a u
                    // double ili u Kompleksni
```

11.10. Nasljeðivanje preoptereæenih operatora

U poglavlju o preoptereæenju operatora objašnjeno je kako su preoptereæeni operatori u suštini zapravo samo posebni funkcijski èlanovi. Prilikom nasljeðivanja operatorskih funkcija vrijede pravila iznesena za nasljeðivanje obiènih funkcija, no u nekim sluèajevima njihova primjena nije oèita i sama po sebi razumljiva. Zbog toga æemo pogledati neke od tipiènih sluèajeva.

Jedno od osnovnih pravila nasljeđivanja kaže da nasljeđivanje ima drukčija svojstva od preopterećenja, o čemu čak i iskusniji korisnici C++ jezika često ne vode računa.

Preopterećene funkcije uvijek moraju biti navedene u istom području imena, dok funkcije u naslijeđenoj klasi skrivaju istoimene funkcije osnovne klase. To pravilo podjednako vrijedi i za operatorske funkcije.

Primjerice, ako u osnovnoj klasi imamo operator koji usporeðuje objekt klase s cijelim brojem te ga u izvedenoj klasi redefiniramo kao operator za usporedbu klase sa znakovnom nizom, sljedeæi programski odsjeèak æe prouzroèiti pogrešku prilikom prevoðenja:

```
class Osnovna {
public:
    // ...
    int operator==(int i);
};

class Izvedena : public Osnovna {
public:
    // ...
    int operator==(char *niz);
};

Izvedena obj;

if (obj == "ab") cout << "Jednako!" << endl;    // OK
if (obj == 1) cout << "Podjednako!" << endl;    // pogreška
```

Prva usporedba se može prevesti jer u klasi `Izvedena` postoji operator koji može usporediti objekt sa znakovnim nizom. Druga usporedba, naprotiv, nije ispravna jer je operator usporedbe definiran u osnovnoj klasi, te ga operator u izvedenoj klasi skriva – nasljeðivanje nije isto što i preoptereæenje. Ako želimo u klasi `Izvedena` ostaviti moguænost usporedbe sa cijelim brojevima te dodati još usporedbu sa znakovnim nizovima, potrebno je ponoviti definiciju operatora `operator==(int)` i u izvedenoj klasi:

```
class Izvedena : public Osnovna {
    // ...
public:
    int operator==(char *niz);
    int operator==(int i) {
        return Osnovna::operator==(i);
    }
};
```

Pravilo koje se takoðer vrlo èesto zaboravlja jest da se operator pridruživanja ne nasljeðuje.

Svaka izvedena klasa mora definirati svoj operator pridruživanja. Jasna je i svrha tog pravila: operator pridruživanja mora inicijalizirati cijeli objekt, a ne samo njegov dio. Operator pridruživanja osnovne klase može inicijalizirati samo dio objekta koji je naslijeðen, a dio objekta koji je dodan ostat æe neinicijaliziran. No pogledajmo što æe se dogoditi u sluèaju da pokušamo prevesti sljedeæi dio koda:

```
class Osnovna {
public:
    // ...
```

```
    Osnovna& operator=(int a);
};

class Izvedena : public Osnovna {
public:
    Izvedena();
    // ...
};

Izvedena izv;
izv = 2; // pogreška prilikom prevođenja
```

Osnovna klasa definira operator kojim je moguæe nekom objektu pridružiti cijeli broj. Izvedena klasa ne definira operator pridruživanja, no definira konstruktor. Kako se operator pridruživanja ne nasljeðuje, klasa `Izvedena` neæe imati operator pridruživanja te æe se primjenjivati podrazumijevani operator koji æe provesti kopiranje objekata jedan u drugi, bit po bit. No da bi se to moglo provesti, objekti moraju biti istog tipa. Prevoditelj æe zbog toga pokušati konvertirati cijeli broj `2` u objekt klase `Izvedena` te æe pronaæi konstruktor s jednim parametrom koji æe obaviti konverziju. Takav konstruktor ne postoji te æe se prijaviti pogreška.

Ako bi kojim sluèajem postojao konstruktor s jednim cjelobrojnim parametrom, operacija dodjeljivanja bi se prešutno i bez upozorenja provela kao kopiranje bit po bit privremenog objekta nastalog kao rezultat konverzije.

U sluèaju korištenja dinamièke dodjele memorije unutar klase `Izvedena` ovakvo pridruživanje æe vrlo vjerojatno rezultirati pogreškom prilikom izvoðenja programa. Da bi se takve situacije izbjegle, potrebno je biti vrlo oprezan prilikom nasljeðivanja klasa s definiranim operatorom pridruživanja.

Također, zbog toga što se konstruktori ne nasljeđuju, konverzije konstruktorom također neće biti naslijeđene. Ako je neka konverzija izričito potrebna, neophodno je u izvedenoj klasi ponoviti konstruktor koji će ju obaviti. Na primjer, klasa `Kompleksni` sadržava konstruktor kojim se realni brojevi mogu konvertirati u kompleksne. Ako bismo napravili klasu `LKompleksni` koja bi nasljeđivala i klasu `Kompleksni` i klasu `Atom` te bi opisivala kompleksni broj u vezanoj listi, konstruktor konverzije bi trebalo ponoviti:

```
class LKompleksni : public Kompleksni, public Atom {
public:
    LKompleksni(double a = 0, double b = 0) :
        Kompleksni(a, b) {}
    // ...
};
```

Kako se konstruktori ne nasljeđuju, ne nasljeđuju se niti konverzije konstruktorom.

Nasljeðivanje operatora `new` i `delete` može prouzroèiti velike probleme u radu programa ako programer nije posebno pažljiv. Naime, ti operatori kao jedan od argumenata imaju parametar tipa `size_t` koji oznaèava koliko se memorijsko podruèje zauzima. Ako implementacija operatora ignorira tu vrijednost te primjerice uvijek dodjeljuje memorijsko podruèje velièine klase u kojoj je operator definiran, tada taj operator neæe funkcionirati ispravno za objekte izvedene klase. Objekti izvedenih klasa mogu imati dodatne podatkovne èlanove te æe zbog toga za njih biti potrebno zauzeti veæe memorijsko podruèje. Operator `new` koji uvijek zauzima fiksni komad memorije neæe zauzeti potrebnu memoriju te æe dio objekta biti smješten na memorijskom podruèju koje mu nije dodijeljeno. Takav program æe vjerojatno pogrešno funkcionirati.

Slično je i s operatorom `delete`; on mora osloboditi točnu količinu memorije koja je određena parametrom tipa `size_t`, u suprotnom se može dogoditi da se neka memorija nikada ne oslobodi te se količina slobodne memorije postupno smanjuje. Program će funkcionirati sve dok se ne zauzme cjelokupna memorija, a tada će jednostavno prijaviti nedovoljno slobodne memorije.

11.11. Principi polimorfizma

Do sada smo upoznali dvije osnovne znaèajke koje svrstavaju C++ jezik u objektno orijentiranu domenu. Prvo, to je enkapsulacija: svojstvo koje oznaèava objedinjavanje podataka i funkcija koje podacima manipuliraju. Umjesto da baratamo s pasivnim strukturama podataka i aktivnim funkcijama, program se sastoji od objekata koji meðusobno stupaju u interakciju preko njihovog javnog suèelja.

Sljedeće važno svojstvo objektno orijentiranih jezika je mogućnost stvaranja hijerarhije klasa. Objekti se mogu izvoditi postepeno tako da se prvo definiraju opća svojstva objekta koja se zatim detaljno opisuju u naslijeđenim klasama. To je očito na primjeru klasa koje opisuju grafičke objekte: klasa `GrafObjekt` definira opća svojstva grafičkih objekata koja posjeduje svaki objekt, dok klase `Linija` i `Poligon` dodaju svojstva koja su specifična za poligone i linije.

Polimorfizam je treće važno svojstvo koje svaki ozbiljniji objektno orijentirani jezik mora podržavati. Ono omogućava definiranje operacija koje su ovisne o tipu. U prethodnim poglavljima opisano je kako se svaki objekt izvedene klase može promatrati i kao objekt bilo koje javne osnovne klase. Time se ne narušava integritet objekta jer su svi članovi osnovne klase prisutni i u objektima izvedene klase pa im se bez problema može i pristupiti. Postoje standardne konverzije opisane u prethodnim poglavljima koje pretvaraju pokazivače, reference i objekte izvedenih klasa u pokazivače, reference i objekte osnovnih klasa.

Važno je uočiti da se, iako je možda pokazivač na objekt klase `Linija` pretvoren u pokazivač na objekt klase `GrafObjekt`, sam objekt na koji se pokazuje nije promijenio.

Operacija crtanja primijenjena na taj objekt bi uvijek trebala rezultirati crtanjem linije. No klasa `GrafObjekt` sadrži funkcijski član `Crtaj()` koji ništa ne radi, zato jer ona definira samo opća svojstva svih grafičkih objekata. Izneseni problem se može ilustrirati sljedećim programskim odsječkom:

```
class GrafObjekt {
public:
    // ...
    void Crtaj() {}
};

class Linija : public GrafObjekt {
public:
    // ...
    void Crtaj();       // ovaj funkcijski član će stvarno
                        // nacrtati liniju
};

Linija *pokLinija = new Linija;
GrafObjekt *pokGO = pokLinija;
```

Ako se pozove funkcijski èlan `Crtaj()` preko pokazivaèa `pokLinija`, prevoditelj æe pozvati funkcijski èlan `Crtaj()` iz klase `Linija` zato jer je `pokLinija` definiran kao pokazivaè na objekt klase `Linija`:

```
pokLinija->Crtaj();     // poziva Linija::Crtaj()
```

Pokazivaè `pokGO` je inicijaliziran tako što mu je dodijeljena vrijednost pokazivaèa `pokLinija`. No on i dalje u biti pokazuje na objekt `Linija`, te bi operacija crtanja i dalje trebala nacrtati crtu. Meðutim, poziv

```
pokGO->Crtaj();         // poziva GrafObjekt::Crtaj()
```

æe pozvati funkcijski èlan `Crtaj()` iz klase `GrafObjekt`, zato jer pokazivaè `pokGO` pokazuje na objekt klase `GrafObjekt`. Ovime operacija crtanja, umjesto da je vezana za objekt, postaje ovisna o naèinu poziva. Takvo ponašanje znatno narušava objektno orijentirani pristup programiranju.

Na prvi pogled se gornje ponašanje može činiti logičnim: korisnik mora naznačiti kako želi gledati na objekt. U prvom slučaju korisnik gleda na objekt kao na liniju, pa se stoga i poziva operacija iz klase `Linija`. U drugom slučaju korisnik želi dotični objekt promatrati kao podobjekt početnog objekta pa se zbog toga poziva pripadna funkcija crtanja. Ovakvo ponašanje je u suprotnosti s konceptima objektno orijentiranog programiranja. Svaki objekt ima operacije koje se uvijek izvode na isti način, što odgovara stvarnom ponašanju objekata u prirodi.

Na primjer, mogli bismo sve ljude klasificirati u hijerarhiju klasa koja počinje s podjelom ljudi po boji kože. Svaki (razuman) čovjek će odgovoriti na pitanje “Kako se

zoveš?” na jednak način, bez obzira na to da li na njega gledamo kao na crnca, pripadnika plemena Zulu ili kao na točno određenog pojedinca. Odgovor na to pitanje nije određen okolnim uvjetima, nego samim objektom.

Navedeno ponašanje dolazi do izražaja u slučaju da bismo željeli sve grafičke objekte nekog programa čuvati u jednom polju. Ono bi se sastojalo od pokazivača na tip `GrafObjekt` koji bi po potrebi pokazivao na objekte izvedenih klasa. Crtanje svih objekata se tada može jednostavno provesti na sljedeći način:

```
int const MAX_ELT = 10;
GrafObjekt *nizObj[MAX_ELT];

// ...
nizObj[0] = new Linija;
nizObj[1] = new Poligon;
nizObj[2] = new ElipsinIsj;
// na isti način se mogu popuniti i preostali članovi polja
// ...

// ovo bi bio praktičan način crtanja svih članova polja
for (int i = 0; i <= MAX_ELT; i++)
    nizObj[i]->Crtaj();
```

Svaki objekt toèno zna kako treba obaviti pojedinu radnju te naèin njenog izvoðenja sada ne ovisi o naèinu gledanja na objekt. No ovaj primjer, kako je napisan, sam za sebe ne radi. Kako je svaki element polja `nizObj` pokazivaè na `GrafObjekt`, operacija crtanja pozivat æe `GrafObjekt::Crtaj()`. Ovakvo odreðivanje funkcijskog èlana pomoæu tipova dostupnih prilikom prevoðenja zove se *statièko povezivanje* (engl. *static binding*).

Promotrimo kako bi se izneseni problem mogao riješiti. Osnovna poteškoća je u tome što prevoditelj prilikom prevođenja ima dostupnu samo informaciju o tipu pokazivača, odnosno o načinu gledanja. Taj pokazivač može pokazivati na razne objekte, te bismo htjeli u pojedinim situacijama pozivati odgovarajuće funkcijske članove različitih klasa. Očito se prilikom prevođenja ne može odrediti koja se funkcija poziva, nego se odluka o samom pozivu mora odgoditi za trenutak izvođenja programa. Zajedno s objektom, u memoriju mora biti pohranjen podatak o stvarnom tipu objekta. Prilikom prevođenja poziva funkcije prevoditelj mora stvoriti kôd koji će pročitati tu informaciju te na osnovu nje odrediti koja se funkcija u poziva. Takvo određivanje pozvanih članova naziva se u objektno orijentiranim jezicima *kasno povezivanje* (engl. *late binding*), a u C++ terminologiji uvriježen je naziv *dinamičko povezivanje* (engl. *dynamic binding*).

Slično ponašanje može se simulirati dodavanjem klasi `GrafObjekt` cjelobrojnog člana koji će označavati tip objekta. Prilikom poziva funkcijskog člana potrebno je, ovisno o vrijednosti tog člana, pretvoriti pokazivač u pokazivač na neki drugi tip:

```
enum TipObj {GRAFOBJEKT, LINIJA, POLIGON, PRAVOKUTNIK,
             ELIPSINISJ, KRUG};
```

```
class GrafObjekt {
    // ...
public:
    TipObj tip;
    void Crtaj() {}
};

// deklaracije ostalih klasa se ne mijenjaju

int const MAX_ELT = 10;
GrafObjekt *nizObj[MAX_ELT];

// niz nizObj se popunjava po istom principu

// crtanje sada izgleda ovako:

for (int i = 0; i < MAX_ELT; i++)
    switch (nizObj[i]->tip) {
        case GRAFOBJEKT:
            nizObj[i]->Crtaj();
            break;
        case LINIJA:
            ((Linija*)(nizObj[i]))->Crtaj();
            break;
        case POLIGON:
            ((Poligon*)(nizObj[i]))->Crtaj();
            break;
        // po istom principu se navode svi ostali tipovi
    }
```

Predloženo rješenje unatoè ispravnom funkcioniranju ima niz nedostataka. Oèito je da je na svakom mjestu u programu na kojem želimo pozvati neki funkcijski èlan potrebno ispisati cijeli `switch` blok, što može biti vrlo nepraktièno i zamorno. Takoðer, ako se kasnije doda novi grafièki objekt, potrebno je proæi kroz cijeli program te dodati nove instrukcije za pozivanje. Ponekad to može biti i nemoguæe, u sluèaju da izvorni kôd programa nije dostupan (na primjer, ako se radi proširenje neke postojeæe komercijalno dostupne biblioteke klasa).

11.11.1. Virtualni funkcijski èlanovi

C++ nudi vrlo elegantno i uèinkovito rješenje gore navedenog problema u obliku virtualnih funkcijskih èlanova. Osnovna ideja je u tome da se funkcijski èlanovi za koje želimo dinamièko povezivanje oznaèe prilikom deklaracije klase. To se èini navoðenjem kljuène rijeèi `virtual`, a takvi èlanovi se nazivaju *virtualnim funkcijskim èlanovima* (engl. *virtual function members*). Prevoditelj automatski održava tablicu virtualnih èlanova koja se pohranjuje u memoriju zajedno sa samim objektom. Prilikom poziva èlana, prevoditelj æe potražiti adresu èlana u tablici koja je privezana uz objekt te na taj naèin pozvati èlan.

Klasu `GrafObjekt` ćemo modificirati tako da funkcijske članove koji moraju biti povezani uz sam objekt označimo virtualnima:

```
class GrafObjekt {
private:
    int Boja;
public:
    void PostaviBoju(int nova) { Boja = nova; }
    int CitajBoju() { return Boja; }
    virtual void Crtaj() {}
    virtual void Translatiraj(int, int) {}
    virtual void Rotiraj(int, int, int) {}
};

class Linija : public GrafObjekt {
private:
    int x1, y1, x2, y2;
public:
    Linija(int lx1, int ly1, int lx2, int ly2) :
        x1(lx1), y1(ly1), x2(lx2), y2(ly2) {}
    virtual void Crtaj();
    virtual void Translatiraj(int vx, int vy);
    virtual void Rotiraj(int cx, int cy, int kut);
};
```

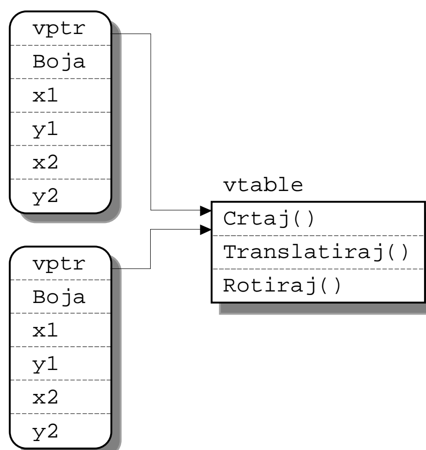
Funkcijski èlanovi `Crtaj()`, `Translatiraj()` i `Rotiraj()` deklarirani su kao virtualni.

Virtualni èlan se deklarira tako da se ispred povratnog tipa èlana navede ključna rijeè `virtual`. Prilikom definicije funkcije izvan klase ta se ključna rijeè ne smije ponoviti.

Na primjer, definicija èlana `Crtaj()` izgledala bi ovako:

```
void Linija::Crtaj() {
    // virtual se ne ponavlja u definiciji
    // ...
}
```

Prevoditelj æe sve virtualne funkcijske èlanove pohraniti u tablicu virtualnih èlanova. Ona se u literaturi èesto oznaèava skraæenicom `vtable`. Toj tablici se ne može direktno pristupiti. Svaki objekt sadržavat æe skriveni pokazivaè `vptr` na virtualnu tablicu. Takav poziv se zove *dinamièki poziv* (engl. *dynamic call*) ili *virtualni poziv* (engl. *virtual call*). To se može prikazati kao na slici 11.7.



Slika 11.7. Mehanizam virtualnih funkcija

Prilikom poziva funkcijskog èlana prevoditelj æe generirati kôd koji æe pomoæu pokazivaèa `vptr` pronaæi tablicu virtualnih funkcija, a zatim æe u njoj pronaæi toènu adresu funkcije koju treba pozvati. Vidi se da je odluka o pozivu odgoðena za trenutak izvoðenja programa te rezultat ovisi o stvarnom objektu za koji se èlan poziva.

Osim konstruktora, sve ostale funkcijske članove možemo učiniti i virtualnima.

Može se postaviti pitanje zašto nisu svi èlanovi podrazumijevano virtualni tako da programer ne mora voditi raèuna o virtualnosti èlanova. Razlog tome leži u efikasnosti prilikom izvoðenja. Poziv virtualnog funkcijskog èlana troši nešto više procesorskog vremena jer je prije poziva potrebno pogledati `vtable`. Takoðer, sama tablica virtualnih funkcija zauzima izvjestan memorijski prostor.

Članove `PostaviBoju()` i `CitajBoju()` ćemo ostaviti nevirtualnima, te će se pozvani funkcijski član odrediti prilikom prevođenja. Postavljanje i čitanje boje su operacije za koje se očekuje da se neće mijenjati prilikom nasljeđivanja, odnosno da će svi objekti na isti način postavljati i čitati svoju boju. Ti su članovi ostavljeni nevirtualnima kako bi se uštedjelo na veličini virtualne tablice i dobilo na brzini izvođenja programa.

Za nevirtualne članove se kaže da se *pozivaju statički* (engl. *static call*). To nipošto ne treba miješati sa statičkim članovima klasa deklariranih ključnom riječi `static`. Ovdje se primarno misli na to da se određivanje člana u pozivu provodi statički, na osnovu podataka dostupnih prilikom prevođenja, za razliku od virtualnih poziva koji se određuju dinamički, na osnovu podataka dostupnih prilikom izvođenja. Pravi statički

članovi nisu vezani za objekt pa niti ne mogu biti virtualni. Da bi se izbjegle zabune i ružan termin *nevirtualni*, takve članove ćemo zvati *članovi sa statičkim pozivom*.

Članovi sa statičkim pozivom osnovne klase mogu biti redeklarirani kao virtualni u izvedenoj klasi. Određivanje da li se član poziva virtualno ili statički u tom se slučaju odnosi na klasu iz koje se poziva član, na primjer:

```
#include <iostream.h>

class A {
public:
    void func() { cout << "A::func()" << endl; }
};

class B : public A {
public:
    virtual void func() { cout << "B::func()" << endl; }
};

class C : public B {
public:
    virtual void func() { cout << "C::func()" << endl; }
};

int main() {
    C objc;
    A *pokA = &objc;
    B *pokB = &objc;
    pokA->func();       // ispisuje A::func()
    pokB->func();       // ispisuje C::func()
    return 0;
}
```

Prvi poziv rezultira pozivom funkcije `A::func()` zato jer se funkcija poziva preko pokazivaèa na klasu `A`. U toj klasi je `func()` definiran bez kljuène rijeèi `virtual` pa se on poziva statièki, a to rezultira pozivom iz klase `A`. U drugom sluèaju èlan se poziva preko pokazivaèa na klasu `B`. U toj klasi èlan je redeklariran kao virtualni, pa se on

Kada je neki funkcijski član deklariran kao virtualan, on je automatski virtualan i u svim naslijeđenim klasama.

poziva pomoæu virtualne tablice. To rezultira pozivom funkcijskog èlana samog objekta, a ne èlana klase `B`. Zato se u stvari poziva èlan `C::func()`.

Ako bi se ispred deklaracije èlana `func()` u klasi `C` i izostavila kljuèna rijeè `virtual`, funkcija bi bila virtualna jer je veæ uèinjena virtualnom u klasi `B`. Funkcijski èlan koji je jednom deklariran virtualnim, ne može se u izvedenoj klasi pretvoriti u èlan sa statièkim pozivom.

11.11.2. Poziv virtualnih funkcijskih èlanova

Virtualni funkcijski èlanovi mogu se pozvati pomoæu objekta klase, preko pokazivaèa ili reference na neki objekt te pozivom unutar klase. C++ jezik definira skup pravila pomoæu kojih se može odrediti kako æe se pozvati neki funkcijski èlan.

Modificirat ćemo hijerarhiju grafičkih objekata tako što ćemo dodati mogućnost brisanja objekta s ekrana te mogućnost premještanja objekta s jednog mjesta na drugo. Kako ne postoje funkcije za crtanje koje bi se na isti način izvodile na svim računalnim platformama, radi općenitosti kôda stavit ćemo da pojedini funkcijski član umjesto stvarnog crtanja na ekran ispisuje poruku o tome koja je funkcija pozvana. Valja uočiti da se postupak premještanja objekta sastoji iz tri osnovne operacije: brisanja objekta s ekrana, promjene koordinata te njegovog ponovnog crtanja. Svaki objekt mora definirati kako se za njega obavlja pojedini od tih koraka, odnosno kako će se objekt nacrtati, kako će se izbrisati te kako će mu se promijeniti koordinate. Operaciju crtanja obavit će funkcijski član `Crtaj()`, operaciju brisanja član `Brisi()`, a operaciju pomaka član `Translatiraj()`. Funkcijski član koji će objedinjavati te tri operacije nazvat ćemo `Pomakni()`, a kao parametre imat će dva broja koji će definirati vektor za koji se pomak obavlja. Taj član će se definirati samo jednom te će pomoću mehanizma virtualnih funkcija pozivati ispravnu funkciju za brisanje, pomak i crtanje.

```
class GrafObjekt {
private:
    int Boja;
public:
    void PostaviBoju(int nova) { Boja = nova; }
    int CitajBoju() { return Boja; }
    virtual void Crtaj() {}
    virtual void Brisi() {}
    virtual void Translatiraj(int, int) {}
    void Pomakni(int px, int py);
};

void GrafObjekt::Pomakni(int px, int py) {
    Brisi();
    Translatiraj(px, py);
    Crtaj();
}
```

Iz gornjeg primjera je vidljivo da se pozivi funkcijskih èlanova iz drugih èlanova klase tretiraju kao virtualni. Takoðer, poziv preko pokazivaèa ili reference na objekt bit æe virtualan:

```
Linija duzina;
GrafObjekt *go1 = &duzina, &go2 = duzina;

go1->Crtaj();       // virtualan poziv
go2.Crtaj();        // ponovo virtualan poziv
```

Postoje tri sluèaja kada se poziv èlana obavlja statièki iako je èlan deklariran kao virtualan. Prvi takav sluèaj je kada se funkcijski èlan poziva preko objekta neke klase. To je i logièno, jer ako se prilikom prevoðenja toèno zna klasa objekta za koji se èlan poziva, moguæe je odmah toèno odrediti i odgovarajuæi funkcijski èlan. To ne vrijedi za dereferencirane pokazivaèe; za njih se pozivi takoðer provode virtualno. Na primjer:

```
duzina.Crtaj();      // statički poziv
(*go1).Crtaj();      // virtualan poziv
```

Prvi poziv može biti statièki jer se toèno zna o kojoj se klasi radi; identifikator `duzina` jednoznaèno oznaèava objekt klase `Linija`. Drugi poziv je virtualan. Vidi se da je virtualan mehanizam u ovom sluèaju potreban, jer `*go1` oznaèava objekt klase `GrafObjekt`, dok je u memoriji stvarno objekt klase `Linija`.

Mehanizam virtualnih poziva može se zaobići i tako da se operatorom za određivanje područja eksplicitno navede klasa iz koje se želi pozvati funkcijski član. Na primjer:

```
duzina.GrafObjekt::Crtaj();     // statički poziva
                                // GrafObjekt::Crtaj()
go1->GrafObjekt::Crtaj();       // statički poziva
                                // GrafObjekt::Crtaj()
```

Neispravno je pomoæu pokazivaèa na neku klasu pozivati funkcijski èlan iz izvedene klase, na primjer:

```
go1->Linija::Crtaj();           // pogreška
```

Gornji primjer je neispravan zato što prilikom prevoðenja nije poznato na koji objekt pokazivaè `go1` pokazuje. Ako bi `go1` pokazivao na objekt klase `GrafObjekt`, èlan `Linija::Crtaj()` bi mogao pristupiti èlanovima `x1` ili `y1` koji ne postoje u objektu `GrafObjekt`, što bi rezultiralo neispravnim radom programa.

Statički poziv pomoću operatora za određivanje područja odnosi se samo na onaj član koji je eksplicitno naveden u pozivu. Eventualni virtualni pozivi iz tog člana ostaju nepromijenjeni. Na primjer:

```
go1->GrafObjekt::Pomakni(5, 7);
```

Èlan `Pomakni()` se poziva statièki, no èlanovi `Brisi()`, `Translatiraj()` i `Crtaj()` koji se pozivaju iz èlana `Pomakni()` se i dalje pozivaju virtualno.

Treći slučaj u kojem se zaobilazi mehanizam virtualnih poziva je poziv iz konstruktora ili destruktora. Razlog tome je što poziv virtualne funkcije u konstruktoru može rezultirati pristupom još neinicijaliziranom dijelu objekta, dok poziv virtualne funkcije u destruktoru može rezultirati pristupom već uništenom dijelu objekta.

11.11.3. Èiste virtualne funkcije

Funkcijski èlanovi `Crtaj()`, `Translatiraj()` i `Rotiraj()` u klasi `GrafObjekt` nemaju neko stvarno znaèenje. Njihova je jedina zadaæa definiranje javnog suèelja koje æe biti detaljno specificirano tek u izvedenim klasama. Takoðer, nema smisla stvarati objekt klase `GrafObjekt` – ta klasa zapravo služi kao temelj za nasljeðivanje. Takve klase se u C++ terminologiji zovu *apstraktne klase* (engl. *abstract classes*), dok se funkcijski èlanovi koji služe samo za definiciju javnog suèelja nazivaju *èisti virtualni funkcijski èlanovi* (engl. *pure virtual function members*).

Virtualni funkcijski èlan se proglašava èistim virtualnim tako da se iza njegove deklaracije stavi oznaka `=0`. Klasa je apstraktna ako sadrži barem jedan èisti virtualni funkcijski èlan.

Klasa `GrafObjekt` redeklarirana kao apstraktna klasa izgleda ovako:

```
class GrafObjekt {
private:
    int Boja;
public:
    void PostaviBoju(int nova) { Boja = nova; }
    int CitajBoju() { return Boja; }
    virtual void Crtaj() = 0;
    virtual void Brisi() = 0;
    virtual void Translatiraj(int, int) = 0;
    void Pomakni(int px, int py);
};
```

Ovakvom deklaracijom se naznaèava da èlanovi `Crtaj()`, `Brisi()` i `Translatiraj()` nemaju definiciju na nivou klase `GrafObjekt` te æe se njihova definicija dati tek u izvedenim klasama. Time je klasa `GrafObjekt` pretvorena u apstraktnu klasu. Više nije moguæe deklarirati objekt te klase – svaki takav pokušaj rezultirat æe pogreškom prilikom prevoðenja.

Nije moguće deklarirati objekte apstraktnih klasa. Moguće je, naprotiv, deklarirati pokazivače i reference na te klase.

Klasa koja nasljeðuje apstraktnu klasu nasljeðuje i sve èiste virtualne funkcije. Ona ih može definirati, èime se zapravo daje smisao javnom suèelju danom u osnovnoj klasi. Ako se èiste virtualne funkcije ne redefiniraju u izvedenoj klasi, one automatski postaju dio izvedene klase pa i izvedena klasa postaje apstraktna.

11.11.4. Virtualni destruktori

Poput svih drugih funkcijskih èlanova, i destruktor može biti virtualan. Razmotrimo razloge zbog kojih bi nam to moglo biti od koristi.

Destruktor se poziva prilikom brisanja objekta iz memorije računala operatorom `delete`. U sljedećem primjeru stvorit ćemo polje pokazivača na tri različita grafička objekta:

```
int const MAX_ELT = 3;
GrafObjekt *nizObj[MAX_ELT];
nizObj[0] = new Linija;
nizObj[1] = new Poligon;
nizObj[2] = new ElipsinIsj;
```

Polje sadrži pokazivaèe na objekte klase `GrafObjekt`. Za pohranjivanje pojedinih objekata u polje koriste se principi polimorfizma: pokazivaèi na pojedine objekte se konvertiraju u pokazivaèe na osnovne javne klase. Mehanizam virtualnih funkcija osigurava da poziv

```
nizObj[1]->Crtaj();
```

zaista i rezultira crtanjem poligona. Funkcijski èlan `Crtaj()` je deklariran kao virtualan, pa iako se u pozivu koristi pokazivaè na osnovnu klasu, poziva se u stvari funkcijski èlan iz odgovarajuæe izvedene klase. Meðutim, promotrimo što se dogaða prilikom brisanja niza operatorom `delete`:

```
for (int i = 0; i <= MAX_ELT; i++)
    delete nizObj[i];
```

Operator `delete` poziva destruktor za objekt na koji pokazuje pokazivaè. No pokazivaè pokazuje na objekt klase `GrafObjekt`, a destruktor je funkcijski èlan sa statièkim pozivom. Takvi se èlanovi pozivaju iz klase èijeg je tipa pokazivaè. To znaèi da bi u ovom sluèaju došlo do poziva destruktora za klasu `GrafObjekt`, umjesto destruktora za odgovarajuæe objekte na koje pokazivaèi iz niza stvarno pokazuju.

Taj problem se može riješiti tako da se destruktori svih klasa u hijerarhiji učine virtualnima. Na primjer:

```
class GrafObjekt {
// ...
public:
    virtual ~GrafObjekt();
};
```

Iako destruktori nemaju zajednièko ime, oni mogu biti virtualni.

Sve klase koje nasljeðuju klasu s virtualnim destruktorom takoðer æe imati virtualne destruktore te nije to potrebno eksplicitno specificirati kljuènom rijeèi `virtual`.

Dakle, nakon gornje promjene u osnovnoj klasi sve izvedene klase æe automatski imati virtualne destruktore. No kako bi se programski kôd uèinio èitljivijim, dobra je praksa i

u izvedenim klasama dodati kljuèni rijeè `virtual` ispred deklaracije destruktora. Tada je svakom odmah jasno kako poziva li se destruktor virtualno ili ne, bez nepotrebnog gledanja u osnovnu klasu.

Destruktori apstraktnih klasa se u pravilu navode kao virtualni. Razlog tome je što objekt virtualne klase ne može biti deklariran, pa niti destruktor sa statičkim pozivom za takav objekt nema smisla.

Iz izloženoga se vidi da je pozivanje ispravnog destruktora željeno ponašanje za svaki objekt. Teško je zamisliti situaciju u kojoj bi pozivanje neispravnog destruktora moglo biti željeno ponašanje. Postavlja se logično pitanje zašto onda uopće postoje destruktori sa statičkim pozivom. Odgovor je praktične naravi: mehanizam virtualnih funkcija proširuje svaki objekt s `vptr` pokazivačem na tablicu virtualnih funkcija. Također, za svaku klasu je potrebno zauzeti u memoriji tablicu virtualnih funkcija. U mnogim slučajevima, naprotiv, klasa neće uopće biti naslijeđena te neće niti doći do opasnosti od poziva neodgovarajućeg destruktora. Zbog toga bi resursi za smještaj objekata u memoriji bili uzalud potrošeni. C++ jezik pruža programeru na volju da sam optimizira svoj program kako smatra najprikladnijim. Kako programer sije, tako će i žeti.

Zadatak. *Listi opisanoj u odsječku 11.2 dodajte član* `UPotrazi()` *koji će kao parametar imati referencu na neki objekt naslijeđen od klase* `Atom`*. On će pretražiti cijelu listu dok se ne nađe član koji odgovara zadanom. Usporedba će se obavljati pomoću preopterećenog operatora* `==`*. Zbog toga je klasu* `Atom` *potrebno proširiti čistim virtualnim operatorom* `==` *te definirati taj operator u klasi koja nasljeđuje klasu* `Atom`*.*

Zadatak. *Umjesto smještaja grafičkih objekata u polje, prepravite klase grafičkih objekata tako da se objekti mogu smještati u vezanu listu definiranu klasom* `Lista`*. (To znači da* `GrafObjekt` *mora naslijediti klasu* `Atom`*.) Napišite program koji će crtati grafičke objekte iz liste, umjesto iz polja.*

Zadatak. *Napišite funkciju* `Animacija()` *koja kao parametar ima pokazivač na grafički objekt te pomiče taj objekt po ekranu. Funkcija mora raditi ispravno sa svakim objektom, što zapravo znači da će koristiti virtualne pozive funkcija za crtanje i brisanje.*

11.12. Virtualne osnovne klase

Postoje sluèajevi kada se neka osnovna klasa može proslijediti izvedenoj klasi više no jednom. Ilustrirat æemo to na sljedeæem problemu.

Česti element u raznim programima koji barataju grafikom je tekst. U hijerarhiju grafičkih objekata dodat ćemo stoga novu klasu `Tekst` koja će opisivati tekst na bilo kojem dijelu ekrana u bilo kojem tipu pisma.

```
class Tekst : public GrafObjekt {
private:
    char *pokTekst;
```

```
public:
    virtual void PostaviTekst(char *niz);
    virtual void Crtaj();
    virtual void Brisi();
    virtual void Translatiraj(int px, int py);
    virtual ~Tekst();
};
```

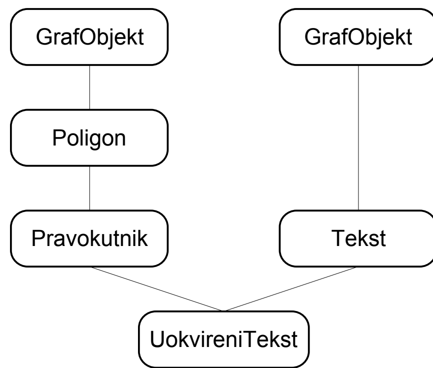
Ponekad je potrebno tekst uokviriti pravokutnikom. Bilo bi vrlo praktièno stvoriti novu klasu `UokvireniTekst` koja bi definirala upravo takav objekt. Takav objekt je neka vrsta križanca izmeðu objekta `Tekst` i objekta `Pravokutnik`; on jest u isto vrijeme i tekst i pravokutnik. Na njega ima smisla primijeniti i operacije koje se primjenjuju na tekst, ali i operacije koje se primjenjuju na pravokutnike. Jedno od moguæih rješenja je iskoristiti mehanizme višestrukog nasljeðivanja:

```
class UokvireniTekst : public Pravokutnik, public Tekst {
    // ...
};
```

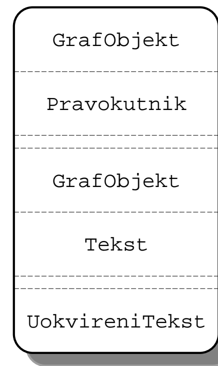
Ovo rješenje je vrlo elegantno, no pogledajmo što smo zapravo dobili. Klase `Pravokutnik` i `Tekst` nasljeðuju od klase `GrafObjekt`, što znaèi da sadržavaju po jedan podobjekt `GrafObjekt`. Rezultirajuæi objekt `UokvireniTekst` imat æe dva objekta klase `GrafObjekt`: jedan dobiven preko klase `Pravokutnik` i jedan dobiven preko klase `Tekst`. Hijerarhija nasljeðivanja prikazana je na slici **11.8**. Na slici 11.9 se vidi struktura samog objekta: postoje dva podobjekta klase `GrafObjekt`.

Ovakva struktura objekta nije poželjna. Dobiveni objekt ima niz nedostataka. Na primjer, nije moguća ugrađena konverzija u pokazivač na `GrafObjekt`. To je razumljivo, jer nije jasno u koji se `GrafObjekt` treba konverzija obaviti. Nadalje, svi članovi klase `GrafObjekt` prisutni su dva puta te prilikom pristupa nije jasno kojem se članu pristupa:

```
UokvireniTekst ut;
ut.PostaviBoju(CRVENA);     // pogreška: nejasan pristup
```



Slika 11.8. Višestruko nasljeđivanje



Slika 11.9. Struktura objekta klase `UokvireniTekst`

Èlan `PostaviBoju()` postoji u oba podobjekta `GrafObjekt` tako da u gornjem primjeru nije jasno kojem se èlanu zapravo pristupa. Da bi se toèno odredilo kojem se èlanu pristupa, potrebno je koristiti operator za odreðivanje podruèja. No poziv

```
ut.GrafObjekt::PostaviBoju(CRVENA); // pogreška: nejasno
```

nije nimalo jasniji: klasa `GrafObjekt` postoji u klasi `UokvireniTekst` dva puta pa je navedeni pristup podjednako nejasan. Za pristup tom èlanu je potrebno toèno navesti kojem se od tih dvaju èlanova pristupa: onom naslijeðenom od klase `Pravokutnik` ili onom naslijeðenom od klase `Tekst`:

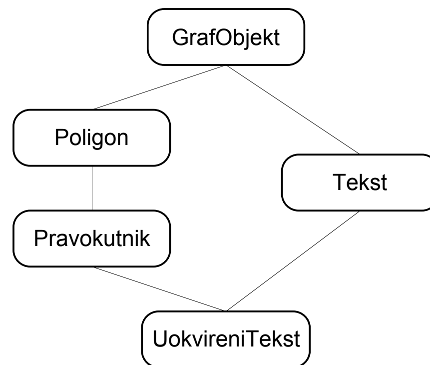
```
ut.Pravokutnik::PostaviBoju(BIJELA);
ut.Tekst::PostaviBoju(ZELENA);
```

U nekim primjenama pojavljivanje osnovne klase više puta u toku nasljeðivanja može biti poželjno. U ovom sluèaju to je oèito pogrešno. Klasa `GrafObjekt` predstavlja osnovnu klasu za sve grafièke objekte u hijerarhiji. Kako je `UokvireniTekst` u cjelini samo grafièki objekt, bilo bi logièno da ta klasa ima klasu `GrafObjekt` proslijeðenu samo jednom. Tako oblikovano hijerarhijsko stablo izgledalo bi kao na slici 11.10.

Mehanizam *virtualnih osnovnih klasa* (engl. *virtual base classes*) omogućava definiranje osnovnih klasa koje se dijele između više izvedenih klase te se prilikom nasljeđivanja izvedenoj klasi prosljeđuju samo jednom.

11.12.1. Deklaracija virtualnih osnovnih klasa

Neka osnovna klasa može se uèiniti osnovnom virtualnom klasom tako da se prilikom nasljeðivanja u listi nasljeðivanja ispred naziva klase umetne kljuèna rijeè `virtual`. Redoslijed kljuènih rijeèi `public`, `private`, `protected` i `virtual` pritom nije bitan. Klasa se ne definira virtualnom prilikom njene deklaracije, nego prilikom nasljeðivanja.



Slika 11.10. Primjena virtualne osnovne klase

To znaèi da se deklaracija klase `GrafObjekt` ne mijenja. Štoviše, ona može jednom biti upotrijebljena kao virtualna, a drugi put kao nevirtualna:

```
class Poligon : public virtual GrafObjekt {
    // ...
};

class Pravokutnik : public Poligon {
    // ...
};

class Tekst : public virtual GrafObjekt {
    // ...
};

class UokvireniTekst : public Pravokutnik, public Tekst {
    // ...
};
```

Kako su klase `Poligon` (a preko nje i klasa `Pravokutnik`) i `Tekst` definirale klasu `GrafObjekt` kao virtualnu osnovnu klasu, klasa `UokvireniTekst` æe sada zaista imati samo jedan podobjekt klase `GrafObjekt`.

Mehanizam virtualnih klasa omogućava nam da riješimo naš problem na zadovoljavajući način, a to je da definiramo jednu klasu zajedničku svim preostalim klasama. No takvo nasljeđivanje uvodi novi niz pravila koja se moraju poštivati da bi se mehanizam mogao uspješno primjenjivati.

11.12.2. Pristup èlanovima virtualnih osnovnih klasa

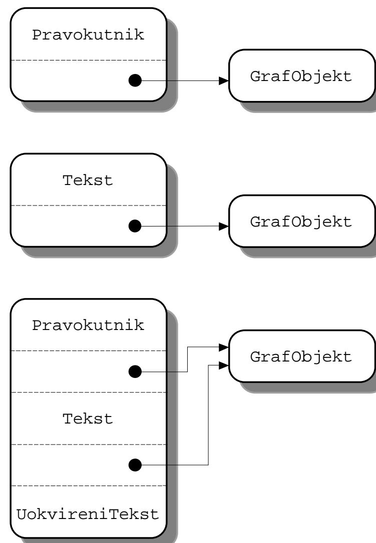
Èlanovima virtualnih osnovnih klasa može se pristupiti bez opasnosti od nejasnoæa prilikom pristupa, jer sada postoji samo jedan podobjekt unutar izvedenog objekta. To se postiže razlikom u alokaciji osnovnih virtualnih klasa. Dijelovi objekta izvedene

klase koji potièu od osnovnih virtualnih klasa nisu sadržani u objektu izvedene klase. Oni se èuvaju u posebnom bloku memorije, a skriveni pokazivaè izvedene klase pokazuje na podobjekt osnovne virtualne klase, kako je prikazano na slici 11.11. Prilikom izvoðenja oba izvedena objekta æe jednostavno pokazivati na isti podobjekt.

Na elemente virtualnih osnovnih klasa primjenjuju se pravila pristupa i tipova izvođenja kao i za obične nevirtualne klase: članovi naslijeđeni po javnom načinu nasljeđivanja zadržavaju svoje originalno pravo pristupa, dok članovi naslijeđeni po privatnom i zaštićenom načinu dobivaju privatno ili zaštićeno pravo pristupa.

No što se događa u slučajevima kada je osnovna virtualna klasa naslijeđena jednom kao javna, a jednom kao privatna? Na primjer:

```
class Poligon : private virtual GrafObjekt {
    // ...
};

class Pravokutnik : public Poligon {
    // ...
};

class Tekst : public virtual GrafObjekt {
    // ...
};

class UokvireniTekst : public Pravokutnik, public Tekst {
    // ...
```



Slika 11.11. Struktura objekata s virtualnim osnovnim klasama

```
};
```

Sada se funkcijski èlan `PostaviBoju()` nalazi na dvije staze nasljeðivanja: jednom kao privatni èlan naslijeðen preko klase `Pravokutnik`, a jednom kao javni èlan naslijeðen preko klase `Tekst`. Pravilo kaže da javni put do nekog èlana uvijek prevladava, odnosno da æe u klasi `UokvireniTekst` taj èlan imati javni pristup. Stoga æe sljedeæi poziv biti ispravan:

```
UokvireniTekst ut;
ut.PostaviBoju(CRNA);   // OK
```

Posebno pravilo *dominacije* (engl. *dominance*) se primjenjuje u situacijama kada u izvedenim klasama postoje èlanovi koji imaju iste nazive kao i èlanovi osnovnih klasa. Tada se prilikom pristupa èlanu èesto može postaviti pitanje kojem se èlanu zapravo pristupa. Na primjer, možemo u klasi `Tekst` definirati novi funkcijski èlan `PostaviBoju()` koji æe obaviti zadatak postavljanje boje na naèin specifièan za tu klasu. Deklaracije tada izgledaju ovako:

```
class Tekst : public virtual GrafObjekt {
public:
    void PostaviBoju(int b);
    // ...
};
```

Sada u klasi `UokvireniTekst` postoje dva funkcijska èlana za postavljanje boje: jedan iz klase `GrafObjekt` dobiven preko klase `Pravokutnik` i drugi definiran u klasi `Tekst`. Ako se prilikom izvoðenja klase `Tekst` ne bi koristilo virtualno nasljeðivanje, prevoditelj ne bi mogao razlikovati ta dva èlana, odnosno prilikom prevoðenja bi dojavio pogrešku. No ako èlan koji se poziva pripada virtualnoj osnovnoj klasi i nekoj klasi koja se nalazi bliže u hijerarhijskom lancu, prevoditelj æe izabrati ovaj potonji. Kaže se da èlanovi iz bližih klasa *dominiraju* nad èlanovima osnovnih virtualnih klasa. Na primjer:

```
UokvireniTekst ut;
ut.PostaviBoju(BIJELA); // poziva Tekst::PostaviBoju(int)
```

11.12.3. Inicijalizacija osnovnih virtualnih klasa

U obiènim sluèajevima nasljeðivanja svaki konstruktor u svojoj inicijalizacijskoj listi može inicijalizirati samo neposredno prethodeæe klase. Na primjer, ako ne bismo koristili virtualne osnovne klase, klasa `Pravokutnik` ne bi mogla inicijalizirati klasu `GrafObjekt`, buduæi da se u stablu nasljeðivanja izmeðu njih nalazi klasa `Poligon`. Virtualne osnovne klase predstavljaju izuzetak u ovom pravilu. Evo i razloga.

Klasa `UokvireniTekst` nasljeđuje od klasa `Tekst` i od klasa `Pravokutnik`. Obje klase inicijaliziraju svoj `GrafObjekt` podobjekt. No objekt klase `UokvireniTekst` sadržava samo jedan `GrafObjekt`. Ako bi se ta klasa inicijalizirala isključivo u neposredno izvedenim klasama, tada bi `Pravokutnik` i `Tekst` pokušali inicijalizirati isti `GrafObjekt` dva puta, vrlo vjerojatno na dva različita načina. Na primjer, uzmimo da klasa `GrafObjekt` sadržava konstruktor kojemu se prosljeđuje početna boja objekta:

```
class GrafObjekt {
public:
    GrafObjekt(int b);
    // ...
};

class Poligon : virtual public GrafObjekt {
public:
    Poligon() : GrafObjekt(BIJELA) {}
    // ...
};

class Tekst : virtual public GrafObjekt {
public:
    Tekst() : GrafObjekt(CRNA) {}
    // ...
};

class Pravokutnik : public Poligon {
    // ...
};

class UokvireniTekst : public Pravokutnik, public Tekst {
public:
    UokvireniTekst() : GrafObjekt(ZELENA) {}
    // ...
};
```

Klasa `Poligon` (pa tako i klasa `Pravokutnik`) inicijalizirat æe klasu `GrafObjekt` na bijelu boju, dok æe klasa `Tekst` postaviti `GrafObjekt` na crno. Kako klasa `UokvireniTekst` sadržava samo jedan `GrafObjekt` podobjekt, on može biti samo jednom inicijaliziran. Zbog toga je uvedeno pravilo koje kaže da se sve virtualne osnovne klase inicijaliziraju konstruktorom èiji je poziv naveden u *najdalje izvedenoj* (engl. *most derived*) klasi. To je u ovom sluèaju klasa `UokvireniTekst`. Pozivi konstruktorima za `GrafObjekt` u inicijalizacijskim listama klasa `Poligon` i `Tekst` se tada ignoriraju, a konstruktori se ne izvode. Kao rezultat, `GrafObjekt` se postavlja na zeleno.

Virtualne osnovne klase se uvijek inicijaliziraju u najdalje izvedenoj klasi. Konstruktori u lancu nasljeđivanja prije najdalje izvedene klase se ne izvode.

Postoje promjene i u redoslijedu izvođenja konstruktora i destruktora pojedinih klasa. Pravilo kaže da se sve virtualne osnovne klase inicijaliziraju uvijek prije nevirtualnih osnovnih klasa, neovisno o njihovoj poziciji u hijerarhijskom stablu ili u listi nasljeđivanja.

Najprije se inicijaliziraju neposredne virtualne klase. Ako postoji nekoliko neposrednih osnovnih virtualnih klasa, one se inicijaliziraju ovisno o redoslijedu u inicijalizacijskoj listi. Potom se inicijaliziraju sve virtualne klase u hijerarhijskom stablu ispod trenutne klase. Na kraju se pozivaju konstruktori za preostale nevirtualne osnovne klase.

Redoslijed pozivanja destruktora je time automatski definiran; on je uvijek obrnut od redoslijeda pozivanja konstruktora.

Moguće su situacije u kojima se jedna klasa prosljeđuje izvedenoj klasi i kao virtualna i kao nevirtualna. U tom slučaju objekt izvedene klase sadržavat će jedan virtualni podobjekt i potreban broj nevirtualnih objekata. No takav pristup programiranju je bolje izbjegavati. Ponovo se javlja problem nemogućnosti razlučivanja imena članova. Ako i sam programer jest u stanju pratiti stablo izvođenja i na ispravan način pozivati pojedine članove, vrlo je velika vjerojatnost da to osoba koja će čitati programski kôd neće moći učiniti.

12. Predlošci funkcija i klasa

Nakon što je Platon definirao čovjeka kao dvonožnu životinju bez perja, Diogen očerupa pijetla te ga donese u Akademiju uz riječi: 'Ovo je Platonov čovjek.' Na osnovu toga definicija bijaše proširena: 'Sa širokim i ravnim noktima'

Diogen iz Sinope, oko 350. p.n.e.

Vrlo korisna moguænost jezika C++ jesu predlošci. Oni omoguæavaju pisanje opæenitog kôda koji vrijedi jednako za razlièite tipove. Time se vrijeme razvoja složenih programa može znaèajno skratiti, a olakšava se i održavanje programa te ispravljanje eventualnih pogrešaka.

Predlošci se dijele u dvije osnovne skupine: predlošci funkcija i predlošci klasa. Prvi omogućavaju pisanje općih funkcija koje se zatim primjenjuju za različite tipove parametara. Potonji omogućavaju definiranje općih klasa koje se zatim aktualiziraju tipovima prilikom poziva. U ovom poglavlju bit će iscrpno objašnjene obje vrste predložaka.

12.1. Uporabna vrijednost predložaka

Ako ste ikad bili u situaciji da morate razviti iole kompleksniji programski paket, tada ste vjerojatno imali prilike primijetiti da postoje algoritmi koji se mogu podjednako primijeniti na više razlièitih tipova podataka. Jedan od tipiènih primjera je sortiranje podataka. Razvijeno je mnogo razlièitih algoritama za sortiranje: *bubble sort*, *quick sort*, *heap sort* i slièni. Svaki od njih koristi određeni princip sortiranja neovisan o tipu podataka koji se sortira. Važno je samo da postoji moguænost usporeðivanja podataka i njihovog premještanja u nizu. Postupak se ne razlikuje sortiramo li nizove znakova, cijele brojeve ili neke korisnièke objekte koji se mogu usporeðivati.

Iako se sam algoritam u suštini ne razlikuje za različite tipove podataka koji se sortiraju, programer do sada nije bio u mogućnosti napisati jednu funkciju koja će definirati opći algoritam sortiranja primjenjiv neovisno o tipu koji se sortira. Problem leži u tomu što je C++ jezik, kao i mnogi drugi srodni jezici (Pascal, Modula 2, Ada) strogo tipiziran. To znači da se prilikom prevođenja mora točno navesti tip podataka s kojima se radi. Dakle, ako želimo napisati funkciju `bubble_sort()` koja će sortirati niz podataka koji joj se prosljeđuje kao parametar, potrebno je u zaglavlju funkcije navesti tip niza koji joj se prosljeđuje. Zbog toga je programer često prisiljen kopirati kôd funkcije i samo promijeniti deklaracije. Ovakav pristup ima dva očita nedostatka:

izvorni kôd programa “buja” preslikavanjem sličnih funkcija, a ako se primijeti pogreška u algoritmu, ispravak treba unijeti u sve preslikane verzije funkcija.

Još jednostavniji ali i znatno opasniji primjer je određivanje minimuma dvaju elemenata. Htjeli bismo napisati funkciju koja će vratiti manji od dva elementa. Problem je u tome što prilikom deklaracije parametara funkcije obavezno moramo navesti tipove podataka koji se prosljeđuju. Privlačna, ali zato suptilno opasna alternativa definiciji funkcije je korištenje makro funkcija (o njima će još biti govora u poglavlju 14):

```
#define manji(a, b)      ((a) < (b) ? (a) : (b))
```

Ovakvo rješenje æe savršeno funkcionirati za jednostavne primjere kao

```
manji(10, 20)
manji('a', 'z')
```

No dobiveno rješenje je zapravo dvolièni Janus koji svoje pravo lice otkriva tek ako mu kao parametar proslijedimo izraz. Naime, makro funkcije se prevode tako da se poziv jednostavno zamijeni tekstom definicije makro funkcije, pri èemu se formalni parametri zamijene stvarnima. Izraz se tako izvodi dva puta, što može bitno utjecati na rezultat:

```
int i = 4, j;
j = manji(++i, 10); // suprotno očekivanju j će sadržavati
                    // broj 6
```

U gornjem primjeru varijabla `j` æe nakon poziva makro naredbe `manji()` umjesto oèekivane vrijednosti 5 sadržavati broj 6. To se vidi ako se ruèno zamijeni definicija makro funkcije stvarnim naredbama, kako bi to uèinio pretprocesor:

```
j = ((++i) < (10) ? (++i) : (10));
```

Poveæavanje varijable `i` se obavlja dva puta: jednom prije i jednom poslije usporedbe, pa se i cjelokupni rezultat dosta razlikuje od oèekivanog.

C++ jezik nudi rješenje na zadane probleme u obliku *predložaka* (engl. *templates*). Oni omogućavaju upravo željeno svojstvo da se samo jednom navede algoritam koji se zatim poziva za različite tipove podataka. Postoje dvije osnovne vrste predložaka: predlošci funkcija i predlošci klasa. Prvi omogućavaju definiciju općih funkcija, dok drugi omogućavaju definiciju općih klasa.

12.2. Predlošci funkcija

Problem odreðivanja minimuma dvaju objekata se može elegantno riješiti predloškom funkcije. Definirat æe se funkcija èiji æe parametri biti opæenitog tipa. Prilikom prevoðenja prevoditelj æe sam generirati definiciju funkcije na osnovu stvarnih tipova parametara navedenih u pozivu funkcije.

12.2.1. Definicija predloška funkcije

Definicija predloška funkcije zapoèinje se kljuènom rijeèi `template`. Svaki predložak ima listu formalnih parametara koja se navodi između znakova < (manje od) i > (veæe od). Lista formalnih parametara ne može biti prazna.

Svaki formalni parametar se sastoji od ključne riječi `class` iza koje se navodi identifikator. Taj identifikator predstavlja neki potencijalni ugrađeni ili korisnički tip koji će se navesti prilikom poziva funkcije. Funkcija `manji()` se pomoću predložaka može ovako napisati:

```
template <class NekiTip>
NekiTip manji(NekiTip a, NekiTip b) {
    return a < b ? a : b;
}
```

Gornja definicija predloška specificira `NekiTip` kao identifikator tipa koji æe se kasnije navesti prilikom poziva funkcije. Iako je zapis funkcije slièan navedenoj makro-naredbi, ovakva definicija oznaèava funkciju, èiji æe se parametri izraèunati prije poziva. Zbog toga æe sada poziv

```
int j = manji(++i, 10);     // sada je OK
```

dati oèekivani rezultat. Predložak nije makro funkcija – prilikom prevođenja predloška prevoditelj neæe parametre funkcije samo ubaciti u definiciju predloška. Ovdje se radi o definiciji (hipotetski) beskonaènog skupa funkcija koje su parametrizirane tipom. Prilikom prevođenja, predložak æe za svaki pojedini tip parametra rezultirati jednom sasvim konkretnom funkcijom, koja æe se pozivati na uobièajeni naèin.

Identifikator tipa se u listi formalnih parametara predloška smije pojaviti samo jednom:

```
template <class T, class T>     // pogreška
void func(T a) {
    // ...
}
```

U gornjem primjeru u definiciji predloška je parametar `T` upotrijebljen dva puta. Međutim, isti identifikator se može koristiti bez ogranièenja u razlièitim definicijama predložaka, na primjer:

```
template <class Tip>
Tip manji(Tip a, Tip b);

template <class Tip>
Tip max(Tip a, Tip b);
```

Svaki formalni parametar mora biti oznaèen kljuènom rijeèi `class`. Sljedeæa definicija je neispravna:

```
// pogreška: mora biti <class Elem, class Zbroji>
template <class Elem, Zbroji>
void Zbroji (Elem *niz, Zbroji zbr);
```

Formalne parametre `NekiTip`, `Tip` i `Elem` iz gornjih primjera treba shvatiti kao simbolièke oznake za tipove koji æe se navesti prilikom poziva funkcije. Na primjer, ako æemo funkciju `manji()` pozvati za usporeðivanje cijelih brojeva, tada æe parametar `NekiTip` poprimiti vrijednost `int`, a ako æemo usporeðivati cipele po velièini (opisane klasom `Cipela`) parametar `NekiTip` æe poprimiti vrijednost `Cipela`.

Iza ključne riječi `template` i liste formalnih parametara navodi se ili deklaracija ili definicija funkcije. Prilikom pisanja definicije funkcije vrijede sva dosad navedena pravila C++ jezika, s tom razlikom što se unutar deklaracije i definicije funkcije za navođenje tipova koji se obrađuju koriste formalni parametri predloška. Na primjer, umjesto da se navede stvarni tip podataka čiji se minimum traži u funkciji `manji()`, navest će se tip `NekiTip`. Formalni parametri se mogu pojaviti na svim mjestima gdje se može pojaviti bilo koja regularna oznaka tipa: u listi parametara, za specificiranje povratne vrijednosti, za deklaraciju varijabli i u izrazima za dodjelu tipa.

Da bismo demonstrirali upotrebu predložaka, realizirat ćemo funkciju koja određuje minimalni član nekog niza. Funkcija će biti realizirana pomoću predloška parametriziranim tipom elemenata koji čine niz. Funkciji se preko parametara prosljeđuje pokazivač na početak niza i duljina niza, a ona vraća referencu na objekt koji je pronađen kao minimalni. Evo realizacije funkcije:

```
template <class Elem>
Elem &manji_u_nizu(Elem *niz, int brElem) {
    Elem *pokNajmanji;          // lokalna varijabla
    pokNajmanji = niz;
    for (int i = 1; i < brElem; i++)
        if (niz[i] < *pokNajmanji) pokNajmanji = &niz[i];
    return *pokNajmanji;
}
```

Formalni parametar predloška je `Elem`. Pomoæu njega je parametar `niz` oznaèen kao pokazivaè na tip `Elem`, kao i lokalna varijabla `pokNajmanji`. Osim toga, povratna vrijednost funkcije je `Elem &`, što zapravo znaèi da se vraæa referenca na tip `Elem`.

> Dozvoljeno je koristiti modifikatore `*` i `&` u kombinaciji s identifikatorom `Elem` kako bi se označili pokazivači i reference na tip.

Prilikom poziva funkcije, prevoditelj æe identifikator `Elem` zamijeniti stvarnim tipom navedenim u pozivu te æe tako odrediti stvarni tip parametara funkcije i lokalne

varijable. Taj postupak zamjene se naziva *instanciranje predloška funkcije* (engl. *function template instantiation*).

Identifikator formalnog parametra nalazi se u području imena koje se proteže do kraja definicije funkcije. Isto ime iz globalnog područja imena je skriveno te mu se može pristupiti samo pomoću operatora za određivanje područja.

Predložak funkcije može biti deklariran kao umetnut, eksterni ili statički, baš kao i kod običnih funkcija. Ključne riječi `inline`, `extern` i `static` se tada stavljaju iza liste parametara predloška:

```
template <class Tip>
inline Tip manji(Tip a, Tip b) {
    return a < b ? a : b;
}

template <class Tip>
static void sortiraj(Tip *niz, int duljina);
```

Deklaracija predloška funkcije može biti odvojena od definicije. U gornjem primjeru je funkcija `manji()` istovremeno deklarirana i definirana, dok je funkcija `sortiraj()` samo deklarirana, a njena definicija se u tom sluèaju mora dati kasnije.

12.2.2. Parametri predloška funkcije

Poseban skup pravila definira pravila kojima se podvrgavaju parametri predloška. Ta pravila su se znaèajno promijenila prilikom razvoja i napretka C++ jezika, pa je prilikom korištenja predložaka potrebno o tome voditi dosta raèuna. Mnogi prevoditelji još ne podržavaju novosti u C++ standardu koje je komitet za standardizaciju izglasao 1994. godine, pa je potrebno informirati se o tome kako vaš prevoditelj podržava predloške.

Važno pravilo za deklaraciju parametara predložaka po staroj verziji C++ standarda jest bilo da svaki formalni parametar obavezno mora biti iskorišten barem jednom u listi parametara funkcije. On se mogao pojaviti u njoj i više puta, ali je morao biti iskorišten barem jednom. Razlog tome ograničenju bio je u tome što je prevoditelj pomoću tipova argumenata funkcije navedenih u pozivu određivao o kojem se predlošku radi. Nije postojao mehanizam kojim bi se parametri, koji nisu bili spomenuti u listi argumenata funkcije, mogli dodatno specificirati.

Također, formalni parametar nije mogao biti iskorišten samo za specificiranje tipa povratne vrijednosti, na primjer:

```
// pogreška u okviru starog C++ standarda: Rez se koristi samo
// za tip povratne vrijednosti
template <class Rez, class Elem>
Rez prosjek(Elem *niz, int brElem);

// pogreška u okviru starog C++ standarda: Rez se ne spominje
// u potpisu funkcije
```

```
template <class Rez, class Elem>
void prosjek(Elem *niz, int brElem);
```

Danas to više nije sluèaj: dozvoljeno je prilikom deklaracije predloška navesti parametre koji se koriste samo u povratnoj vrijednosti funkcije. Štoviše, moguæe je određeni parametar uopæe ne koristiti niti u argumentima niti u povratnoj vrijednosti. Prethodne su dvije deklaracije stoga danas sasvim ispravne. Tipovi koji nedostaju specificiraju se prilikom instantacije predloška tako da se navedu u paru znakova `<` i `>`. Tako æe se verzija predloška `prosjek` koja uzima polje `float` brojeva, a vraæa `double`, specificirati ovako:

```
float polje[100];
double pros = prosjek<double, float>(polje, 100);
```

Znak `<` za specifikaciju parametara predloška se obavezno mora pisati neposredno uz naziv funkcije. U suprotnom æe se on interpretirati kao operator usporedbe *manje-od*.

Parametar predloška funkcije koji nije iskorišten u listi argumenata može se koristiti za, primjerice, podešavanje preciznosti raèunanja:

```
template <class Rez, class Elem, class Preciznost>
Rez prosjek(Elem *niz, int brElem);
```

Prilikom uvođenja predložaka u C++ jezik izabrana je kljuèna rijeè `class` kao rijeè koja specificira da određeni parametar predloška predstavlja neki tip. Podrazumijevalo se da taj tip ne mora biti klasa, veæ može biti i ugrađeni tip ili pobrojenje. Kako bi se istakla uloga pojedinog parametra u deklaraciji predloška, uvedena je nova kljuèna rijeè `typename` koja se može koristiti umjesto kljuène rijeèi `class`. Time se eksplicitnije naznaèava da se određeni argument predloška zapravo ime tipa, a ne klasa. Kljuèna rijeè `typename` ima dodatno znaèenje za predloške – njome se unutar predloška može odrediti da određeni izraz predstavlja tip, a ne nešto drugo. Na primjer:

```
template <typename T>       // T označava tip
void funkcija() {
    T::A *pok1;             // ove dvije deklaracije imaju
    typename T::A *pok2;    // različito značenje
}
```

U gornjem primjeru naoko bezopasna deklaracija `T::A` ima razlièito znaèenje ovisno o tome stavi li se ispred nje `typename` ili ne. U našem sluèaju `T` je neki tip, unutar kojeg je moguæe definirati druge tipove. Pretpostavimo da, zbog potreba problema koji rješavamo, svaki tip kojim æemo instancirati predložak mora imati u sebi definiran tip `A` (na primjer, ugniježđenu klasu koja se zove `A`). Programer bi zasigurno želio pristupiti tom tipu unutar funkcije, no pitanje je kako. Prva deklaracija ne èini ono što želimo –

ona se interpretira tako da se unutar tipa `T` pristupi identifikatoru `A`, no dobiveni rezultat se ne tretira kao tip koji se može naæi u deklaraciji, nego kao: *uzmi èlan `A` iz tipa `T` te ga pomnoži s `pok1`*. Oèito je to pogreška.

U drugom slučaju ključna riječ `typename` označava prevoditelju da `T::A` predstavlja tip – prilikom instantacije funkcije specificirat će se stvarni tip te će se pristupiti tipu `A` koji se nalazi u navedenom tipu. `pok2` se stoga deklarira kao pokazivač na taj tip. Ako je određeni tip potrebno često spominjati unutar predloška, moguće je deklarirati sinonim za tip pomoću deklaracije `typedef`:

```
typedef typename T::A T_ov_A;
```

Ključna rijeè `typename`, baš kao i eksplicitno navoðenje tipa u pozivu predloška nisu podržani od mnogih C++ prevoditelja pa ih koristite s oprezom.

Parametar predloška funkcije može biti i neka konstanta kojom se dodatno upravlja radom funkcije. Na primjer, moguće je predložak funkcije za učitavanje polja podataka proizvoljnog tipa s analogno/digitalnog pretvarača parametrizirati cjelobrojnom konstantom:

```
template <class TipPolja, int brElem>
void ADCUcitajPolje(TipPolja *polje) {
    TipPolja pomocnoPolje[brElem];
    // ...
}
```

U gornjem primjeru je deklariran predložak funkcije koji je parametriziran tipom podataka koji se uèitava s pretvaraèa, ali i s brojem elemenata koji se uèitavaju. Taj broj æe se navesti prilikom instantacije predloška, a u tijelu funkcije se koristi baš kao da je naveden u listi parametara funkcije. Razlika je u tome što æe se taj parametar prilikom instantacije funkcije zamijeniti navedenim parametrom, a neæe se prenijeti funkciji prilikom poziva. Tako æe se, primjerice, generirati funkcija `ADCUcitajPolje<Vektor, 10>()` koja æe uèitavati polje vektora od deset elemenata. Za uèitavanje polja vektora od dvadeset èlanova koristit æemo funkciju `ADCUcitajPolje<Vektor, 20>()`. Prilikom generiranja kôda obiju funkcija prevoditelj æe zamijeniti pozive `brElem` unutar funkcije sa stvarnom konstantom. Time smo dobili moguænost deklaracije lokalnog polja `pomocnoPolje` s promjenjivim brojem elemenata: kako je vrijednost `brElem` uvijek poznata prilikom prevoðenja, dozvoljeno je koristiti parametar predloška za deklaraciju polja. No takvu pogodnost plaæamo time što za svaku duljinu za koju pozovemo funkciju dobijemo posebnu verziju generirane funkcije. U sluèaju intenzivnijeg korištenja predloška to može dovesti do prekomjernog bujanja kôda (engl. *code bloat*). Kako u prvobitnoj verziji C++ standarda nije bilo moguæe navesti eksplicitno parametre predloška prilikom instantacije, moguænost konstantnih parametara također nije postojala.

Slično podrazumijevanim parametrima funkcija, nova verzija C++ standarda omogućava korištenje podrazumijevanih parametara predložaka. Ako, na primjer, često

koristimo predložak kojemu poseban tip specificira preciznost računa, i taj tip u većini slučajeva iznosi double, predložak možemo definirati ovako:

```
template <class T, class Preciznost = double>
void Inverz(T[100][100]);
```

Time æemo si uštedjeti na tipkanju prilikom poziva predloška: umjesto navoðenja Inverz<double>() dovoljno je navesti Inverz<>().

Prilikom korištenja podrazumijevanih parametara predloška ako je lista parametara predloška prazna potrebno je iza naziva funkcije navesti prazan par znakova <>.

12.2.3. Instanciranje predloška funkcije

Definicija predloška daje prevoditelju samo opæi algoritam koji se koristi za rješenje nekog problema. No prevoditelj ne može stvoriti kôd same funkcije prije nego što stvarno dozna na koji je naèin potrebno zamijeniti formalne parametre stvarnima. Zbog toga æe prevoditelj prilikom prevoðenja kôda preskoèiti definiciju predloška: ona može sadržavati i grube pogreške, ali je sva prilika da one na tom mjestu uopæe neæe biti prepoznate.

Podaci o aktualnim tipovima postaju dostupni tek kada se funkcija pozove. Tada se obavlja postupak instantacije. Prevoditelj provodi zamjenu tipova, prevodi funkciju i generira njen izvedbeni kôd. Na primjer:

```
template <class Tip>
Tip manji(Tip a, Tip b) {
    return a < b ? a : b;
}

int main() {
    cout << "Od 5 i 6:  " << manji(5, 6) << endl;
    cout << "Od 'a' i 'b' je " << manji('a', 'b') << endl;
    return 0;
}
```

U gornjem programu funkcija manji() se poziva dva puta. Prvi put se kao parametri navode cjelobrojne konstante, dok se drugi put navode znakovne konstante. Za svaki od ova dva poziva stvara se po jedna verzija funkcije. Prevoditelj koristi tipove argumenata da bi odredio koji se predložak funkcije koristi. Zbog toga se i svi formalni argumenti predloška obavezno moraju pojaviti u listi parametara funkcije.

U prvom slučaju se općeniti tip Tip zamjenjuje tipom int, a u drugom slučaju s char. Nakon supstitucije tipova, stvaraju se dvije preopterećene verzije funkcije manji(), čiji bi ekvivalent u izvornom kôdu bio:

```
int manji(int a, int b) {
    return a < b ? a : b;
}

char manji(char a, char b) {
    return a < b ? a : b;
}
```

Stvaraju se dvije verzije funkcije, baš kao da smo i sami napisali obje verzije, no posao generiranja razlièitih verzija za iste parametre je prebaèen na prevoditelja. Time je, takoðer, uklonjena moguænost pogreške prilikom ruènog kopiranja kôda u razlièite varijante funkcije.

U posljednjoj verziji C++ standarda je parametre predloška moguće eksplicitno navesti unutar para znakova `< i >`. Unutar tih znakova moguće je navesti eventualne konstantne parametre koje predložak može imati. To svojstvo se najviše koristi prilikom specificiranja parametara koji određuju povratni tipova ili koji se uopće ne spominju u deklaraciji funkcije. Na primjer:

```
template <class lokalniPodaci>
int Usporedi(char *niz1, char *niz2);
```

U gornjem primjeru deklarirali smo predložak funkcije koja æe obavljati usporedbu dvaju znakovnih nizova, s time da æe uvažavati set znakova definiran u abecedi korisnika programa. U tom sluèaju moguæe je definirati klasu za svaki pojedini jezik koja æe definirati naèin usporedbe (primjerice, posjedovat æe odreðenu statièku funkciju koja æe obavljati usporedbu). Ta klasa se navodi prilikom poziva predloška. Na primjer:

```
char *s1, *s2;
int usp1 = Usporedi<Hrvatski>(s1, s2);
int usp2 = Usporedi<English>(s1, s2);
```

Ako se argument predloška ne navede eksplicitno, prevoditelj æe ga pokušati razluèiti pomoæu liste argumenata navedenih u pozivu funkcije. Moguæ je kombinirani poziv: neki parametri se navedu eksplicitno, a preostali se zakljuèe iz liste. Na primjer:

```
template <class Preciznost, class Tip>
Preciznost Prosjek(Tip *polje, int brElem);
```

Predložak se može pozvati tako da se eksplicitno navede samo vrijednost parametra `Preciznost`, dok æe se vrijednost parametra `Tip` zakljuèiti iz poziva:

```
float polje[100];
double pros = prosjek<double>(polje, 100);
```

Pri tome, ako se navode parametri predloška, dozvoljeno je izostaviti samo parametre s kraja liste – nije moguæe, primjerice, izostaviti prvi parametar i oèekivati da æe on biti zakljuèen iz poziva, a drugi specificirati eksplicitno, na primjer:

```
template <class Tip, class Preciznost>
Preciznost Prosjek(Tip *polje, int brElem);

float polje[100];
double pros = prosjek<double>(polje, 100);  // pogrešno
```

U gornjem primjeru prevoditelj æe rezonirati ovako: prvi navedeni argument je `double`, što æe se dodijeliti parametru `Tip`, dok se drugi argument `Preciznost` ne može zaključiti iz poziva (nije naveden u listi parametara funkcije) pa je poziv neispravan.

Prilikom navođenja liste parametara potrebno je znak < koji započinje listu uvijek “nalijepiti” uz naziv funkcije: time se prevoditelju daje na znanje da slijedi lista parametara. U suprotnom će prevoditelj taj znak protumačiti kao znak za *manje-od*.

Algoritam za pronalaženje parametara predloška koji nisu eksplicitno navedeni uspoređuje potpis predloška funkcije s pozivom funkcije. Da bi instanciranje uspjelo, mora se moæi postiæi slaganje stvarnih i formalnih parametara. Na primjer, funkcija `manji_u_nizu()`za prvi argument ima pokazivaè na tip, pa stoga poziv mora kao prvi argument također imati pokazivaè:

```
template <class Elem>
Elem &manji_u_nizu(Elem *niz, int brElem);

// pogreška: prvi argument mora biti pokazivač
int a, b;
b = manji_u_nizu(a, b);

// OK: formalni i stvarni tipovi se slažu
int niz[] = {1, 4, 7, 2, 4, 1, 3};
int i = manji_u_nizu(niz, 7);
```

Također, neophodno je poštivati i `const` modifikator. Na primjer, opæe je pravilo da se pokazivaè na konstantu ne može pridodijeliti pokazivaèu na promjenjivu vrijednost:

```
// pogreška: niz je pokazivač na const
const int niz[] = {1, 7, 3, 2};
int k = manji_u_nizu(niz, 4);
```

Postupak uspoređivanja formalnih i stvarnih parametara prilikom poziva predloška funkcije teèe po sljedeæem redoslijedu:

1. Svaki formalni parametar predloška funkcije se ispituje sadrži li neki formalni tip predloška.

2. Ako je pronađen formalni tip predloška, ustanovljava se tip odgovarajućeg stvarnog parametra navedenog u pozivu.
3. Tipovi formalnog i stvarnog parametra se tada uparuju tako da se uklone svi dodatni modifikatori tipa.

Na primjer, ako bismo funkciju `manji_u_nizu()` pozvali kao u sljedeæem primjeru, formalni parametar `Elem` bi predstavljao tip `int *`:

```
int **i;
manji_u_nizu(i, 1);
```

Ako se formalni parametar predloška pojavljuje u listi parametara funkcije više puta, prilikom poziva funkcije na svakom njegovom mjestu u listi parametara mora se navesti doslovno isti tip. Pritom se ne provode nikakve konverzije tipa.

Na primjer:

```
// pogreška: prvi parametar je unsigned int, a drugi je int
unsigned int i = 5, j;
j = manji(i, -5);
```

U gornjem primjeru dobit æemo pogrešku prilikom prevoðenja – predložak funkcije `manji()` specificira da æe prvi i drugi parametri biti istog tipa, što ovdje nije sluèaj (prvi parametar je `unsigned int`, a drugi je `int`).

Niti na argumente funkcije èiji tip nije definiran formalnim parametrom predloška se ne primjenjuju pravila konverzije.

Na primjer:

```
// pogreška: drugi parametar ne odgovara po tipu
int niz[] = {3, 5, 6, 2, 4};
long dulj = 5;
int m = manji_u_nizu(niz, dulj);
```

Gornji poziv neæe biti uspješan jer je potrebno postiæi potpuno slaganje tipova, a u konkretnom pozivu je za drugi parametar umjesto podatka tipa `int` naveden podatak tipa `long`. Da bi poziv uspio, potrebno je primijeniti eksplicitnu dodjelu tipa:

```
m = manji_u_nizu(niz, (int)dulj);
```

Nakon uparivanja formalnih i stvarnih argumenata prevoditelj æe generirati verziju funkcije za dane tipove te æe se tada provjeriti sintaksa samog predloška. Prilikom

prevođenja samog predloška prevoditelj zapravo uopæe ne analizira napisani kôd; on može biti i potpuno sintaktièki neispravan. Tek kada se prilikom instanciranja funkcije doznaju tipovi parametara, prevoditelj ima sve podatke potrebne da instancira predložak.

Zbog toga prevođenje predloška jednom može uspješno proći, dok će za neke druge formalne tipove prevoditelj pronaći niz pogrešaka. To najčešće ovisi o svojstvima samog tipa koji se navodi kao stvarni parametar. Na primjer, naša funkcija `manji_u_nizu()` može funkcionirati ispravno ako je za tip koji joj je proslijeđen definirana operacija usporedbe. Kako se ugrađeni cjelobrojni tip može uspoređivati, poziv

```
int niz[] = {5, 2, 7, 3};
int m = manji_u_nizu(niz, 4);
```

se može prevesti bez problema. Prevoditelj ima sve što mu je potrebno da generira ispravnu verziju funkcije. Naprotiv, ako pozovemo funkciju za tip za koji nema definirane operacije usporedbe, dobit æemo pogrešku prilikom prevođenja. Na primjer:

```
Vektor *niz, m;
// inicijaliziraj niz
m = manji_u_nizu(niz, 5);
```

Ovakav poziv rezultira pogreškom zato jer prevoditelj prilikom instanciranja funkcije ne zna kako treba usporediti dva vektora. Ako izmislimo naèin usporeðivanja vektora i ugradimo ga u klasu `Vektor` tako da preopteretimo operator `<`, gornji poziv postaje ispravan i on stvarno nalazi najmanji vektor s obzirom na zadani kriterij usporedbe.

Zadatak. *Napišite predložak funkcije `srednja_vrijednost()`. Kao parametar ona će uzimati polje i cijeli broj koji će pokazivati duljinu polja. Rezultat će biti istog tipa kojeg su i tipovi elemenata polja. Pri tome se za zbrajanje članova polja i za dijeljenje člana polja cijelim brojem koriste operatori `+` i `/` definirani u klasi koja opisuje tip elemenata polja. Pretpostavite da član polja ne mora imati podrazumijevani konstruktor, ali sigurno ima ispravan konstruktor kopije.*

Zadatak. *Napišite predložak funkcije `binarno_pretrazi()` koja će kao parametre imati polje, referencu na objekt istog tipa kojeg su i elementi polja te cijeli broj koji definira duljinu polja. Funkcija pretpostavlja da je ulazni niz sortiran po veličini te će provesti binarno pretraživanje proslijeđenog polja. Kao rezultat će biti cijeli broj koji će pokazivati indeks proslijeđenog objekta u polju ili `-1` ako objekt nije pronađen. Uputa: binarno pretraživanje radi tako da se prvo pogleda element na sredini polja. Ako je zadani element manji od traženog, pretražuje se preostala gornja polovica polja (jer je polje sortirano), a u suprotnom donja polovica. Postupak se ponavlja po istom principu na sve manjem i manjem dijelu polja, dok se traženi objekt ne pronađe ili se interval ne svede na jedan član. Tada je ili taj član jednak traženom ili se traženi objekt ne nalazi u polju. Za usporedbu koristite operatore `<`, `>`, `<=`, `>=`, `==` te `!=` definirane u klasi koja opisuje pojedini član polja.*

12.2.4. Eksplicitna instantacija predloška

U prethodnom odsjeèku objašnjena je implicitna instantacija predloška – prilikom poziva funkcije prevoditelj sam zakljuèuje da funkcija nije obièna funkcija, veæ se radi o predlošku te generira željenu varijantu funkcije. To je vrlo praktièno, jer se time posao s programera u cijelosti prebacuje na prevoditelj.

No postoji nekoliko problema na koje će svaki ozbiljniji C++ programer neminovno naići. Osnovno je to što da bi predložak bio uspješno instanciran, prevoditelj mora imati dostupan izvorni kôd predloška. To može biti dosta nezgodno ako želimo napraviti biblioteku funkcija definiranih pomoću predloška, te biblioteku želimo dalje distribuirati – zasigurno ne bismo htjeli pokazati konkurenciji izvorni kôd. Osim toga, čak i da predloške koristimo isključivo za vlastite potrebe, naići ćemo na probleme ako isti predložak koristimo u više datoteka izvornog kôda. Naime, prevoditelj će u svaku datoteku umetnuti po jednu verziju funkcije predloška. Prilikom povezivanja ćemo ili dobiti pogrešku o tome da je pojedina funkcija deklarirana višestruko ili će povezivač biti prisiljen izbaciti višestruke definicije predloška i koristiti samo jednu.

Svi navedeni problemi se mogu izbjeći tako da se u jednu datoteku smjeste svi predlošci koji se koriste u programu – u druge datoteke nećemo morati uključiti izvorni kôd predloška, već ćemo funkciju samo deklarirati, a prilikom povezivanja će se ti pozivi povezati s definicijom iz druge datoteke. Time predlošci zaista postaju slični običnim funkcijama – moguće je čak napraviti biblioteku predložaka koju samo prilikom povezivanja uključimo. Doduše, ta biblioteka mora sadržavati instancirane predloške za sve moguće tipove koje korisnik poželi koristiti u predlošku. To ponekad nije moguće (na primjer, za predložak funkcije koja ispisuje polje objekata na ekran pomoću operatora `>>` – taj je operator moguće preopteretiti za bilo koji tip podataka i tako iskoristiti predložak za bilo koji tip) te je u tom slučaju potrebno distribuirati izvorni kôd predloška. No za neke vrste predložaka to nije ograničenje.

Kako bi se omogućilo elegantno rješenje gornjih problema, u posljednju varijantu C++ jezika je ubačena mogućnost eksplicitne instantacije predloška. Mnogi prevoditelji to još ne podržavaju, ali vjerujemo da će ta mogućnost biti uskoro dodana većini prevoditelja jer je često vrlo bitna za uspješnu primjenu.

Eksplicitna instantacija se provodi tako da se iza ključne riječi `template` navede naziv predloška s listom parametara predloška i listom parametara funkcije:

```
template <class Preciznost, class Tip>      // deklaracija
Preciznost Prosjek(Tip *polje, int brElem);

// eksplicitne instantacije
template double Prosjek<double, float>(float *, int);
template long Prosjek<long, int>(int *, int);
```

U gornjem primjeru stvorit æe se dvije funkcije instancirane s razlièitim setom parametara: `<double, float>` i `<long, int>`. Prilikom eksplicitne instantacije potrebno je navesti puni potpis funkcije – time je omoguæena instantacija samo željene funkcije od ukupnog skupa preoptereæenih funkcija predložaka.

Kao i prilikom implicitne instantacije, pojedini tip se može izostaviti ako se njegova vrijednost može odrediti pomoću liste argumenata:

```
// argument Tip će se postaviti na float:
template Prosjek<double>(float *, int);
```

12.2.5. Preoptereæivanje predložaka funkcija

Predložak funkcije može biti preoptereæen proizvoljan broj puta pod uvjetom da se potpisi funkcija razlikuju po tipu i/ili broju argumenata. Na primjer, možemo definirati funkciju `zbroji()` koja zbraja objekte:

```
// zbrajanje dvaju objekata
template <class Tip>
Tip zbroji(Tip a, Tip b);

// zbrajanje triju objekata
template <class Tip>
Tip zbroji(Tip a, Tip b, Tip c);

// zbrajanje dvaju nizova
template <class Tip>
void zbroji(Tip *niza, Tip *nizb, Tip *rez, int brElem);
```

Nije moguæe preopteretiti funkcije tako da se razlikuju samo u povratnom tipu:

```
// pogreška: funkcija se razlikuje samo u povratnom tipu
template <class Tip>
int zbroji(Tip a, Tip b);
```

Prilikom poziva preoptereæenih predložaka funkcija potrebno je biti vrlo pažljiv. Naime, reèeno je da ako se formalni argument predloška pojavljuje nekoliko puta u listi parametara funkcije, onda se prilikom poziva on mora svaki put zamijeniti istim tipom. To znaèi da sljedeæi poziv neæe uspjeti:

```
int i = 9, k;
unsigned int j = 6;
k = zbroji(i, j);   // krivo: i je int, a j je unsigned int
```

U gornjem primjeru prvi parametar je tipa `int`, a drugi je `unsigned int`. Kako se konverzije tipova ne provode prilikom poziva predložaka funkcija, gornji poziv ne uspijeva. Ako bismo željeli zbrajati razlièite tipove preoptereæenom verzijom funkcije `zbroji()`, morali bismo dodati novu verziju predloška:

```
template <class Tip1, class Tip2>
Tip1 zbroji(Tip1 a, Tip2 b);
```

No ovakvo rješenje dodatno komplicira problem: što je s pozivom kada su oba argumenta istog tipa?

```
int i = 5, j = 7, k;
k = zbroji(i, j);   // koji se poziva: zbroji(Tip, Tip) ili
                    // zbroji(Tip1, Tip2) ?
```

Sada se ravnopravno mogu pozvati dvije varijante funkcije `zbroji()`, pa æe prevoditelj javiti pogrešku prilikom prevođenja. Na prvi pogled se èini da bismo mogli jednostavno izbaciti prvu definiciju s jednim tipom. To je donekle i toèno: poziv kada su oba argumenta istog tipa se uvijek može obuhvatiti varijantom funkcije koja prima razlièite tipove za parametre. No problem je koji od tipova `Tip1` i `Tip2` treba staviti kao rezultat?

Odgovor glasi da zapravo nema odgovora. Rezultat mora ponekad biti tipa `Tip1`, a drugi put tipa `Tip2`, ovisno o redoslijedu navođenja stvarnih parametara prilikom poziva funkcije. Zbog toga bi bilo poželjno dodati treći parametar predlošku koji će specificirati povratni tip. U najnovijoj verziji C++ jezika to se može učiniti prilično jednostavno, na sljedeći način:

```
template <class Rez, class Tip1, class Tip2>
Rez zbroji(Tip1 a, Tip2 b);     // OK u posljednjoj verziji
```

Argument `Rez` je potrebno navesti prilikom instantacije predloška:

```
float a;
double b;
double rez = zbroji<double>(a, b);
```

Ako posjedujete prevoditelj koji ne podržava eksplicitno navođenje parametara, ovakvo rješenje nije moguæe. Tada programeri obièno pribjegavaju lukavom triku: funkciji se doda treæi parametar koji neæe prenositi nikakav konkretan sadržaj, osim što æe saopæiti prevoditelju povratni tip:

```
template <class Rez, class Tip1, class Tip2>
Rez zbroji(Tip1 a, Tip2 b, Rez);
```

Treæi argument nema ime, èime se signalizira prevoditelju da se on u funkciji ne koristi i da nije potrebno generirati upozorenje o suvišnom parametru koji se ne koristi u funkciji. Prilikom instantacije predloška potrebno je navesti neku vrijednost kako bi prevoditelj doznao povratni tip:

```
float a;
double b, samoZaPovratniTip;
double rez = zbroji(a, b, samoZaPovratniTip);
```

12.2.6. Specijalizacije predložaka funkcija

Postoje situacije u kojima opæeniti algoritam predloška može biti neodgovarajuæi ili neefikasan za pojedine tipove podataka. Na primjer, naša funkcija `manji()` neæe ispravno usporediti dva znakovna niza, zato jer se operator < ne interpretira kao usporedba nizova, nego kao usporedba pokazivaèa. U takvim sluèajevima programer može napisati *specijalizaciju predloška funkcije* (engl. *function template specialization*) koja se koristi samo za taj tip podataka.

Specijalizacija se definira tako da se bez korištenja ključne riječi `template` jednostavno napiše željena funkcija. Na primjeru funkcije `manji()` to izgleda ovako:

```
#include <string.h>

// predložak za općenite tipove
template <class Tip>
Tip manji(Tip a, Tip b) {
    return a < b ? a : b;
}

// specijalizacija za char slučaj
char *manji(char *a, char *b) {
    return strcmp(a, b) < 0 ? a : b;
}
```

Za sve pozive funkcije `manji()` koristi se odgovarajuæa instantacija predloška, osim u sluèaju poziva kada se traži minimum niza znakova. Tada se poziva specijalizacija:

```
// poziva specijalizaciju
char *rez = manji("manji", "veći");
```

U posljednjoj verziji C++ jezika moguæe je prilikom specijalizacije unutar znakova < i > navesti parametre za koje se definira specijalizacija. Na primjer:

```
char *manji<char *>(char *a, char *b);
```

Lista parametara predloška može biti i prazna, ako se oni mogu zakljuèiti iz liste argumenta funkcije:

```
char *manji<>(char *a, char *b);
```

Kod predložaka s više parametara moguæe je napraviti i takozvanu *djelomiènu specijalizaciju* (engl. *partial specialization*): specijalizira se samo dio parametara, dok se ostali ostave opæima:

```
template <class T1, class T2>
void f();               // originalni predložak
```

```
template <class T>
f<T, char>();            // djelomična specijalizacija
```

U gornjem primjeru prva deklaracija definira opæi predložak parametriziran s dva tipa. Druga deklaracija je djelomièna specijalizacija: ona je parametrizirana jednim tipom, no drugi argument je `char`. Ta specijalizacija æe se koristiti za sve oblike funkcije kod kojih navedemo `char` kao drugi argument: `f<long, char>()`, `f<Kompleksni, char>()`, `f(int (*)(int, Kompleksni &), char>()` itd.

Postupak određivanja odgovarajuće funkcije prilikom poziva izvodi se po sljedećem algoritmu:

1. Prvo se pretražuju sve specijalizacije predloška te se traži točno podudaranje argumenata. Ako se nađe više funkcija koje odgovaraju, dojavljuje se pogreška prilikom prevođenja. Ako se nađe točno jedna odgovarajuća funkcija, postupak završava. U protivnom se prelazi na sljedeći korak.
2. Pretražuju se svi predlošci funkcije te se traži točno podudaranje parametara. Ako se pronađe više od jednog odgovarajućeg predloška, javlja se pogreška prilikom prevođenja. Ako se pronađe točno jedan predložak, funkcija je određena te se provjerava je li već generirana varijanta funkcije za zadane parametre. Ako nije, prevoditelj prevodi predložak te generira traženu funkciju. Ako nije pronađen odgovarajući predložak, prelazi se na sljedeći korak.
3. Sve specijalizacije se promatraju kao skup preopterećenih funkcija te se pokušava odrediti odgovarajuća funkcija pravilima za poziv preopterećene funkcije, što uključuje i moguće konverzije tipa.

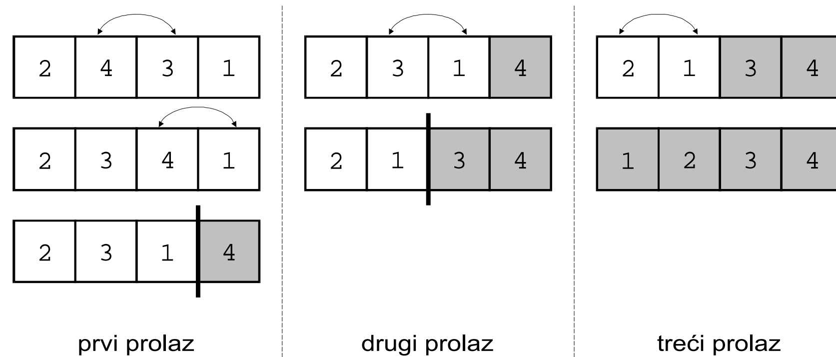
Zadatak. *Napišite specijalizaciju funkcije* `binarno_pretrazi()` *iz zadatka na stranici 401 tako da ona ispravno radi za znakovne nizove.*

12.2.7. Primjer predloška funkcije za *bubble sort*

Na kraju odsjeèka o predlošcima funkcija, evo konkretnog primjera njihovog korištenja. Dana je funkcija `bubble_sort()` koja je u stanju sortirati niz bilo kojih tipova elemenata za koje postoji operacija usporeðivanja (konkretnije operator <). Takoðer, zadani tip mora imati i definiran operator pridruživanja. Parametri su niz i duljina niza. Funkcija `bubble_sort()` također poziva funkciju `zamijeni()` koja provodi zamjenu dva elementa niza.

Evo kratkog objašnjenja kako radi navedena metoda sortiranja. Prolazi se kroz niz te se uspoređuju susjedni elementi. Ako je neki element veći od sljedećeg, potrebno ih je zamijeniti kako bi se dobio željeni rastući poredak. Nakon prvog prolaska kroz niz najveći element će se naći na kraju niza, čime je on dospio na svoje odredište. Zatim se postupak ponavlja za preostale elemente niza, s time da očito nije potrebno ići do kraja niza. Postupak sortiranja za niz od četiri broja je prikazan na slici 12.1.

Evo kôda funkcija `zamijeni()` i `bubble_sort()`:



Slika 12.1. Prikaz *bubble sort* algoritma za niz od četiri broja

```
template <class Elem>
void zamijeni(Elem& a, Elem& b) {
    Elem priv = a;
    a = b;
    b = priv;
}


template <class Elem>
void bubble_sort(Elem *niz, int brElem) {
    int ok;
    for (int i = brElem - 1; i > 0; i--) {
        ok = 1;
        for (int j = 0; j < i; j++)
            if (niz[j + 1] < niz[j]) {
                zamijeni(niz[j], niz[j + 1]);
                ok = 0;
            }
        if (ok) break;
    }
}
```

Nakon sortiranja rezultat se može ispisati funkcijom `pisi_niz()` koja se također može realizirati predloškom. Ona može ispisati bilo koji niz pod uvjetom da je za zadani tip definiran operator << za ispis na izlazni tok.

```
#include <iostream.h>

template <class Elem>
void pisi_niz(Elem *niz, int brElem) {
    cout << "{ ";
    for (int i = 0; i < brElem; i++) {
        cout << niz[i];
```

```
            //zarez se ne štampa nakon zadnjeg elementa:
            if (i < brElem - 1) cout << ", ";
        }
        cout << "}" << endl;
    }
```

Evo i primjera poziva tih funkcija, jednom za niz cijelih brojeva, a jednom za niz realnih brojeva:

```
    int main() {
        int niz1[] = {4, 5, 2, 7, 1, 0, 9};
        double niz2[] = {2.6, 7.9, 1.4, 8.9, 9.9, 3.3};
        bubble_sort(niz1, 7);
        cout << "Niz cijelih brojeva:" << endl;
        pisi_niz(niz1, 7);
        bubble_sort(niz2, 6);
        cout << "Niz realnih brojeva:" << endl;
        pisi_niz(niz2, 6);
        return 0;
    }
```

12.3. Predlošci klasa

Slièno predlošcima funkcija koji omoguæavaju definiranje opæih algoritama, koji se zatim konkretiziraju stvarnim tipovima nad kojima djeluju, postoje i *predlošci klasa* (engl. *class templates*) za definiranje opæih klasa koje se zatim konkretiziraju stvarnim tipovima. Tipièan primjer za takve klase su *kontejnerske klase* (engl. *container classes*). To su klase koje manipuliraju skupom objekata neke druge klase te se èesto koriste za pohranjivanje nizova objekata. Na primjer, neki program može sve svoje podatke držati u obliku vezane liste. Lista se može predoèiti objektom, a klasa koja definira listu je kontejnerska klasa jer ona obraðuje skup podataka neke druge klase.

Važno je uočiti da pravila za održavanje liste ne ovise o stvarnim objektima koji se smještaju u listu. Postoje standardne operacije koje se mogu obavljati nad listom: dodavanje elemenata na početak, ubacivanje elemenata, brisanje elementa, pretraživanje liste, unija dviju listi i slično. Te operacije se obavljaju na konceptualno jednak način bez obzira sadrži li lista nizove znakova, cijele brojeve ili objekte korisničkih klasa. Zbog toga je vrlo korisno definirati opći predložak klase koji definira opće algoritme za održavanje liste. Kasnije, prilikom korištenja liste navodi se stvarni tip koji će se pohraniti u listi. Prevoditelj će tada stvoriti kopiju klase koja će djelovati na točno navedeni tip koji se sprema u listi.

Time se programer oslobađa dosadnog posla kopiranja i promjene zajedničkog kostura liste za svaki pojedini tip koji se čuva u listi. Također, ako je mehanizam liste potrebno proširiti novim svojstvima ili ispraviti pogreške u algoritmu, promjena će se unijeti na jednom mjestu i automatski prenijeti na sve pozive liste. Velika je prednost i u načinu imenovanja. Ta generička klasa se može nazvati `Lista` i ona ne mijenja naziv

radi li se o listi cijelih ili kompleksnih brojeva. Ako bismo primijenili metodu ručnog kopiranja i promjene kôda klase, tada bi svaka dobivena klasa morala imati svoje posebno ime: `CjelobrLista`, `KomplLista`, `ListaNizaZnakova` i sl.

12.3.1. Definicija predloška klase

Klasu `Lista` iz poglavlja 9 i realizirat æemo pomoæu predložaka. Definirat æemo predložak koji æe opisivati opæu listu objekata proizvoljnog tipa. No prvo moramo ustanoviti koje operacije želimo podržati:

- `void UgurajClan(element, izaKojeg)` dodaje `element` iza člana na kojeg pokazuje `izaKojeg`,
- `void GoniClan(element)` izbacuje element na koji element pokazuje i
- `int prazna()` vraća jedinicu ako je lista prazna.

Za realizaciju trebamo dvije klase:

- klasa `Lista` će definirati opća svojstva liste, vodit će računa o njenom početnom i završnom članu te će definirati javno sučelje liste i
- klasa `ElementListe` koja će definirati objekt koji se smješta u listu. Taj objekt će sadržavati pokazivače na prethodni i sljedeći element te samu vrijednost objekta.

Definicija predloška klase se ne razlikuje znaèajnije od definicije obiènih klasa. Deklaracija se zapoèinje tako da se ispred kljuène rijeèi `class` umetne kljuèna rijeè `template` iza koje se unutar znakova `<` i `>` (znakovi manje-od i veæe-od) navode argumenti predloška. Na primjer:

```
template <class Tip>
class Lista;
```

Argumenti predloška klase se navode slièno kao i argumenti predloška funkcije: svaki argument sastoji se od kljuène rijeèi `class` i imena parametra. Kljuèna rijeè `template` se mora staviti ispred svake deklaracije unaprijed (kao u gornjem primjeru) ili ispred definicije klase. Lista parametara ne smije biti prazna. Više parametara se navodi razdvojeno zarezima:

```
template <class T1, class T2, class T3>
class NekaKlasa;
```

Slièno predlošcima funkcije, parametar predloška klase može biti i izraz. Taj parametar se u deklaraciji navodi slièno obiènim parametrima funkcija: umjesto kljuène rijeèi `class` stavlja se naziv tipa iza kojeg se stavlja identifikator. Na primjer, ako bismo htjeli ogranièiti broj elemenata u listi te time sprijeèiti preveliko zauzeæe memorije, mogli bismo dodati predlošku klase cjelobrojni parametar `maxElem` tipa `int`:

```
template <class Tip, int maxElem>
class OgranicenaLista;
```

Klase definirane kao predlošci koriste se u programu na isti naèin kao i obiène klase, s tom razlikom što se iza naziva klase u kutnim zagradama (< >) obavezno moraju navesti parametri. Na primjer, lista cijelih brojeva se oznaèava kao

```
Lista<int>
```

Lista kompleksnih brojeva se oznaèava s

```
Lista<Kompleksni>
```

Puni naziv liste kompleksnih brojeva ogranièene na dvadeset elemenata je

```
OgranicenaLista<Kompleksni, 20>
```

Evo primjera definicije klase `ElementListe`:

```
template <class Tip>
class ElementListe {
private:
    Tip vrij;
    ElementListe *prethodni, *sljedeci;
public:
    ElementListe *Prethodni() { return prethodni; }
    ElementListe *Sljedeci() { return sljedeci; }
    void PostaviPrethodni(ElementListe *pret)
            { prethodni = pret; }
    void PostaviSljedeci(ElementListe *sljed)
            { sljedeci = sljed; }
    ElementListe(const Tip &elem);
    ElementListe() {}
    Tip &DajVrijednost();
};
```

Unutar definicije predloška identifikator `Tip` se koristi za oznaèavanje tipa koji æe se navesti prilikom poziva klase. Pomoæu tog identifikatora možemo raditi sve što možemo raditi i sa svim drugim tipovima: stvarati objekte (kao na primjer objekt `vrij`) i prosljeðivati ih kroz parametre (kao u sluèaju konstruktora). Unutar klase se identifikator `ElementListe` koristi bez parametara, što je i logièno: parametri su navedeni u `template` deklaracije ispred poèetka deklaracije klase. Naprotiv, izvan deklaracije klase, pored identifikatora klase se uvijek moraju navesti parametri.

Unutar definicije klase naziv klase se može navesti bez parametara.

Promotrimo kako to izgleda u našem primjeru prilikom definicije konstruktora i funkcijskog èlana `DajVrijednost()`. Konstruktor æemo realizirati kao `inline`:

```
template <class Tip>
inline ElementListe<Tip>::ElementListe(const Tip &elem) :
    vrij(elem), prethodni(NULL), sljedeci(NULL) {}

template <class Tip>
inline Tip &ElementListe<Tip>::DajVrijednost() {
    return vrij;
}
```

Obratimo još pažnju i na naèin na koji je definirana klasa `ElementListe`. Naime, prilikom izrade kontejnerske klase programer je èesto suoèen s problemom da li æe klasa pamtiti objekte koji se stvaraju i uništavaju u samoj klasi, ili æe se pamtiti samo pokazivaèi odnosno reference na objekte koji su stvoreni izvan klase. U našem sluèaju smo izabrali pristup kada lista stvara kopiju vanjskog objekta. Lokalni objekt koji èuva vrijednost elementa liste je deklariran s

```
Tip vrij;
```

èime se zapravo specificira da æe svaki `ElementListe` sadržavati kopiju vanjskog objekta. Ako bismo htjeli u listi držati samo reference na objekte koji su stvoreni izvan liste, tada bismo promijenili gornju deklaraciju u

```
Tip &vrij;
```

Time bi svaki `ElementListe` sadržavao samo referencu na vanjski objekt. Okolni program æe biti odgovoran za njegovo stvaranje i uništavanje.

Nema jednoznačnog odgovora na pitanje koji je pristup bolji i ispravniji. Izabrano rješenje će ovisiti o našim potrebama. Prvo rješenje ima prednost u tome što je lista potpuno izolirana od vanjskog utjecaja: osim ako ne primjenjujemo prljave trikove nećemo nehotice uništiti sadržaj liste. Promjena u vanjskom objektu neće utjecati na promjenu objekta u listi. No zato će stvaranje svakog elementa liste biti popraćeno stvaranjem lokalne kopije vanjskog objekta koji se smješta u listu. Također, kada se uništava lista, uništavaju se i svi elementi koje ona sadržava. Drugi pristup je brži: objekt se stvori jednom i zatim se samo pokazivač (odnosno referenca) na njega ubaci u listu.

Također, u prvom slučaju jedan objekt se ne može naći istodobno u dvije različite liste. Svaka lista će sadržavati svoju kopiju, te promjena na objektu u jednoj listi neće imati utjecaja na objekt u drugoj listi. U drugom slučaju direktno možemo pokazivač (referencu) na isti objekt jednom ubaciti u jednu, a drugi put u drugu listu.

Prilikom građenja kontejnerskih klasa koje sadržavaju pokazivače na objekte postoji opasnost od *visećih referenci* (engl. *dangling references*). Naime, stvaranje i uništavanje objekata je izvan nadležnosti kontejnera. Tako možemo u listu koja je deklarirana globalno ubaciti pokazivač na lokalni objekt. Nakon što dotična funkcija završi, lokalni objekt će biti automatski uništen. Kontejner će tada sadržavati viseći pokazivač: pokazivač je ostao u listi, a objekt je uništen. Taj pokazivač sada pokazuje na

dio memorije koji uopće ne sadržava željeni objekt. Pokušaj pristupa objektu vjerojatno će rezultirati pogreškom u radu programa, a također može znatno poremetiti rad operacijskog sustava.

Zbog opasnosti od visećih referenci valja biti vrlo oprezan prilikom korištenja kontejnera koji čuvaju pokazivače na članove.

Kao što se vidi, nema opæe preporuke koju je moguæe bezuvjetno slijediti. Programer mora prilikom razvoja sustava odvagnuti razloge za i protiv te se odluèiti za pristup koji najbolje odgovara danom podruèju primjene.

I na kraju, promotrit ćemo još način na koji je izveden konstruktor klase. Kao prvo, konstruktoru se kao parametar prosljeđuje referenca na objekt. U suprotnom bi se prilikom poziva konstruktora objekt nepotrebno kopirao u lokalni objekt čime bi se usporilo izvođenje kôda. Također, lokalni objekt `vrij` se inicijalizira u inicijalizacijskoj listi konstruktora. Time se izbjegava nepotrebna inicijalizacija tog objekta podrazumijevanim konstruktorom i naknadno kopiranje parametra u taj objekt.

U posljednjoj verziji C++ jezika moguće je koristiti podrazumijevane vrijednosti parametara. Na primjer, zamislimo predložak klase `NizZnakova` koji može biti parametriziran tipom znaka koji se koristi (`char` ili `wcahr_t`, ovisno o situaciji). Gotovo sigurno ćemo u 99 % slučajeva koristiti `char`, pa je praktično `char` postaviti kao podrazumijevani parametar:

```
template <class Tip = char>
class NizZnakova;
```

Sada se niz znakova char može deklarirati ovako:

```
NizZnakova<> niz;   // OK
```

Upotreba praznih znakova `<>` je obavezna: oni prevoditelju kazuju da je `NizZnakova` predložak, bez obzira što mu je lista argumenata prazna. Deklaracija

```
NizZnakova niz;     // pogrešno
```

bi prouzroèila pogrešku prilikom prevoðenja.

12.3.2. Instanciranje predložaka klasa

Predložak klase se instancira tako da se iza naziva klase unutar znakova < i > navedu svi stvarni parametri predloška. Na primjer:

```
ElementListe<int> eli(5);
ElementListe<Kompleksni> elk(Kompleksni());
ElementListe<char *> elz("Listam");
```

Gornje definicije treba promatrati na sljedeæi naèin: ime `ElementListe` oznaèava sve moguæe elemente liste te ne opisuje željeni objekt dovoljno precizno. Zbog toga nije moguæe napraviti objekt klase navodeæi samo naziv klase, veæ je potrebno stvoriti konkretnu realizaciju predloška. Tako je `ElementListe<int>` zapravo potpuni naziv jedne konkretne klase koja opisuje element liste cijelih brojeva. Ako se prihvati da je to puno ime jedne klase (koja je dobivena preko predložaka, no sa stanovišta njenog korištenja nema nikakve razlike s ostalim klasama koje nisu definirane preko predložaka), onda je odmah jasno kako se takve klase mogu koristiti. Objekti klasa predložaka deklariraju se isto kao i objekti obiènih klasa, s time da se mora navesti puni naziv klase.

Puni naziv klase stvorene na osnovu predloška uključuje parametre koji se navode unutar znakova < i >. Za stvaranje objekata klase potrebno je koristiti njen puni naziv.

Nakon prethodnih deklaracija prevoditelj æe zapravo stvoriti tri klase. To æe biti tri nezavisne klase èiji æe puni nazivi glasiti `ElementListe<int>`, `ElementListe<Kompleksni>` i `ElementListe<char *>`. Možemo si zamisliti kao da æe prevoditelj u suštini provesti *pretraži-i-zamijeni* operaciju tri puta. Svaki put æe formalni argument `Tip` zamijeniti stvarnim tipom i dobiveni kôd ponovo prevesti. Tek tada æe se provjeriti sintaktièka ispravnost dobivenog kôda. Tako se èesto može dogoditi da predložak ispravno radi s jednim tipom kao parametrom, a ne radi s nekim drugim. U našem sluèaju, za ispravan rad predloška, klasa koja se prosljeðuje kao parametar mora imati definiran konstruktor kopije (ili podrazumijevani konstruktor kopije mora odgovarati). U suprotnom može doæi do problema prilikom kopiranja proslijeðenog objekta u lokalni objekt `vrij`. Ovo je primjer kada se predložak može prevesti, ali ne funkcionira ispravno. No postoje i sluèajevi kada instanciranje jednim tipom prolazi bez problema, dok instanciranje drugim tipom prouzrokuje pogreške prilikom prevoðenja. To se dešava kada primjerice neki od funkcijskih èlanova klase koristi operator usporedbe za tipove kojima je predložak parametriziran. Da bi se takva klasa uspješno instancirala, tip koji joj se prosljeðuje kao parametar mora imati definirane operatore usporedbe.

Sve uobičajene konvencije koje vrijede za obične klase vrijede i za predloške klasa. Tako je moguće neke objekte definirati vanjskima pomoću ključne riječi `extern` ili statičkima pomoću ključne riječi `static`. Moguće je definirati polja objekata, pokazivače i reference na njih, te dodavati kvalifikatore `volatile` i `const`:

```
extern ElementListe<int> vanjski;
ElementListe<Kompleksni> poljeZ[20];
const ElementListe<int> postojaniKositreniInt(0);
ElementListe<Kompleksni> *pok = poljeZ, &ref = poljeZ[0];
```

Iako bi to iz dosadašnjeg izlaganja trebalo biti samo po sebi jasno, klase istog predloška razlièitih parametara predstavljaju potpuno razlièite tipove i nije ih moguæe miješati. Na primjer, nije ih moguæe usporeðivati, pridruživati ili prosljeðivati kroz parametre (osim

ako ne postoje definirani operatori konverzije ili pridruživanja). Dakle, uz gornje deklaracije naredba

```
poljeZ[0] = vanjski;        // pogreška
```

nije sintaktièki ispravna i nema smisla. Ona pokušava međusobno pridružiti dva potpuno razlièita tipa. Uostalom, ako se malo bolje pogleda, gornje pridruživanje niti nema nekog logièkog smisla.

Pojedine instance neke klase definirane predloškom predstavljaju različite tipove te ih nije dozvoljeno miješati.

Može se dogoditi da se predložak treba instancirati unutar definicije drugog predloška. Tada se parametar predloška koji se definira može koristi kao stvarni parametar predloška kojeg se instancira. Pretpostavimo da svakoj varijanti klase `ElementListe` želimo pridružiti funkciju `Ispisi()` koja æe ispisati sadržaj elementa na ekran. Takva funkcija æe se morati definirati predloškom:

```
template <class Tip>
void Ispisi(ElementListe<Tip> &Elem) {
    ElementListe<Tip> *pok = &Elem;
    // ...
}
```

U gornjem primjeru parametar funkcije je zapravo instantacija klase `ElementListe`, slièno kao i deklaracija lokalne varijable `pok`. Pri tome se naziv `ElementListe` parametrizira identifikatorom `Tip`.

Posebnu pažnju treba obratiti na instanciranje predložaka koji imaju izraze kao parametre. Promotrimo to na primjeru klase `OgranicenaLista` koja može sadržavati samo neki određeni maksimalan broj članova. Takva lista može biti deklarirana predloškom

```
template <class Tip, int maxElem>
class OgranicenaLista {
private:
    Tip lista[maxElem];
    int elem_u_listi;
public:
    OgranicenaLista() : elem_u_listi(0) {}
    // ...
};
```

Isti uèinak može se postiæi tako da se broj elemenata proslijedi konstruktoru – deklaracija bi onda izgledala ovako:

```
template <class Tip>
class OgrLista {
private:
    Tip *lista;
    int elem_u_listi;
    int maxElem;
public:
    OgrLista(int mel) : lista(new Tip[mel]),
        maxElem(mel), elem_u_listi(0) {}
    // ...
};
```

Između gornja dva rješenja postoji suštinska razlika. Da bi se ona razumjela, potrebno je sjetiti se da se predlošci instanciraju tako da se identifikatori parametara zamijene stvarnim parametrima po *pretraži-i-zamijeni* principu te svaki put dobivamo zasebnu klasu. To znaèi da æe instantacije

```
OgranicenaLista<int, 5> ol5;
OgranicenaLista<int, 6> ol6;
```

definirati dvije odvojene klase. Svaka od tih klasa imat æe svoj skup funkcijskih èlanova. Dobiveni izvršni program bit æe stoga dulji jer æe sadržavati verziju funkcijskih èlanova za 5 i za 6 elemenata. Ako bi se u programu koristile još liste drugih duljina, dobiveni izvedbeni kôd bi znatno rastao, pogotovo ako je klasa opširna i s mnogo funkcijskih èlanova. Takoðer, parametar `maxElem` mora biti konstantan izraz, odnosno mora biti poznat prilikom prevođenja.

Drugo rješenje nema taj nedostatak: postoji jedna klasa koja se jednom instancira za određeni tip. Duljina liste se određuje prilikom poziva konstruktora. Tim više, duljina ne mora biti konstantan izraz, nego se može izračunati prije poziva konstruktora.

Postavlja se pitanje zašto bi onda netko uopće htio koristiti izraze kao parametre predlošku. Razlog leži u brzini izvođenja. Naime, kod prvog rješenja se alokacija memorije potrebne za spremanje liste obavlja vrlo brzo jer se alocira jedno polje objekata poznate duljine. U drugom slučaju koristi se dinamička dodjela memorije, koja radi dosta sporije. Alokacija preko polja je moguća zato jer je parametar `maxElem` poznat prilikom prevođenja. U drugom slučaju parametar konstruktoru se određuje prilikom izvođenja, pa nije moguće alokaciju obaviti na tako elegantan način.

Ako se neki funkcijski član klase ne koristi u programu za određeni skup argumenata, prevoditelj ga neće generirati (osim u slučaju eksplicitne instantacije – pogledajte sljedeći odsječak). To omogućava da neke operacije budu definirane samo za neke tipove. Primjerice, klasa `Lista` može sadržavati funkcijski član za sortiranje članova u rastući poredak. Pri tome će se za usporedbu koristiti operator < koji mora biti preopterećen za zadani tip. No mnogi tipovi, primjerice `Kompleksni`, nemaju definirane operatore usporedbe, jer se za njih usporedba ne može definirati na neki razuman način. Ako bi prevoditelj prilikom instantacije generirao sve članove predloška klase, tada ne bismo mogli koristiti `Lista<Kompleksni>` jer član za sortiranje ne bi prošao prevođenje.

Zbog toga prevoditelj neće generirati član za sortiranje ukoliko ga programer eksplicitno ne pozove (a tada odgovornost za operator usporedbe pada na njega). Tako je moguće koristiti `Lista<Kompleksni>` bez sortiranja.

Zadatak. *Klasu `Tablica` iz poglavlja 6 o klasama prepravite pomoću predložaka tako ona ne sadržava samo cijele brojeve, nego i bilo koji drugi tip.*

Zadatak. *Klasi `Tablica` dodajte funkcijski član `Pretrazi()`. Kao parametra taj član će uzimati referencu na tip koji se čuva u tablici, a kao rezultat će vraćati redni broj elementa u tablici ili –1 ako parametar nije pronađen. Usporedba podataka će se obaviti pomoću operatora == definiranog u klasi koja opisuje tip koji se čuva u tablici.*

12.3.3. Eksplicitna instantacija predložaka klasa

U odsjeèku 12.2.4 naveli smo razloge zbog kojih je èesto potrebno eksplicitno instancirati neki predložak funkcije Slièno vrijedi i za predloške klasa: moguæe je dati prevoditelju zahtjev da instancira određeni predložak za zadani skup parametara.

Cijela klasa se instancira tako da se iza ključnih riječi `template class` navede naziv klase s parametrima:

```
template class Lista<Kompleksni>;
```

Ovime će se instancirati svi funkcijski članovi klase Lista s tipom Kompleksni kao parametrom. Osim instanciranja cijele klase, moguće je instancirati samo željeni funkcijski član klase:

```
template void Lista<Vektor>::UgurajClan(Vektor, int);
```

12.3.4. Specijalizacije predložaka klasa

Poneki funkcijski èlanovi predloška klase mogu biti neadekvatni za neke konkretne tipove koji se mogu proslijediti predlošku kao parametar. U tom sluèaju moguæe je definirati specijalizaciju funkcijskog èlana koja æe precizno definirati naèin na koji dotièni èlan mora biti obrađen.

Na primjer, ako bismo klasu `ElementListe` parametrizirali tipom `char *`, konstruktor klase ne bi djelovao ispravno. Konstruktor izgleda ovako:

```
template <class Tip>
inline ElementListe<Tip>::ElementListe(const Tip &elem) :
    vrij(elem), prethodni(NULL), sljedeci(NULL) {}
```

Za tip `char *` taj konstruktor æe lokalni objekt `vrij` inicijalizirati pokazivaèem na znakovni niz, što nije ispravno. Ono što bismo htjeli jest alocirati zasebni memorijski prostor za proslijeđeni niz te kopirati proslijeđeni niz u to mjesto. U takvom sluèaju smo prisiljeni napraviti specijalizaciju predloška klase.

Specijalizacija pojedinog funkcijskog člana predloška se definira tako da se iza naziva člana umjesto formalnih navedu stvarni tipovi na koje se specijalizacija odnosi:

```
#include <string.h>

inline ElementListe<char *>::ElementListe(char * const & elem)
        : vrij(new char[strlen(elem) + 1]), prethodni(NULL),
          sljedeci(NULL) {
    strcpy(vrij, elem);
}
```

U gornjem primjeru parametar konstruktoru je `char * const &`. Nije sasvim oèito, no radi se o referenci na konstantan pokazivaè na znak. Nije bilo dovoljno navesti `char *`, `const char *` niti `char const *`. Evo i zašto: konstruktor deklariran u klasi kao parametar ima referencu na konstantan tip. Tip specijaliziranog konstruktora mora tome odgovarati – zato nije dovoljno samo prenijeti pokazivaè na znak, nego referencu na pokazivaè. Uz ovakvu dopunu, sada æe se za sve elemente liste koji sadržavaju pokazivaèe na znak koristiti dani konstruktor.

Ponekad implementacija definirana predloškom nije dovoljno efikasna za neke tipove. Također, moguće je da za neki tip klasa treba dodatak javnom sučelju kako bi funkcionirala ispravno. Tada možemo definirati specijalizaciju cijele klase za pojedini tip. Specijalizacija se definira tako da se nakon naziva klase unutar znakova `<>` navede tip za koji se specijalizacija definira. Ona se ne mora poklapati u sadržanim funkcijskim članovima s predloškom klase. Specijalizacija predstavlja jednu zasebnu klasu koja može biti potpuno drukčija od preostalih klasa dobivenih iz predloška. No sa specijalizacijama ne treba pretjerivati.

Ako dotična klasa uistinu predstavlja objekt koji nema sličnosti s predloškom, onda je ispravnije ne definirati ga kao specijalizaciju predloška nego kao neku drugu klasu zasebnog imena.

U našem sluèaju, ako klasa `ElementListe` sadržava znakovne nizove, ona iziskuje dodatni funkcijski èlan – destruktor. Predložak klase ne definira destruktor jer æe svi podatkovni èlanovi objekta biti automatski uništeni. Objekt koji se èuva u listi bit æe automatski uništen pomoæu podrazumijevanog destruktora klase. No u sluèaju znakovnih nizova podrazumijevani destruktor neæe osloboditi zauzetu memoriju. Zbog toga æemo definirati specijalizaciju klase za tip `char *`:

```
class ElementListe<char *> {
private:
    char *vrij;
    ElementListe *sljedeci, *prethodni;
public:
    ElementListe(char *elem);
    ~ElementListe();
    char *DajVrijednost() { return vrij; }
```

```
    void Kopiraj(char *buff);
};

ElementListe<char *>::~ElementListe() {
    delete [] vrij;
}



void ElementListe<char *>::Kopiraj(char *buff) {
    strcpy(buff, vrij);
}

ElementListe<char *>::ElementListe(char *elem) :
            vrij(new char[strlen(elem) + 1]),
            sljedeci(NULL), prethodni(NULL) {
    strcpy(vrij, elem);
}
```

Dodani destruktor æe sada ispravno uništiti objekt jer æe i osloboditi zauzetu memoriju. Po istom principu smo dodali funkcijski èlan `Kopiraj()`. On kopira znakovni niz u memorijski spremnik te je svojstven samo implementaciji za znakovne nizove. Također, kako specijalizirana klasa može definirati sasvim drukèije èlanove, više nije potrebno patiti se s èudnim konstruktorima – konstruktor je sada deklariran prirodno (lat. *declaratio naturalis*).

Specijalizacija predloška se može navesti samo nakon navođenja opæeg predloška. Također, ako se specijalizirani primjerak klase instancira, potrebno je definirati sve funkcijske èlanove eksplicitno.

To znaèi da nije moguæe u gornjem primjeru izostaviti definiciju funkcijskog èlana `DajVrijednost()` i oèekivati da æe prevoditelj primijeniti opæu varijantu funkcije. Specijalizacija mora biti deklarirana u cijelosti i to neovisno o opæem predlošku.

Slično predlošcima funkcija, moguće je definirati i djelomične specijalizacije predloška klase. Na primjer:

```
template <class T1, class T2>
class XX;                        // opći predložak

template <class T>
class XX<T, char>;               // parcijalna specijalizacija
```

Zadatak. *Napišite specijalizaciju klase* `Tablica` *iz zadatka na stranici* ***Error! Bookmark not defined.*** *tako da omogućite stvaranje tablice znakovnih nizova. Za to je potrebno definirati specijalizirani konstruktor kopije, destruktor, članove za dodavanje i pristup znakovnim nizovima te član* `Pretrazi()`*.*

12.3.5. Predlošci klasa sa statièkim èlanovima

Prilikom definiranja predloška klase moguæe je navesti statièke èlanove. U tom sluèaju svaka instanca navedenog predloška æe imati svoj zasebni skup statièkih èlanova. Svaki od tih èlanova se mora inicijalizirati zasebno.

Uzmimo na primjer da želimo brojati ukupan broj listi koje imamo u memoriji. To se jednostavno može učiniti tako da klasi `Lista` dodamo statički član. U konstruktoru klase potrebno je povećati član za jedan, a u destruktoru klase smanjiti za jedan. Statički član se i prilikom korištenja predložaka klasa može definirati na isti način kao i kod običnih klasa:

```
template <class T>
class Lista {
private:
    ElementListe<T> *glava, *rep;
public:
    static int brojLista;

    Lista();
    ~Lista();
    void Dodaj(T *elt);
    void Brisi(T *elt);
    ElementListe<T> *Pocetak() { return glava; }
    ElementListe<T> *Kraj() { return rep; }
};
```

Konstruktor i destruktor klase æe osim uobièajenih poslova inicijalizacije i deinicijalizacije objekta obavljati posao ažuriranja brojaèa `brojLista`:

```
template <class T>
Lista<T>::Lista() : glava(NULL), rep(NULL) {
    brojLista++;
}

template <class T>
Lista<T>::~Lista() {
    brojLista--;
}
```

Statièki èlanovi se u zaglavlju klase samo deklariraju, no inicijalizirati se moraju izvan klase. To se izvodi ovako:

```
template <class T>
int Lista<T>::brojLista = 0;
```

Smisao gornje naredbe je sljedeæi: "Za svaku klasu napravljenu po predlošku stvori cjelobrojni statièki èlan `brojLista` i inicijaliziraj ga na nulu". Moguæe je takoðer definirati specijalizirane inicijalizacije statièkih èlanova. U sluèaju gornje inicijalizacije

sve æe se liste poèeti brojati od nule. Ako bismo htjeli poèeti brojati liste realnih brojeva od broja 5 (ne ulazeæi u smisao takvog brojanja), to bismo mogli uèiniti na sljedeæi naèin:

```
int Lista<float>::brojLista = 5;
```

Ako postoji, specijalizirana inicijalizacija se uvijek primjenjuje umjesto opæe varijante. Također, specijalizirana inicijalizacija nema izravne veze sa specijalizacijom predloška – sama klasa ne mora imati specijalizirane èlanove, ali za neki tip možemo imati specijaliziranu inicijalizaciju i obrnuto.

Specijalizirani predložak može koristiti opću inicijalizaciju, a opći predložak može koristiti specijaliziranu inicijalizaciju.

Prilikom pristupa statièkim èlanovima klase definirane predloškom potrebno je navesti puno ime klase. Naime, svaka varijanta klase `Lista` imat æe svoj brojaè koji æe brojati samo objekte tog tipa pa je prilikom pristupa potrebno je naznaèiti kojem se brojaèu pristupa:

```
int main() {
    Lista<int> lista1, lista2;
    Lista<char *> lista3;
    cout << "Broj cjelobrojnih lista: " <<
        Lista<int>::brojLista << endl;
    cout << "Broj lista znakovnih nizova: " <<
        Lista<char *>::brojLista << endl;
    return 0;
}
```

Prilikom izvođenja, u prvom retku æe se ispisati broj 2 jer u memoriji postoje dvije cjelobrojne liste, dok æe se u drugom retku ispisati broj 1 jer postoji samo jedna lista nizova znakova. Pristup

```
cout << Lista::brojLista << endl;   // pogreška
```

nije ispravan jer nije jasno kojem se brojaèu pristupa.

Zadatak. *Klasu `Tablica` proširite statičkim članom koji će pamtiti ukupan broj članova u svim tablicama u programu. Uputa: za to je potrebno promijeniti i konstruktor tablice i članove za promjenu veličine.*

12.3.6. Konstantni izrazi kao parametri predložaka

Formalni parametar predloška, osim tipa može biti i konstantni izraz. Na primjer, moguæe je definirati klasu `Spremnik` koja definira memorijski međuspremnik za èuvanje podataka prilikom komunikacije s uređajima u raèunalu, kao što su disk ili

komunikacijski prikljuèak. Spremnik može biti razlièitih duljina, pa æemo tu klasu realizirati pomoæu predloška koji æe kao parametar imati cjelobrojni izraz. Parametar predloška æe odreðivati duljinu spremnika:

```
template <int duljina>
class Spremnik {
private:
    char podrucje[duljina];
public:
    void Puni(char *pok, int dulj);
    void Citaj(char *pok, int dulji);
};
```

Parametar `duljina` mora biti poznat prilikom prevoðenja. To znaèi da on mora biti sastavljen od samih konstanti. Prilikom prevoðenja sve pojave identifikatora `duljina` se zamjenjuju vrijednošæu navedenog izraza. Prilikom instantacije klase nije dozvoljeno parametre definirati varijablama, jer je njihova vrijednost poznata tek prilikom izvoðenja. Na primjer:

```
Spremnik<20> s1;
Spremnik<10 + 10> s2;
Spremnik<10 * 2> s3;
Spremnik<30> s4;
```

Gornje instantacije su ispravne. Generiraju se dvije klase, `Spremnik<20>` i `Spremnik<30>`. Prve tri deklaracije rezultiraju instantacijom samo jedne klase, jer izrazi u parametru imaju istu vrijednost. Posljednja deklaracija stvara zasebnu klasu, jer izraz u parametru ima razlièitu vrijednost od prva tri.

Važno je razumjeti da, iako su klase `Spremnik<20>` i `Spremnik<30>` slične sa stanovišta implementacije, one predstavljaju dvije zasebne i odvojene klase. Ne postoji nikakvo njihovo međusobno srodstvo. Pokazivač na objekt jedne klase se ne može implicitno pretvoriti u pokazivač na objekt druge klase (eksplicitna pretvorba uz dodjelu tipa je moguća uvijek i između potpuno nekompatibilnih tipova, ali na vlastitu odgovornost). Također, svaka od tih dviju klasa ima zaseban skup funkcijskih članova. Ako je klasa dugačka, mnoge instantacije za različite vrijednosti parametra `duljina` će generirati mnogo funkcijskih članova te će izvršni kôd biti dugačak.

Donja instantacija nije ispravna, jer izraz koji se stavlja kao parametar nije poznat prilikom prevođenja, nego tek samo prilikom izvođenja:

```
int n = 5;
Spremnik<n * 20> uzas;                  // pogreška
```

Tip izraza koji je stavljen na mjesto parametra mora biti potpuno jednak tipu navedenom u deklaraciji predloška – nema konverzije:

```
Spremnik<6.7 * 6.4> bez_konverzije;     // pogreška
```

U gornjem primjeru prevoditelj æe prijaviti pogrešku, jer nema konverzije iz realnog u cjelobrojni tip. To se može urediti pomoæu eksplicitne dodjele tipa:

```
Spremnik<(int)(6.7 * 6.4)> sada_radi;
```

12.3.7. Predlošci i ugniježđeni tipovi

Razmotrimo detaljnije implementaciju klasa `ElementListe` i `Lista`. Objekti klase `ElementListe` nemaju smisla osim u kontekstu klase `Lista`. Teško je zamisliti neki naèin na koji bi glavni program mogao imati koristi od klase `ElementListe`, pa bi bilo vrlo pogodno onemoguæiti stvaranje objekata klase iz glavnog programa.

No u pročišćavanju strukture programa može se ići i dalje. Bilo bi vrlo zgodno ukloniti identifikator `ElementListe` iz javnog područja imena. Ima smisla deklarirati klasu `ElementListe` ugniježđenu u klasu `Lista`.

U staroj varijanti C++ jezika koju još i danas mnogi prevoditelji podržavaju to nije bilo moguće – predlošci su mogli biti definirani samo u globalnom području, a ne unutar klase. Na primjer, sljedeća deklaracija je neispravna:

```
template <class Tip1>
class Lista {
private:
    template <class Tip2>   // pogreška u starom C++ jeziku
    class ElementListe {
        // ...
    };
};
```

No u našem sluèaju to i ne bi predstavljao neki problem: klasa `ElementListe` mora biti parametrizirana istim tipom kao i klasa `Lista`. Dakle, svaka `Lista` ima odgovarajuæi `ElementListe`. Dotièni problem se može riješiti na sljedeæi naèin:

```
template <class Tip>
class Lista {
private:

    class ElementListe {
    public:
        Tip vrij;
        ElementListe *prethodni, *sljedeci;
    };


    ElementListe *glava, *rep;
};
```

Ako su ugniježđeni tipovi javni, može im se pristupati i iz okolnog programa. Pri tome je potrebno potpuno navesti put do klase kojoj se želi pristupiti. Svaka varijanta klase `Lista` definira jednu klasu `ElementListe`. Tako je moguæe koristiti tipove

```
Lista<int>::ElementListe
Lista<char *>::ElementListe
```

Predložak klase također može imati ugniježđena pobrojenja. Njima se isto tako može pristupiti iz okolnog podruèja pod uvjetom da se navede puna staza do tipa. Na primjer, klasa `Spremnik` može definirati pobrojenja koja opisuju redni broj zadnjeg elementa u spremniku te kritiènu velièinu spremnika koja je jednaka 75% ukupne velièine:

```
template <int duljina>
class Spremnik {
private:
    char podrucje[duljina];
public:
    enum {kriticna = (int)(0.75 * duljina),
        zadnji = duljina - 1, maksimum = 1000};
    void Puni(char *pok, int dulj);
    void Citaj(char *pok, int dulji);
};
```

Sada je moguæe iz vanjskog programa pristupati pobrojenjima, ali tako da se navede ukupna staza do identifikatora:

```
cout << Spremnik<20>::kriticna << endl;
cout << Spremnik<99>::zadnji << endl;
```

Punu stazu je potrebno navesti i prilikom pristupa identifikatoru `maksimum`, bez obzira na to što se vrijednost tog identifikatora ne mijenja u razlièitim instantacijama klase:

```
cout << Spremnik::maksimum << endl;     // pogreška
cout << Spremnik<20>::maksimum << endl; // ispravno
```

12.3.8. Ugniježđeni predlošci

Posljednja verzija C++ jezika podržava ugnježđivanje predložaka. Moguæe je definirati predložak neke klase unutar klase ili unutar predloška klase:

```
template <class T1>
class X {
public:
    template <class T2>
    class Y {
        void Funkcija();
        T1 *pok;
```

```
    };
};
```

Funkcijski èlan `Funkcija()` je sada parametriziran s dva tipa: `T1` i `T2`. Njegova definicija izvan klase izgleda ovako:

```
template <T1> template <T2>
void X<T1>::Y<T2>::Funkcija() {
    // ...
}
```

Jedna instanca klase `Y` ovisi i o parametru klase `X` i o parametru klase `Y`:

```
X<char>::Y<Kompleksni> obj;
```

Neuki programer bi mogao pomisliti zašto je `Y` parametriziran s `T1`; na kraju, `Y` predstavlja tip neovisan od `X`, s tom razlikom što je njegovo podruèje ugniježðeno u podruèje klase `X`. No valja se prisjetiti da su formalni argumenti u definiciji predloška vidljivi kroz cijelu definiciju do njenog kraja. To znaèi da je unutar definicije klase `Y` dozvoljeno koristiti tip `T1` – na kraju, to je i uèinjeno za èlan `pok`. Stoga æe izgled klase `Y` definitivno, osim samo o `T2`, ovisiti i o `T1`.

Osim deklaracija predložaka klase unutar drugih klasa i predložaka klasa, veliku primjenu ima i deklaracija predložaka funkcijskih članova unutar klasa i predložaka klasa. To svojstvo je vrlo važno za definiranje različitih funkcija konverzije između klasa.

Preradit ćemo klasu `Kompleksni` tako da bude parametrizirana tipom koji određuje preciznost kompleksnog broja:

```
template <class Tip>
class Kompleksni {
private:
    Tip x, y;
public:
    void PostaviXY(Tip a, Tip b) { x = a; y = b; }
    Tip DajX() { return x; }
    Tip DajY() { return y; }
};
```

Neka sada imamo funkciju `sqrt()` koja raèuna korijen iz kompleksnog broja. Ona sada mora biti parametrizirana odreðenom instancom predloška – uzet æemo da kao parametar uzima `Kompleksni<double>` zato jer se time postiže najveæa preciznost:

```
Kompleksni<double> sqrt(Kompleksni<double>);
```

No što se dešava ako želimo funkciji kao parametar proslijediti `Kompleksni<float>`? To ne možemo uèiniti direktno, jer su `Kompleksni<float>` i `Kompleksni<double>` dva razlièita tipa između kojih nema ugraðenih pravila konverzije. Konverziju moramo

sami dodati, i to tako da klasi dodamo predložak konstruktora. On æe biti parametriziran nekim drugim tipom Tip2 te æe konvertirati objekt Kompleksni<T2> u objekt Kompleksni<Tip>:

```
template <class Tip>
class Kompleksni {
    // ...
public:
    template <class Tip2>
    Kompleksni(const Kompleksni<Tip2>& ref) :
            x(ref.x), y(ref.y) {}
    // ...
};
```

Taj konstruktor æe omoguæi konverziju Kompleksni<Tip2> u Kompleksni<Tip> samo ako postoji konverzija tipa Tip2 u Tip. Ako ne postoji, konstruktor neæe biti generiran ako ga korisnik ne pozove, te neæe doæi do pogreške prilikom prevoðenja.

Postoji važno ograničenje na funkcijske članove definirane predloškom: oni ne mogu biti virtualni. Na primjer:

```
class X {
public:
    template <class T>
    virtual void f(T&) = 0;      // neispravno
};

class Y : public X {
public:
    template <class T>
    virtual void f(T&);          // neispravno
};
```

Problem je u tome što bi za svaki poziv funkcije f() s razlièitim parametrom bilo potrebno dodati stavku u virtualnu tabelu klase X. Kako se ti pozivi s razlièitim parametrima mogu protezati kroz više datoteka izvornog kôda, jedino povezivaè može sagledati kompletnu situaciju. To unosi dodatne komplikacije, jer bi tada povezivaè trebao ponovo proæi kroz cijeli kôd i promijeniti adresiranje virtualne tabele. Realizirati takav mehanizam bi bilo vrlo složeno: jednostavnije je zabraniti predloške virtualnih funkcijskih èlanova. Meðutim, to i nije neko veliko ogranièenje, jer se predlošci funkcijskih èlanova najèešæe koriste za definiranje konverzije.

Predlošci funkcijskih èlanova ne mogu biti virtualni.

Nije dozvoljeno stvarati predloške lokalnih klasa (klasa deklariranih unutar funkcija ili funkcijskih èlanova).

12.3.9. Predlošci i prijatelji klasa

Postoje tri naèina na koji jedan predložak klase možemo uèiniti prijateljem nekog drugog predloška.

1. Predložak klase se deklarira kao prijatelj neke druge funkcije ili klase (ne predloška funkcije ili klase). Takvom deklaracijom funkcija ili klasa postaje prijateljem svih mogućih klasa koje se dobiju iz predloška:

```
class A {
    void FClan();
};


class B {
public:
    void FClan();
};

void Funkcija();

template <class Tip>
class C {
    friend class A;
    friend void Funkcija();
    friend void B::FClan();
    // ...
};
```

   Ovaj sluèaj je razmjerno jednostavan: klasa `A`, funkcija `Funkcija()` i funkcijski èlan `FClan()` klase `B` æe biti prijatelji svih klasa koje æe nastati iz predloška klase `C`.

2. *Vezano prijateljstvo* (engl. *bound template friendship*) se ostvaruje kada se svaka klasa koja proizlazi iz predloška veže s točno jednom klasom nekog drugog predloška:

```
template <class Tip>
class Lista {
    // ...
};

template <class Tip>
class Red {
public:
    void Dodaj(Tip *);
};

template <class Tip>
void Ispisi(Tip *) {
    // ...
}
```

```
template <class Tip>
class ElementListe {
    friend class Lista<Tip>;
    friend void Red<Tip>::Dodaj(Tip *);
    friend void Ispisi(Tip*);
    // ...
};
```

U ovom sluèaju se svakoj klasi ElementListe dodjeljuju po jedna klasa Lista, funkcijski èlan Dodaj() klase Red i funkcija Ispisi() parametrizirani istim tipom kojim je parametrizirana i klasa ElementListe. To znaèi da æe Lista<char> imati pristup svim privatnim i zaštiæenim èlanovima klase ElementListe<char>, ali ne i èlanovima klase ElementListe<int>.

3. *Nevezano prijateljstvo* (engl. *unbound template friendship*) se ostvaruje kada se svaka klasa koja proizlazi iz predloška veže sa svim klasama nekog drugog predloška:

```
template <class Tip>
class ElementListe {
    template <class T>
        friend class Lista<T>;
    template <class T>
        friend void Red<T>::Dodaj(T *);
    template <class T>
        friend void Ispisi(T *);
};
```

U ovom æe sluèaju svaka klasa Lista imati pristup privatnim i zaštiæenim èlanovima svih klasa ElementListe.

Zadatak. *Napišite funkciju Zbroji() koja će kao parametar imati tablicu definiranu klasom Tablica. Ta funkcija mora biti prijatelj klase Tablica te će direktnim pristupom implementaciji klase zbrajati pojedine članove tablice. Za zbrajanje koristite operator + definiran u klasi koja opisuje tip koji se čuva u tablici.*

12.3.10. Predlošci i nasljeðivanje

Predlošci klasa mogu se koristiti kao osnovne klase te za izgradnju parametrizirane hijerarhije klasa. Postoje tri osnovna naèina nasljeðivanja.

Prvi je način kada predložak klase služi kao osnovna klasa za izvođenje konkretne klase koja nije predložak. U tom slučaju osnovna klasa mora biti parametrizirana konkretnim tipovima, te se zapravo nasljeđuje jedna konkretna instanca nekog predloška.

Na primjer, definirajmo klasu SkupRijeci koja opisuje objekt koji čuva niz riječi, može ga pretraživati i sortirati. Takav objekt bi bio od koristi u programu koji održava rječnik. Klasu SkupRijeci možemo izvesti nasljeđivanjem iz klase Lista<char *>

koja nam daje mehanizam za održavanje skupa. Klasa `SkupRijeci` će samo dodati nove funkcijske članove za pretraživanje, sortiranje i slično. Takvo nasljeđivanje će se postići na sljedeći način:

```
class SkupRijeci : Lista<char *> {
    // ...
};
```

Druga moguænost jest da se predložak klase izvodi iz neke obiène klase. Na primjer, moguæe je definirati apstraktnu baznu klasu `Kontejner` koja æe definirati opæa svojstva objekta koji sadržava druge objekte, na primjer sposobnost odreðivanja broja elemenata u kontejneru, pražnjenje kontejnera i slièno. Klasa `Lista` æe zatim naslijediti tu klasu te æe definirati navedene operacije u skladu sa svojom implementacijom. No kako klasa `Lista` može sadržavati objekte razlièitog tipa, ona mora biti definirana predloškom. Takvo izvoðenje se može provesti ovako:

```
class Kontejner {
public:
    virtual int BrojElemenata()=0;
    virtual void Prazni()=0;
};

template <class Tip>
class Lista : public Kontejner {
    // ...
};
```

Nakon ovakvog nasljeðivanja sve varijante klase `Lista` æe imati po jedan podobjekt klase `Kontejner` koji nije parametriziran predloškom.

Treća mogućnost nasljeđivanja je nasljeđivanje u kojem se predložak klase izvodi iz nekog drugog predloška. Na primjer, klasa `Lista` definira opća svojstva objekta koji čuva druge objekte. No može nam zatrebati i klasa `Stog` koja će biti slična klasi `Lista`, s tom razlikom što će se dodavanje i uklanjanje elemenata uvijek obavljati na početku liste. Ta klasa također mora biti realizirana kao predložak, jer će taj stog čuvati elemente različitih tipova. U tom slučaju predložak klase `Lista` može poslužiti za izvođenje predloška klase `Stog`, što će se obaviti ovako:

```
template <class Tip>
class Stog : public Lista<Tip> {
    // ...
};
```

Iz svake varijante klase `Lista` izvodi se po jedna varijanta klase `Stog`. Prilikom korištenja osnovnih klasa definiranih predloškom valja se voditi sljedeæim pravilom: ako se kao osnovna klasa koristi predložak klase, on mora imati listu parametara. Naprotiv, izvedena klasa nikada nema listu parametara; ona se specificira korištenjem

kljuène rijeèi `template`. U gornjem primjeru je klasa `Lista` parametrizirana opæim tipom `Tip` èime se prevoditelju daje na znanje da se izvoðenje obavlja iz predloška. No mogli bismo definirati i predložak klase `Stog` koja se izvodi iz toèno odreðene varijante osnovne klase `Lista`, primjerice klase `Lista<char>`. U tom sluèaju æe se opæi parametri osnovne klase zamijeniti stvarnim:

```
template <class Tip>
class Stog : Lista<char> {
    // ...
};
```

Ovaj sluèaj je identièan prvom, s tom razlikom što se kao osnovna klasa koristi jedna varijanta predloška.

Zadatak. *Realizirajte klasu `Stog` pomoću klase `Tablica`. Stog je podatkovna struktura kod koje je članove moguće dodavati i čitati samo na vrhu stoga. Uputa: klasa `Stog` će biti predložak te će nasljeđivati predložak klase `Tablica`.*

12.4. Mjesto instantacije

Prilikom korištenja predložaka, bilo funkcija bilo klasa, postoje popratne pojave koje u prvi mah nisu oèite, a mogu prouzroèiti velike glavobolje programerima koji ih ne razumiju. Problem je definirati *mjesto instantacije* (engl. *point of instantiation*). Odmah napominjemo da su primjeri navedeni u ovom odsjeèku dosta nastrani, no to je stoga jer se radi o primjerima sa samo nekoliko linija kôda. U velikim programima vrlo je moguæe doæi u situacije opisane u ovim poglavljima nehotice: perverzija tada prelazi u realnost.

Promotrimo sljedeći primjer deklaracije i korištenja predloška funkcije `OpciExp()` za računanje eksponencijalne funkcije preko reda potencija. Za one koji su "markali" taj sat analize, eksponencijalna funkcija se može razviti u sljedeći red:

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \mathrm{K}$$

Iako razlog za to možda nije odmah oèit, funkcija je realizirana predloškom – argument predloška odreðuje tip za koji se eksponencijalna funkcija raèuna. Tako se jednim potezom može dobiti više funkcija za raèunanje eksponencijalne funkcije realnog broja, ali i kompleksnog broja, pa èak i matrice (u naprednoj analizi moguæe je definirati eksponencijalnu funkciju za matrice, i to upravo preko gore napisanog beskonaènog reda). Kako ipak nemamo cijelu vjeènost na raspolaganju da bismo pozbrajali svih beskonaèno èlanova, dodatni parametar æe nam biti cijeli broj koji æe odreðivati broj iteracija:

```
double faktorijel(double);
double NaPotenciju(double baza, int eksponent);
```

```
template <class T>
T OpciExp(T x, int brKoraka) {
    T rez = 0;
    for (int i = 0; i <= brKoraka; i++)
        rez += NaPotenciju(x, i) / faktorijel(i);
    return rez;
}
```

No prevoditelj prilikom prevoðenja predloška uopæe ne zna za koji æe tip predložak biti iskorišten. Jedino što se zna jest da se tip mora moæi inicijalizirati nulom, mora se moæi dijeliti faktorijelama te meðusobno zbrajati. Takoðer, mora postojati funkcija `NaPotenciju()` za taj tip.

Očito se povezivanje funkcije `NaPotenciju()` te aritmetičkih operatora mora odgoditi do mjesta na kojemu se predložak instancira, tek kada se definira tip `T`. No što je primjerice, s funkcijom `faktorijel()`? Mogli bismo reći da ćemo sve identifikatore povezati na onom mjestu na kojemu se predložak instancira. To može ponekad biti pogubno, ako bismo u nastavku definirati i `long` varijantu funkcije `faktorijel()` (za neupućene, faktorijele od realnih brojeva se definiraju preko gama-funkcija):

```
// Ovaj kôd slijedi iza prethodne deklaracije

long NaPotenciju(int, int);
long faktorijel(int);

int main() {
    double e = OpciExp(1, 20);
    cout << e << endl;      // mogli bismo se iznenaditi
    return 0;
}
```

U gornjem primjeru smo pozvali funkciju `OpciExp()` za najobièniji `int`, no uz vezanje identifikatora na mjestu instantacije neæemo dobiti ispisan broj *e*. Evo i zašto: na mjestu instantacije definirane su cjelobrojne funkcije `faktorijel()` i `NaPotenciju()`. U samoj funkciji dijeljenje æe se svesti na cjelobrojno dijeljenje te æemo imati sasvim pogrešan raèun (greška u raèunu nastaje i zbog tipa pomoæne varijable `rez` i tipa povratne vrijednosti, ali to bi se moglo ispraviti tako da se uvede dodatni tip predloška koji æe se odreðivati sporni tip). Oèito jest da nema idealnog rješenja: imat æemo problema i ako se odluèimo vezivati identifikatore na mjestu deklaracije i na mjestu instantacije.

Kako bi se izbjegle ovakve nedoumice, C++ standard uvodi lukavo rješenje. On kaže da će se identifikatori koji ne ovise o parametru predloška vezati na mjestu deklaracije, dok će se identifikatori koji ovise o tipu predloška vezati na mjestu instantacije.

To znači da će se, u gornjem slučaju, poziv funkcije `faktorijel()` uvijek vezati za `faktorijel(double)`, jer je samo ta verzija funkcije definirana na mjestu deklaracije predloška. Funkcija `faktorijel(int)` je deklarirana iza, pa se niti ne

uzima u obzir. Time se sprečava nehotično “skupljanje smeća” iz okolnog područja prilikom instantacije.

Naprotiv, identifikatori koji su vezani za parametar predloška vežu se na mjestu instantacije. To znači da ako želimo pozvati `OpciExp()` za kompleksni broj, tada klasu `Kompleksni`, operatore za zbrajanje i dijeljenje te funkciju za potenciranje kompleksnog broja `NaPotenciju(Kompleksni, int)` možemo deklarirati bilo gdje u kôdu, ali obavezno prije korištenja funkcije `OpciExp()`.

Standard jezika specificira točno što znači biti vezan za parametar predloška. Primjerice, funkcija ovisi o parametru predloška ako njen parametar ovisi o parametru predloška. Tako funkcija `NaPotenciju()` ovisi o parametru jer joj je argument `x` zapravo tipa `T`. Prilikom korištenja predložaka, u “normalnim” programima korisnik u principu o tome ne mora voditi računa, jer se jezik ponaša intuitivno. No ako želite profurati neki svoj hakeraj, tada ćete se time itekako pozabaviti.

12.5. Realizacija klase `Lista` predloškom

I na kraju, evo kompletne realizacije klase `Lista` pomoæu predložaka. Implementacija je slièna onoj iz poglavlja o nasljeðivanju, s neznatnim razlikama. Prvo i osnovno, pojedini èlanovi liste se ne identificiraju preko pokazivaèa, nego preko cijelog broja. Prvi èlan (onaj na kojeg pokazuje `glava`) ima indeks 1. Takoðer, za izravan pristup pojedinom èlanu liste dodan je preoptereæeni operator `[]`.

```
template <class Tip>
class Lista {
private:

    class ElementListe {
    private:
        Tip vrij;
        ElementListe *prethodni, *sljedeci;
    public:
        ElementListe *Prethodni() { return prethodni; }
        ElementListe *Sljedeci() { return sljedeci; }
        void StaviPrethodni(ElementListe *pret)
                { prethodni = pret; }
        void StaviSljedeci(ElementListe *sljed)
                { sljedeci = sljed; }
        ElementListe(const Tip &elem) : prethodni(NULL),
                                        sljedeci(NULL),
                                        vrij(elem) {}
        Tip &DajVrijednost() { return vrij; }
    };

    ElementListe *glava, *rep;
public:
    Lista() : glava(NULL), rep(NULL) {}
    ElementListe *AmoGlavu() { return glava; }
```

```
    ElementListe *AmoRep() { return rep; }
    void UgurajClan(Tip pok, int redBr);
    void GoniClan(int koga);
    Tip &operator [](int ind);
};

template <class Tip>
void Lista<Tip>::UgurajClan(Tip pok, int redBr) {
    ElementListe *elt = new ElementListe(pok);
    ElementListe *izaKojeg = NULL;
    if (redBr) {
        izaKojeg = glava;
        redBr--;
        while (redBr) {
            izaKojeg = izaKojeg->Sljedeci();
            redBr--;
        }
    }
    // da li se dodaje na početak?
    if (izaKojeg != NULL) {
        // ne dodaje se na početak.
        // da li se dodaje na kraj?
        if (izaKojeg->Sljedeci() != NULL)
            // ne dodaje se na kraj
            izaKojeg->Sljedeci()->StaviPrethodni(elt);
        else
            // dodaje se na kraj
            rep = elt;
        elt->StaviSljedeci(izaKojeg->Sljedeci());
        izaKojeg->StaviSljedeci(elt);
    }
    else {
        // dodaje se na početak
        elt->StaviSljedeci(glava);
        if (glava != NULL)
            // da li već ima članova u listi?
            glava->StaviPrethodni(elt);
        glava = elt;
    }
    elt->StaviPrethodni(izaKojeg);
}

template <class Tip>
void Lista<Tip>::GoniClan(int koga) {
    ElementListe *pok = glava;
    koga--;
    while (koga) {
        pok = pok->Sljedeci();
        koga--;
    }
    if (pok->Sljedeci() != NULL)
```

```
        pok->Sljedeci()->StaviPrethodni(pok->Prethodni());
    else
        rep = pok->Prethodni();
    if (pok->Prethodni() != NULL)
        pok->Prethodni()->StaviSljedeci(pok->Sljedeci());
    else
        glava = pok->Sljedeci();
    delete pok;
}

template <class Tip>
Tip &Lista<Tip>::operator [](int ind) {
    ElementListe *pok = glava;
    ind--;
    while (ind) {
        pok = pok->Sljedeci();
        ind--;
    }
    return pok->DajVrijednost();
}
```

13. Imenici

Slava i ništavilo imena.

Lord Byron (1788-1824)

U ovom poglavlju æe biti objašnjen važan C++ novitet – *imenici* (engl. *namespaces*). Oni su u standard uvedeni razmjerno kasno, a služe prvenstveno kao pomoæ programerima prilikom razvoja složenih programa. Pomoæu njih je moguæe identifikatore pojedinih klasa, funkcija i objekata upakirati u jednu cjelinu koja se zatim na proizvoljnom mjestu u programu može uključiti i iskoristiti.

13.1. Problem podruèja imena

Mnoga informatièka poduzeæa u svijetu su se specijalizirala ne za izradu gotovih programskih rješenja, veæ za pisanje biblioteka koje rješavaju samo specifièan segment nekog problema. Te biblioteke zatim drugi programeri ugraðuju u svoje programe. Kako jedan program može koristiti nekoliko razlièitih biblioteka razlièitih proizvoðaèa, sasvim su moguæe situacije gdje oba proizvoðaèa u svojim bibliotekama koristite iste identifikatore za svoje funkcije i klase.

To je slučaj sa često korištenim simbolima: na primjer, mnoge biblioteke koje obavljaju poslove vezane s diskovnim sustavom računala vrlo će vjerojatno deklarirati funkciju `Read()` za čitanje podataka. Obje biblioteke će imati svoje zasebne datoteke zaglavlja, koje kad se uključe u glavni program ne mogu funkcionirati ispravno: obje deklariraju isti identifikator:

```
bibl1.h
```

```
int Read(int);
```

```
bibl2.h
```

```
void Read();
```

```
glavni.cpp
```

```
#include <bibl1.h>
#include <bibl2.h>  // problemi: redeklaracija imena Read
```

Zbog gore navedenog razloga mnoge su programerske kuæe nazivima klasa i funkcija u biblioteci davali vrlo duga imena kako bi se smanjila vjerojatnost da neka druga biblioteka koristi isto ime. No za klase koje se vrlo èesto koriste nije praktièno prilikom svakog korištenja utipkavati dugaèko ime.

Gore opisani problem se često u literaturi naziva *zagađenjem globalnog područja* (engl. *global scope pollution*). Njegovo rješenje je u C++ jezik uvedeno u obliku imenika, čime je učinjen kompromis između oprečnih zahtjeva koji se postavljaju na područje imena.

13.2. Deklaracija imenika

Identifikatori klasa, funkcija i drugih elemenata C++ jezika se mogu smjestiti u imenike te ih se na taj naèin može ukloniti iz globalnog podruèja imena. Deklaracija imenika se obavlja kljuènom rijeèi `namespace` iza koje se navodi naziv imenika. U vitièastim zagradama se navode deklaracije i definicije:

```
namespace RadiSDiskom {
    int duljina;
    int Citaj(char *buf, int dulj);
    int Pisi(char *buf, int dulj);
    class Datoteka {
        // ovdje ide deklaracija klase
    };
    class Podaci;
    const int samoCitanje = 0x20;
}
```

Iza deklaracije imenika se ne stavlja toèka-zarez.

Unutar deklaracije imenika moguæe je deklarirati, ali i definirati identifikatore. U gornjem primjeru funkcija `Citaj()` je samo deklarirana, dok je klasa `Datoteka` i definirana. Funkcija `Citaj()` se može definirati izvan imenika tako da se navede puni naziv pomoæu operatora za razluèivanje imena:

```
int RadiSDiskom::Citaj(char *buf, int dulj) {
    // definicija
}
```

Klasa `Podaci` je unutar imenika samo deklarirana unaprijed, pa bi njena definicija izgledala ovako:

```
class RadiSDiskom::Podaci {
    // definicija klase
};
```

Za razliku od ostalih elemenata C++ jezika koji ne mogu biti definirani na više mjesta, imenici se mogu definirati u više navrata. Ako, primjerice, kasnije u programu primijetimo da je imeniku `RadiSDiskom` potrebno dodati još i klasu `Direktorij`, to se može uèiniti tako da se jednostavno ponovi deklaracija imenika u kojoj su navedeni novi èlanovi:

```
namespace RadiSDiskom {
    class Direktorij {
        // ...
    };
}
```

Ovakva deklaracija se naziva *proširenjem imenika* (engl. *namespace extension definition*). Dovitljivi èitatelj æe na osnovu toga primijetiti da se definicija èlanova imenika, primjerice funkcije `Pisi()`, može obaviti i u sklopu proširenja:

```
namespace RadiSDiskom {
    int Pisi(char *buf, int dulj) {
        // definicija funkcije
    }
}
```

Imenici se mogu pojaviti samo u globalnom podruèju imena ili unutar nekog drugog imenika. Naziv imenika mora biti jedinstven u podruèju u kojem se pojavljuje.

Evo primjera u kojem je unutar imenika ugniježðen drugi imenik:

```
namespace A {
    int i;
    namespace B {
        int j;      // puni naziv objekta je A::B::j
    }
}
```

Objekti koji su ugniježðeni u više imenika imaju puni naziv koji se dobije tako da se operatorom `::` razdvoje nazivi svih imenika na koje se nailazi na putu do objekta.

Moguće je definirati alternativno ime za neki imenik, tako da se iza ključne riječi `namespace` navede alternativno ime, stavi znak `=` te se navede ime originalnog imenika. U nastavku je deklariran imenik `RSD` koji je alternativno ime za imenik `RadiSDiskom`:

```
namespace RSD = RadiSDiskom;
```

Gornji oblik uvoðenja alternativnog imena je vrlo koristan. Naime, ako koristimo imenike, tada nazivi pojedinih klasa i funkcija više ne predstavljaju problem. No i dalje ostaje problem jedinstvenosti naziva imenika. Zbog toga æe pojedini imenici opet imati dugaèke nazive. Korištenje takvih naziva je vrlo naporno, no zato je moguæe definirati alternativno kratko ime za svaki pojedini imenik iz biblioteke.

Posebna vrsta imenika je *bezimeni imenik* (engl. *nameless namespace*). On se deklarira tako da se u deklaraciji izostavi naziv:

```
namespace {
    int i;
    class X;
}
```

Ovakav imenik æe sadržavati elemente jedinstvene za datoteku u kojoj je dana ovakva deklaracija. Varijabla `i` i klasa `X` se ponašaju kao da su globalni identifikatori, no oni nisu vidljivi izvan datoteke u kojoj su deklarirani. Na taj naèin imenik bez naziva omoguæava deklaraciju "statièkih" identifikatora vidljivih iskljuèivo unutar datoteke gdje su deklarirani.

ANSI standard C++ jezika preporuèuje korištenje imenika bez naziva umjesto korištenja statièke smještajne klase.

13.3. Pristup elementima imenika

Pojedini element imenika se može pozvati tako da se navede puno ime elementa pomoæu operatora `::` za razluèivanje podruèja. Ako je potrebno deklarirati objekt klase `Datoteka`, to se može uèiniti na ovakav naèin:

```
// imenik RadiSDiskom je deklariran u prethodnom odsječku

RadiSDiskom::Datoteka dat;
```

Funkcija `Citaj()` æe se pozvati na sljedeæi naèin:

```
if (!RadiSDiskom::Citaj(pok, 800)) // ...
```

Pojedinom elementu imenika se može pristupiti samo nakon njegove deklaracije unutar imenika. Na primjer:

```
namespace A {
    int i;
}
```

```
void f() {
    A::j = 9;   // pogreška: j još nije dio imenika A
}


namespace A {
    int j;
}

void g() {
    A::j = 10;  // OK: j je prethodno uveden u imenik A
}
```

Èlanovima imenika bez naziva se pristupa bez eksplicitnog navoðenja imenika. Kako bezimeni imenici nemaju ime, nije moguæe eksplicitno navesti èlan iz takvog imenika:

```
int i;
namespace {
    int i;
    int j;
}

void f() {
    j = 0;      // OK: pristupa se elementu imenika
    i = 0;      // pogreška: nije navedeno je li to
                // globalni i ili i iz imenika
    ::i = 0;// OK: globalni i
}
```

Iz imenika nije potrebno navoditi naziv, jer se on podrazumijeva:

```
namespace X {
    int i;
    void f() {
        i++;    // podrazumijeva se član X::i
        // ...
    }
}
```

Ovakav pristup elementima imenika ima tu manu što je èesto potrebno navoditi naziv imenika. Kako bi se to izbjeglo, u C++ jezik je uvedena kljuèna rijeè `using` koja omoguæava uvoðenje identifikatora iz imenika u drugo podruèje imena. Ona dolazi u dvije varijante: deklaracija `using` i direktiva `using`.

13.3.1. Deklaracija `using`

Identifikator iz nekog imenika je moguæe uvesti u neko podruèje imena tako da se iza kljuène rijeèi `using` navede puni naziv identifikatora. Na primjer, ovako æemo u funkciji `UcitajDatoteku()` iskoristiti funkciju `Citaj()` iz imenika `RadiSDiskom`:

```
void UcitajDatoteku() {
    using RadiSDiskom::Citaj;
    char buf[50];
    if (Citaj(buf, 50)) // radi nešto korisno, npr. okopaj vrt
}
```

U gornjem primjeru je pomoæu kljuène rijeèi `using` u podruèje imena funkcije `UcitajDatoteku()` uveden identifikator `Citaj()` iz imenika `RadiSDiskom`. Sada se `Citaj()` može koristiti bez eksplicitnog navođenja imenika.

Deklaracija `using` se može naći i unutar definicije imenika. U tom slučaju će imenik u kojem je deklaracija navedena sadržavati sinonim za član iz nekog drugog imenika. Također, moguće je uvesti i globalni član, tako da se ispred operatora `::` ne stavi nikakvo ime.

```
int i = 0;

namespace X {
    int j = 9;
    using ::i;       // sinonim za globalni i
}

namespace Y {
    using X::i;      // sinonim za globalni i
    using X::j;      // sinonim za j iz imenika X
}

int main() {
    using Y::j;
    cout << Y::i << endl;   // ispisuje 0
    cout << j << endl;      // ispisuje 9
    return 0;
}
```

U deklaraciji `using` se ne navode tipovi èlanova ili povratni tipovi i lista parametara za funkcije. Ako neki naziv pripada preoptereæenoj funkciji, `using` uvodi sinonime za sve preoptereæene funkcije.

Evo primjera:

```
namespace Ispis {
    void Ispisi(int);
```

```
    void Ispisi(char);
    void Ispisi(char *);
    void Ispisi(Kompleksni &);
}

int main() {
    using Ispis::Ispisi;        // odmah su dostupne sve
                                // varijante od Ispisi
    Ispisi(5);
    Ispisi(Kompleksni(5.0, 6.0));
    return 0;
}
```

Pri tome vrijedi imati na umu da `using` deklaracija uvodi u neko podruèje samo identifikatore koji su deklarirani u imeniku do tog mjesta. Ako se u nastavku imenik proširuje još dodatnim preoptereæenim funkcijama, one æe biti ukljuèene u imenik tek nakon deklaracije:

```
namespace Ispis {
    // isto kao i gore
}

int main() {
    using Ispis::Ispisi;
    Ispisi(Vektor(6., 7.)); // pogreška: Ispisi(Vektor&)
                            // na ovom mjestu još nije član
                            // imenika
    return 0;
}

namespace Ispis {
    void Ispisi(Vektor &);  // proširenje imenika je iza
                            // gornje using deklaracije
}
```

U gornjem primjeru, na mjestu gdje je navedena `using` deklaracija imenik `Ispis` još ne sadrži preoptereæenu verziju `Ispisi(Vektor &)`. Zbog toga poziv preoptereæenog èlana neæe uspjeti – prevoditelj æe javiti kako nije pronaðen èlan s odgovarajuæim parametrima. U dijelu kôda koji slijedi nakon proširenja imenika gornji poziv bi bio ispravan.

Deklaracija `using` se može naći i na globalnom području te se tada stvara globalni sinonim za član nekog imenika:

```
namespace Ispis {
    void Ispisi(int a) { cout << "int: " << a << endl; }
}

using Ispis::Ispisi;
```

```
namespace Ispis {
    void Ispisi(char *s) {
        cout << "char *: " << s << endl;
    }
}

int main() {
    Ispisi(5);        // OK: using deklaracija je stvorila
                      // globalni sinonim
    Ispisi("C++");    // pogreška: na mjestu using
                      // deklaracije Ispisi(char *) još nije
                      // bio član imenika
    return 0;
}
```

Deklaracija `using` jest *deklaracija*: ona æe na mjestu gdje je navedena deklarirati èlanove iz odreðenog imenika, i to samo one èlanove koji su u imeniku u tom trenutku. Zbog toga, iako je nakon deklaracije `using`, a prije poziva imenik proširen funkcijom `Ispisi(char *)`, taj èlan neæe biti obuhvaæen deklaracijom `using`, te mu se neæe moæi pristupati bez eksplicitnog navoðenja imenika. Ovakvo ponašanje je suprotno od direktive `using`, koja je objašnjena u sljedeæem odjeljku.

Deklaracija `using` se može naći i u deklaraciji klase. Pri tome ona može referencirati samo član iz osnovne klase koji se tada uvodi u područje izvedene klase. Da bi se član osnovne klase mogao uključiti u područje izvedene klase, on mora biti dostupan.

```
class A {
private:
    int i;
protected:
    int j;
public:
    A(int a, int b) : i(a), j(b) {}
};

class B {
public:
    int k;
};

class C : public A {
public:
    using B::k;       // pogreška: B nije osnovna klasa od A
    using A::i;       // pogreška: i nije dostupan u klasi C
    using A::j;       // OK: j će u klasi C imati javni
                      // pristup
};
```

Pravo pristupa sinonimu odreðeno je mjestom na kojemu se nalazi `using` deklaracija. To znaèi da æe èlan `j` u klasi `C` imati javni pristup, iako je u osnovnoj klasi bio deklariran zaštiæenim:

```
class C : public A {
public:
    C() : A(5, 10) {}
    using A::j;
};

int main() {
    C obj;
    cout << obj.j << endl;      // ispisuje 10
    return 0;
}
```

Deklaracija `using` unutar klase posebno je korisna ako se prisjetimo pravila koje kaže da nasljeðivanje nije isto što i preoptereæenje, te da funkcijski èlan nekog naziva u izvedenoj klasi skriva istoimeni èlan drukèijeg potpisa u osnovnoj klasi:

```
class Osnovna {
public:
    void func(int a) { cout << a << endl; }
};

class Izvedena : public Osnovna {
public:
    void func(char *a) { cout << a << endl; }
};

int main() {
    Izvedena obj;
    obj.func(5);    // pogreška
    return 0;
}
```

U gornjem primjeru `func(char *)` u izvedenoj klasi skriva `func(int)` iz osnovne klase iako su potpisi èlanova razlièiti. Ako bismo prilikom nasljeðivanja htjeli samo preopteretiti neki funkcijski èlan osnovne klase, do sada smo morali ponoviti deklaraciju u izvedenoj klasi i eksplicitno pozvati èlan osnovne klase. Pomoæu deklaracije `using` to više nije potrebno – moguæe je jednostavno ukljuèiti sve dosadašnje preoptereæene funkcije osnovne klase u podruèje imena izvedene klase:

```
class Izvedena : public Osnovna {
public:
    using Osnovna::func;
    void func(char *a) { cout << a << endl; }
};
```

Sada bi se prethodna funkcija `main()` uspješno prevela i pozvala èlan iz osnovne klase. Pri tome æe funkcijski èlanovi osnovne klase zaobiæi (engl. *override*) virtualne funkcijske èlanove osnovne klase. To znaèi da æe, iako se u podruèje izvedene klase uvodi virtualan èlan osnovne klase, u podruèju izvedene klase vrijediti èlan naveden u toj klasi:

```
class B {
public:
    virtual void f(int);
    virtual void f(char);
    void g(int);
    void h(int);
};

class D : public B {
public:
    using B::f;
    void f(int);// OK: D::f(int) će zaobići B::f(int)
    using B::g;
    void g(char);   // OK: ne postoji B::g(char)
    using B::h;
    void h(int);// pogreška: D::h(int) je u konfliktu s
                // B::h(int) jer B::h(int) nije virtualan
};

void f(D* pd) {
    pd->f(1);       // poziva D::f(int)
    pd->f('a');     // poziva B::f(char)
    pd->g(1);       // poziva B::g(int)
    pd->g('a');     // poziva D::g(char)
}
```

13.3.2. Direktiva `using`

Vidjeli smo kako se pomoæu `using` deklaracije može neki identifikator iz nekog imenika uvesti u podruèje imena. No ako neki imenik ima mnogo èlanova koji se èesto ponavljaju, bit æe vrlo zamorno navoditi posebice svako ime kada ga želimo koristiti. Jednim potezom moguæe je kompletan imenik uvesti u podruèje imena, tako da se nakon kljuènih rijeèi `using namespace` navede naziv imenika:

```
void UcitajDatoteku() {
    using namespace RadiSDiskom;
    Datoteka dat;           // poziv klase iz imenika
    char buf[50];
    if (Citaj(buf, 50))     // poziv funkcije iz imenika
    // ...
}
```

Direktiva `using` se može navesti i u globalnom podruèju, èime se svi identifikatori iz imenika uvode u globalno podruèje. Slièno kao i kod `using` deklaracije, identifikator iz imenika se ne može koristiti prije nego što se uvede u imenik:

```
namespace Ispis {
    void Ispisi(int);
    void Ispisi(char);
    void Ispisi(char *);
    void Ispisi(Kompleksni&);
}

using namespace Ispis;

void f() {
    Ispisi(5);                  // OK
    Ispisi(Vektor(5., 6.));     // pogreška: član još nije
                                // ubačen u imenik
}

namespace Ispis {
    void Ispisi(Vektor &);
}

int main() {
    Ispisi(Vektor(5., 6.));     // OK: član je sada dio
                                // imenika
    return 0;
}
```

U gornjem primjeru, iako na mjestu direktive `using` imenik nije sadržavao funkciju `Ispisi(Vektor &)`, èim je funkcija dodana ona je automatski dostupna bez ponovnog navođenja direktive `using`. To je zato jer direktiva `using` ne deklarira èlanove imenika na mjestu gdje je navedena, veæ samo upuæuje prevoditelja da, ako neki identifikator ne može raspoznati, neka dodatno pretraži i podruèje navedenog imenika. Takvo ponašanje je u suprotnosti s deklaracijom `using` iz prethodnog odjeljka, no razlog tome je što deklaracija `using` deklarira èlan, a direktiva `using` ne.

Direktiva `using` može se navesti i u sklopu deklaracije imenika, čime se svi identifikatori iz jednog imenika uključuju u drugi imenik. Pri tome valja biti oprezan, jer ako dva imenika sadrže članove istog naziva, članovi se neće moći jednoznačno razlučiti:

```
namespace X {
    int i = 0;
    int j = 9;
}

namespace Y {
    using namespace X;
```

```
    int i = 2;
}

int main() {
    using namespace Y;
    cout << i << endl;      // pogreška: koji i?
    cout << j << endl;      // OK: ispisuje 9
    return 0;
}
```

U gornjem primjeru nije jasno kojem `i` se želi pristupiti: onom iz imenika `X` ili iz imenika `Y`. U tom sluèaju je potrebno simbol jednoznaèno odrediti navodeæi puno ime:

```
int main() {
    using namespace Y;
    cout << X::i << endl;   // ispisuje 0
    cout << Y::i << endl;   // ispisuje 2
    cout << j << endl;      // ispisuje 9
    return 0;
}
```

Zadatak. *Klase koje definiraju grafičke objekte upakirajte u jedan imenik* `GrafickaBiblioteka`*. Napišite program koji crta grafičke objekte na ekranu tako da poziva navedeni imenik.*

14. Rukovanje iznimkama

Ja nikada ne zaboravljam lica, ali u Vašem slučaju rado ću učiniti iznimku.

Groucho Marx (1895 - 1977)
(Leo Rosten: "People I have Loved, Known or Admired", 1970)

Iznimke (engl. *exceptions*) su situacije u kojima program ne može nastaviti svojim normalnim tokom, veæ je potrebno prekinuti nit izvoðenja te izvoðenje prenijeti na neku posebnu rutinu koja æe obraditi novonastalu situaciju. Upravo to omoguæava posebno svojstvo C++ jezika koje se naziva rukovanje iznimkama.

Iznimka može biti uzrokovana raznim događajima: to može biti posljedica pogreške u radu računala (kao na primjer nedostatak memorije ili nepostojanje neke datoteke na disku) ili jednostavno situacija koja je u programu uzrokovana samim izvođenjem programa. Rutina na koju se izvođenje prenosi može dobiti podatke o iznimci te na taj način poduzeti odgovarajuće akcije, na primjer osloboditi memoriju ili ispisati poruku o pogreški.

14.1. Što su iznimke?

Prilikom izvoðenja programa èesto može doæi do raznih odstupanja od predviðenog tijeka instrukcija. Na primjer, potrebno je alocirati memoriju za neko polje cijelih brojeva koje se kasnije obraðuje u matematièkom dijelu programa. No prilikom pisanja programa ne možemo sa sigurnošæu ustvrditi da æe se dodjela memorije uvijek uspješno izvesti. Naime, ovisno o kolièini memorije u raèunalu, broju aktivnih programa i sliènih okolnosti može se dogoditi da jednostavno dodjela nije moguæa jer nema slobodne memorije. Programer mora zbog toga provjeriti rezultat dodjele prije nego što nastavi s izvoðenjem kako bi osigurao ispravno funkcioniranje programa. U suprotnom, program bi vjerojatno izazvao probleme u radu raèunala, te možda èak i "srušio" raèunalo.

Nadalje, često se mogu pojaviti problemi s neispravnim pristupom korisničkim podacima. Na primjer, C++ jezik ne obavlja provjeru indeksa prilikom pristupa polju. Zbog toga, će se niže navedeni kôd sasvim uspješno prevesti, ali će vrlo vjerojatno prouzročiti probleme prilikom izvođenja:

```
int main() {
    int mat[10];
    mat[130] = 0;       // vrlo opasna instrukcija!
```

```
    return 0;
}
```

Da bismo sprijeèili takve pogreške, moguæe je uvesti klasu `Niz` i njome opisati polje koji provodi provjeru granica:

```
class Niz {
private:
    int duljina;
    int *pokNiz;
    int pogreska;
public:
    Niz(int d) : duljina(d), pokNiz(new int[d]) {}

    int JeLiPogreska() { return pogreska; }
    int& operator[] (int indeks);
};

inline int& Niz::operator[] (int indeks) {
    pogreska = 0;
    if (0 <= indeks && indeks <= duljina)
        return pokNiz[indeks];      // vraćanje reference
    else {
        // Kako vratiti neku suvislu vrijednost?
        pogreska = 1;
        return pokNiz[0];   // Nije dobro rješenje!
    }
}
```

U gornjem primjeru koristi se preoptereæeni operator `[]` za pristup èlanovima niza. On provjerava je li indeks unutar dozvoljenih granica. Ako jest, onda vraæa referencu na željeni èlan niza. Zbog toga što se vraæa referenca, moguæe je smjestiti operator `[]` na lijevu stranu operatora pridruživanja te na taj naèin simulirati uobièajenu sintaksu polja.

Razmotrimo što se dešava u slučaju ako je indeks izvan opsega dozvoljenih vrijednosti. Tada bi bilo potrebno prekinuti normalan tok izvođenja programa i obavijestiti korisnika da je došlo do problema s pristupom članovima polja. Potrebno je pronaći mehanizam kojim će se završiti funkcija i signalizirati pogreška. Jedan od mogućih načina je naveden u primjeru. Svaki niz ima svoj podatkovni član `pogreska`, koji, ako je postavljen na vrijednost 1, signalizira pogrešku prilikom pristupa, a ako je postavljen na 0 ispravan pristup. No potrebno je vratiti i nekakvu vrijednost: funkcija mora završiti `return` naredbom. Zbog toga se vraća referenca na početni objekt.

Ovakvo rješenje nije dobro zbog nekoliko razloga. Promotrimo sljedeći primjer koji će nam prikazati velike nedostatke:

```
Niz a(10);
a[0] = 80;
a[130] = 0;
```

```
if (a.JeLiPogreska())
    cout << "Pogreška u pristupu nizu a." << endl;
```

Veliki nedostatak je potreba eksplicitne provjere ispravnosti nakon svakog pristupa objektu. To može biti zaista zamorno u sluèaju da se piše program koji èesto pristupa poljima. Takoðer, iako je objekt `a` detektirao pogrešku prilikom pristupa, vratio je referencu na poèetni èlan (zato jer operator `[]` mora nešto vratiti) pa je pridruživanje ipak prepisalo poèetni èlan niza. Sada nije moguæe jednostavno izvesti oporavak od pogreške, odnosno nastaviti izvoðenje programa imajuæi na umu da je došlo do neispravnog pristupa. Poèetni èlan je nepovratno izgubljen u bespuæima binarne zbiljnosti!

Rješenje za ovakve i slične probleme nudi nam podsistem C++ jezika koji se naziva mehanizmom za *rukovanje iznimkama* (engl. *exception handling*). Osnovni pojam tog sustava je *iznimka* (engl. *exception*) – događaj u računalu koji onemogućava normalan nastavak izvođenja programa te zahtjeva posebnu obradu. Kada se detektira takva situacija, program *podiže iznimku* (engl. *to raise an exception*). Njegovo izvođenje se prekida, te se iznimka prosljeđuje rutini za *oporavak od iznimke* (engl. *exception recovery routine*).

14.2. Blokovi pokušaja i hvatanja iznimaka

Cijeli C++ program se razbija u niz blokova koji sadržavaju neke operacije koje bi mogle biti problematiène prilikom izvoðenja programa. Ti blokovi se nazivaju *blokovima pokušaja* (engl. *try block*). Oni se specificiraju tako da se navede kljuèna rijeè `try` te se iza nje u vitièastim zagradama navedu problematiène instrukcije. Svaki blok pokušaja mora biti popraæen barem jednim blokom hvatanja, koji se navodi pomoæu kljuène rijeèi `catch`. Prema tome, opæa sintaksa `try`-`catch` blokova ima oblik:

```
try {
    // ovo je blok pokušaja
}
catch (parametar) {
    // ovo je blok hvatanja
}
```

Evo kako bismo problematiènu operaciju pridruživanja smjestili u blok pokušaja:

```
Niz a(10);
try {
    // ovdje dolazi problematičan dio programa,
    // na primjer, naš pristup polju:
    a[130] = 0;
}
// nije još gotovo...
```

Za podizanje iznimke (ponekad se koristi i izraz *bacanje iznimke* – engl. *throwing an exception*) unutar bloka pokušaja koristi se kljuèna rijeè `throw`. Iza nje potrebno je navesti toèno jedan parametar koji æe opisivati tip iznimke koji je nastao. Naime, sustav rukovanja pogreškama omoguæava razlikovanje iznimaka koje se javljaju prilikom izvoðenja. Tako je moguæe definirati razlièite rutine za obradu pojedinih iznimaka. Kao parametar kljuènoj rijeèi `throw` može se navesti objekt, pokazivaè ili referenca na bilo koji ugraðeni ili korisnièki definiran tip podataka koji opisuje iznimku. Na primjer:

```
char *pokZn = new char[10000];
if (!pokZn) throw 10000;
```

Nailaskom na kljuènu rijeè `throw` prilikom izvoðenja prekida se normalan tijek izvoðenja programa te se skaèe na rutinu za obradu pogreške. Parametar naveden kljuènoj rijeèi `throw` bit æe proslijeðen toj rutini kako bi se identificirao razlog zbog kojeg je do iznimke došlo. U gornjem primjeru doæi æe do podizanja iznimke u sluèaju da dodjela memorije nije uspješno obavljena. Kao parametar koji opisuje iznimku navodi se duljina niza koja je izazvala problem. Ova iznimka je cjelobrojnog tipa, zato jer je kao parametar prilikom podizanja iznimke naveden cjelobrojni tip. Moguæe je, primjerice, uèiniti i sljedeæe:

```
if (!pokZn) throw "Ne mogu alocirati memoriju...";
```

U ovom sluèaju iznimka je tipa `const char *`. No èesto se definira poseban korisnièki objekt koji se baca u trenutku podizanja iznimke:

```
class NizoveIznimke {
private:
    char *pogreska;
public:
    NizoveIznimke(char *np) :
        pogreska(new char[strlen(np)+1]) {
            strcpy(pogreska, np); }
    NizoveIznimke(NizoveIznimke &ref) :
        pogreska(new char[strlen(ref.pogreska)+1]) {
            strcpy(pogreska, ref.pogreska); }
    ~NizoveIznimke() { delete [] pogreska; }

    char *Pogreska() { return pogreska; }
};
```

Definirali smo klasu `NizoveIznimke` za opis pogreške koje æe bacati funkcijski èlanovi klase `Niz` u sluèaju da doðe do nepravilnog izvoðenja bilo koje operacije. Ona sadržava tekstualan opis pogreške, a zbog svoje strukture potreban je i konstruktor kopije i destruktor (mehanizam za upravljanje pogreškama èesto kopira i uništava privremene objekte koji rukuju pogreškama). Sada je moguæe operator pristupa napisati na sljedeæi naèin:

```
inline int& Niz::operator[] (int indeks) {
    if (0 <= indeks && indeks <= duljina)
        return pokNiz[indeks];
    else
        throw NizoveIznimke("Pogrešan pristup nizu.");
}
```

Više nije potrebno postavljanje podatkovnog èlana `pogreska` – umjesto toga, u problematiènim situacijama jednostavno se podigne izuzetak.

Pažljiv čitatelj može se pitati kakva je razlika između ovakvog bacanja iznimke i jednostavnog bacanja iznimke:

```
        // ...
        throw "Pogrešan pristup nizu.";
        // ...
```

U oba sluèaja iznimka zapravo sadržava znakovni niz koji opisuje pogrešku do koje je došlo, no prvi primjer pomoæu zasebnog objekta ima veliku prednost. Naime, u jednom programu moguæe je koristiti mnogo razlièitih objekata odjednom. Ako primjerice istodobno koristimo i skupove, razumno je uvesti klasu `Skup` koja æe opisivati tu strukturu podataka. Prilikom obrade skupova također su moguæe iznimke. Ako bi i klasa `Skup` i klasa `Niz` podizale iznimke tipa `const char *`, ne bi bilo moguæe razlikovati te dvije vrste iznimaka. Naprotiv, ako za iznimke koje baca klasa `Skup` uvedemo novu klasu `SkupoveIznimke`, moguæe je zasebno napisati rutinu za obradu jednih te rutinu za obradu drugih iznimaka.

Nakon svakog bloka pokušaja mora slijediti barem jedan blok za obradu iznimaka. Taj blok se specificira ključnom riječi `catch`. Kaže se da on *hvata* (engl. *catch*) odgovarajuće iznimke iz bloka pokušaja. Nakon ključne riječi `catch` u okruglim zagradama navodi se točno jedan formalni parametar (isto kao prilikom deklaracije funkcije) koji označava tip iznimke koji se hvata tim blokom. Nakon toga se u vitičastim zagradama navodi tijelo rutine:

```
Niz a(10);
try {
    a[130] = 0;
}
catch (NizoveIznimke &pogr) {
    cerr << pogr.Pogreska() << endl;
}
```

U gornjem sluèaju rutina za obradu pogrešaka jednostavno hvata sve pogreške tipa `NizoveIznimke &` (s time da vrijede pravila standardne konverzije!) te ispisuje opis pogreške. Iza pojedinog bloka pokušaja moguæe je navesti i više blokova hvatanja iznimaka. Svaki od njih hvatat æe iznimke samo određenog tipa:

```
try {
    // problematičan dio koda
}
catch (NizoveIznimke &pogr) {
    cerr << pogr.Pogreska() << endl;
}
catch (SkupoveIznimke &pogr) {
    // obrada pogrešaka koje je izazvala klasa skup
}
```

14.3. Tijek obrade iznimaka

Razmotrimo malo detaljnije što se zapravo događa kada se prilikom izvođenja programa naiðe na kljuènu rijeè `throw`. Na primjer, promotrimo što æe se dogoditi u sljedeæem sluèaju (pri tome vrijede deklaracija i definicija klase `Niz` iz prethodnog odsjeèka):

```
void Obracunaj(Niz &n) {
    int u;
    u = n[10];      // moguća pogreška ako niz n ima
                    // manje od deset članova
    u += n[3];      // i ovo je diskutabilno mjesto
}

void Poziv() {
    Niz b(6);
    try {
        Niz a(11);
        cout << "Obračunavam se s poljem \'a\'" << endl;
        Obracunaj(a);
        cout << "Obračunavam se s poljem \'b\'" << endl;
        Obracunaj(b);
    }
    catch (NizoveIznimke& pogr) {
        cerr << pogr.Pogreska() << endl;
    }
    cout << "Kraj programa." << endl;
}
```

U gornjem primjeru funkcija `Poziv()` definira lokalna polja `a` i `b` te ih prosljeđuje kao parametre funkciji `Obracunaj()` koja koristi operator indeksiranja za pristup proslijeðenom nizu. U funkciji može doæi do iznimke prilikom pristupanja desetom èlanu polja ako polje ima manje od deset èlanova. Tada æe se podiæi iznimka unutar operatora indeksiranja. Izvoðenje operatora se prekida te se stvara privremeni objekt klase `NizoveIznimke` koji sadržava podatke o iznimci. Iznimka se zatim prosljeðuje u funkciju `Obracunaj()` zato jer je ta funkcija neposredno ispred operatora indeksiranja u lancu poziva. Operacija pridruživanja se također prekida. Uništavaju se svi lokalni i privremeni objekti koji su definirani u bloku u kojem je došlo do iznimke. To zapravo

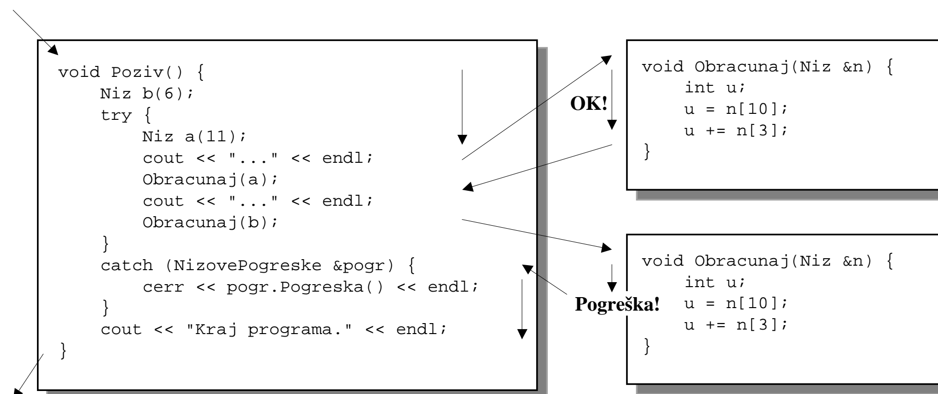
znaèi da se uništava lokalna varijabla `u` te funkcija završava. Iznimka se prosljeðuje u blok iz kojeg je problematièna funkcija pozvana, a to je pokušajni blok u funkciji `Poziv()`. Izvršavanje pokušajnog bloka se prekida te se takoðer uništavaju svi lokalni objekti – poziva se destruktor za objekt `a` te se oslobaða memorija koju je on zauzeo. Kontrola se prenosi na prvi blok hvatanja iznimke iza bloka pokušaja koji prihvaæa baèenu iznimku.

Nakon što završi blok hvatanja iznimke, uništava se privremeni objekt koji je čuvao podatke o iznimci i kontrola se prenosi na prvu naredbu iza zadnjeg bloka hvatanja. U našem primjeru to je poruka o završetku programa. Zatim završava sama funkcija `Poziv()` te se uništava njena lokalna varijabla `b`. Na slici 14.1 prikazan je tijek poziva pojedinih funkcija i obrade iznimaka.

U slučaju da unutar pokušajnog bloka ne dođe do ikakve iznimke, preskaču se svi blokovi za hvatanje iznimke te se kontrola prenosi na prvu naredbu nakon posljednjeg `catch` bloka.

Ako pak unutar pokušajnog bloka dođe do iznimke koja po tipu ne odgovara niti jednom bloku hvatanja koji neposredno slijedi pokušajni blok, tada se izvođenje te funkcije prekida, a iznimka se prenosi na prvi nadređeni pokušajni blok.

Na primjer, ako bismo promijenili funkciju `Obracunaj()` tako da ona u svom radu koristi i klasu `Skup` koja baca iznimke tipa `SkupoveIznimke`, te ako u toku izvođenja stvarno dođe do takve iznimke, blok za hvatanje iznimaka `NizoveIznimke` neće uhvatiti `SkupoveIznimke`. Izvođenje funkcije `Poziv()` će se prekinuti, uništit će se lokalni objekt `b`, te će se iznimka proslijediti u funkciju koja je pozvala funkciju `Poziv()`.



Slika 14.1. Tijek obrade iznimke

Gore opisani postupak za vraæanje poziva funkcija unatrag te uništavanja svih lokalnih objekata se zove *odmatanje stoga* (engl. *stack unwinding*) te funkcionira tako da programer može biti siguran da æe se do mjesta obrade iznimke uništiti svi lokalni i privremeni objekti koji su se mogli stvoriti do mjesta na kojem je iznimka podignuta.

Ako se iznimka ne obradi sve do završetka funkcije `main()`, poziva se funkcija `terminate()` iz standardne biblioteke koja prekida program. To zapravo znači da se cijeli C++ program može zamisliti kao da je napisan na sljedeći način:

```
try {
    main();
}
catch (...) {
    terminate();
}
```

Pri tome oznaka `...` u parametrima kljuène rijeèi `catch` oznaèava hvatanje svih iznimaka (bit æe kasnije detaljnije objašnjeno).

Nakon bacanja iznimke provodi se postupak odmatanja stoga koji æe uništiti sve lokalne objekte stvorene unutar pokušajnog bloka. Time sustav obrade iznimaka osigurava integritet memorije.

14.4. Detaljnije o kljuènoj rijeèi `catch`

Deklaracija parametara bloku hvatanja ima svojstva slièna deklaraciji preoptereæenih funkcija. Odgovarajuæi blok hvatanja se pronalazi tako da se uspoređuju tipovi objekta koji je baèen prilikom podizanja iznimke i parametra navedenog u bloku hvatanja. Ako iza bloka pokušaja postoji nekoliko blokova hvatanja, traženje odgovarajuæeg bloka se obavlja u redoslijedu njihovog navođenja. Ako niti jedan blok hvatanja ne odgovara svojim parametrima, pretražuje se nadređeni blok. Također, slièno funkcijama, ako se ne namjerava pristupati vrijednosti parametra unutar bloka, dovoljno je samo navesti tip parametra, a naziv se može izostaviti.

Iza pokušajnog bloka može se navesti više blokova hvatanja. Pri tome svaki može definirati svoj tip i na taj način obraditi odgovarajuće iznimke.

Baèeni objekt æe se predati bloku hvatanja ako je zadovoljen jedan od sljedeæih tri uvjeta na tip objekta i tip argumenta bloka:

1. Oba tipa su potpuno jednaka. U tom se slučaju ne provodi nikakva konverzija.
2. Tip parametra je klasa koja je javna osnovna klasa bačenog objekta. Tada se bačeni objekt svodi na objekt osnovne klase.
3. Ako je bačeni objekt pokazivač i ako je argument bloka pokazivač, podudaranje će se postići ako se bačeni pokazivač može svesti na pokazivač parametra standardnom konverzijom pokazivača.

Određivanje bloka hvatanja se provodi tako da se blokovi pretražuju u redoslijedu navođenja nakon bloka pokušaja.

Potrebno je obratiti pažnju na redoslijed navoðenja blokova hvatanja. Na primjer, sljedeæi redoslijed nije ispravan, jer bez obzira koji se pokazivaè baci, pomoæu standardne konverzije pokazivaèa moguæe ga je svesti na `void *` pa se drugi blok neæe nikada niti razmatrati:

```
try {
    // ...
}
catch (void *pok) {
    // sve pokazivačke iznimke će završiti ovdje
}
catch (char *pok) {
    // ovaj blok hvatanja nikada se neće dohvatiti, pa čak
    // ako se i baci objekt tipa char *
}
```

Ispravno bi bilo zamijeniti redoslijed posljednja dva bloka – tada æe iznimke tipa `char *` završiti u odgovarajuæem bloku, dok æe sve ostale iznimke koje su tipa pokazivaèa biti svedene na `void *` i proslijeðene drugom bloku.

Ponekad je potrebno napisati blok hvatanja koji će preuzimati sve iznimke. To se može učiniti tako da se kao parametar `catch` ključnoj riječi stavi znak `...` , slično kao i prilikom deklaracije funkcije s neodređenim parametrima. Na primjer:

```
try {
    // ...
}
catch (...) {
    // ovdje će završiti sve iznimke
}
```

Kako nije dano niti ime niti tip parametru, nije moguæe pristupiti vrijednosti objekta koji je baèen. Jasno je da ako postoji više blokova za hvatanje, ovakav blok mora doæi kao posljednji. Kako u njemu završavaju sve iznimke, eventualni naknadni blokovi hvatanja nikada neæe biti iskorišteni.

Obrada iznimke se često mora obaviti u nekoliko koraka. Na primjer, zamislimo da je do iznimke došlo u funkciji koja je pozvana kroz četiri prethodne funkcije. Pri tome neka je svaka funkcija zauzela određenu količinu dinamički dodijeljene memorije koju je u slučaju iznimke potrebno osloboditi. U tom slučaju svaka funkcija mora uhvatiti iznimku, osloboditi memoriju i proslijediti iznimku nadređenoj funkciji kako bi ona ispravno oslobodila svoje resurse. Iako je moguće iznimku proslijediti nadređenom bloku tako da se ponovo baci iznimka istog tipa, jednostavnije je navesti ključnu riječ `throw` bez parametra. Obrađivana iznimka će biti ponovno bačena i proslijeđena nadređenom bloku. Važno je primijetiti da je takva sintaksa dopuštena samo unutar bloka hvatanja – izvan njega ključna riječ `throw` bez parametara prouzročit će pogrešku prilikom prevođenja. Na primjer:

```
void Problematicna() {
    throw 10;
}

void Func1() {
    char *mem = new char[100];
    try {
        Problematicna();
    }
    catch (...) {
        delete [] mem;  // čišćenje zauzetih resursa
        throw;          // prosljeđivanje iznimke
    }
    delete [] mem;
}

void Func2() {
    int *pok = new int[50];
    try {
        Func1();
    }
    catch (int) {
        delete [] pok;
        throw;
    }
    delete [] pok;
}
```

Potrebno je obratiti pažnju na razliku između bloka hvatanja u funkciji `Func1()` i bloka hvatanja u funkciji `Func2()`. U oba nije moguæe pristupiti vrijednosti baèenog objekta – prilikom specifikacije drugog bloka izostavljen je naziv parametra. No prvi blok hvatanja hvata sve, a drugi samo cjelobrojne iznimke. Takoðer, iako se u prvom bloku hvatanja ne može identificirati objekt koji je baèen, moguæe je proslijediti iznimku nadreðenom bloku pomoæu kljuène rijeèi `throw` bez parametara.

Posebnu pažnju je potrebno posvetiti obrađivanju iznimaka u slučajevima kada se koristi dinamička dodjela memorije.

Dinamička memorija se ne oslobađa automatski nakon bacanja iznimke. To znači da će memorija zauzeta u bloku pokušaja ostati u memoriji ako ju ne obrišemo u bloku hvatanja.

Drugim rijeèima, ovakva funkcija neæe ispravno raditi:

```
void Func1() {
    char *mem = new char[100];
    try {
        Problematicna();
    }
```

```
        catch (...) {
            throw;          // prosljeđivanje iznimke
        }
        delete [] mem;
    }
```

Prilikom podizanja iznimke u funkciji `Problematicna()` izvoðenje programa æe se prekinuti, a kontrola toka æe se prenijeti u blok hvatanja. On æe ponovo baciti iznimku i uništiti objekt `mem`, no neæe osloboditi memoriju. Program koji èesto poziva tu funkciju æe se nakon nekog vremena vrlo vjerojatno zaglaviti jer æe ponestati slobodne memorije.

Također, potrebno je biti oprezan i s alokacijama memorije pokazivačima koji su deklarirani unutar bloka pokušaja. Ta situacija je još lošija, jer blok hvatanja uopće nema prilike osloboditi zauzetu memoriju:

```
    try {
        char *mem = new char[100];
        Problematicna();
    }
    catch (...) {
        throw;          // prosljeđivanje iznimke
    }
```

Kada funkcija `Problematicna()` podigne iznimku, automatski æe se uništiti lokalni pokazivaè `mem`, a memorija neæe biti osloboðena, niti æe se moæi osloboditi u bloku hvatanja, jer nemamo više pokazivaè.

Ako se promotri funkcija `Func1()`, može se činiti suvišnim ponavljati naredbu za brisanje podataka i u bloku hvatanja i na kraju funkcije. No to je neophodno: operator `delete` u bloku hvatanja pozivat će se samo u slučaju iznimke. Ako iznimka izostane, funkcija regularno završava te je također potrebno osloboditi zauzetu memoriju.

Osnovni razlog gornjeg ponašanja jest u tome što se pokazivač alocira na stogu, dok se memorija na koju on pokazuje alocira dinamički. Bilo bi potrebno uvesti vezu između tih dvaju objekata. To se može učiniti tako da se uvede nova klasa koja simulira pokazivač na objekt, s time da ta klasa u svom destruktoru oslobodi zauzeti objekt:

```
    class PokZnak {
    private:
        char *pokZn;
    public:
        PokZnak(int duljina) : pokZn(new char[duljina]) {}
        PokZnak(char *pz) : pokZn(pz) {}
        ~PokZnak() { delete [] pokZn; }
        operator char *() { return pokZn; }
    };
```

Sada bi se funkcija `Func1()` mogla napisati ovako:

```
void Func1() {
    PokZnak mem = new char[100];
    try {
        Problematicna();
    }
    catch (...) {
        throw;          // prosljeđivanje iznimke
    }
}
```

Pokazivaè `PokZnak` se u ovom sluèaju stvara na stogu; njegov destruktor æe automatski osloboditi memoriju te æe se memorija “oèistiti” automatski nakon prosljeðivanja iznimke nadreðenom bloku. Više nije potrebno èistiti memoriju èak niti nakon završetka funkcije `Func1()`.

Na kraju, potrebno je obratiti pažnju na razliku između objekta kao parametra i reference na objekt kao parametra. Na primjer, promotrimo razliku između prvog i drugog bloka za hvatanje:

```
class Izn {
public:
    int izn;
    Izn(int i) : izn(i) {}
};

void Func1() {
    try {
        // ... nekakva obrada
        throw Izn(10);
        // ostatak koda ...
    }
    catch (Izn& iznim) {
        // sljedeća naredba će ispisati 10
        cout << "U Func1: " << iznim.izn << endl;
        iznim.izn = 20;
        throw;
    }
}

void Func2() {
    try {
        Func1();
    }
    catch (Izn iznim) {
        // sljedeća naredba će ispisati 20
        cout << "U Func2: " << iznim.izn << endl;
        iznim.izn = 30;
        throw;
    }
}
```

```
void Func3() {
    try {
        Func2();
    }
    catch (Izn iznim) {
        // sljedeća naredba će ispisati opet 20
        cout << "U Func3: " << iznim.izn << endl;
    }
}
```

U prvom sluèaju parametar bloku hvatanja je definiran kao referenca na objekt klase `Izn`. To znaèi da æe blok hvatanja baratati zapravo s baèenim objektom, a ne s njegovom kopijom. To, nadalje, znaèi da æe svaka promjena na objektu biti upisana u originalni objekt. Ako se taj objekt proslijedi nadreðenom bloku, na njemu æe biti upisane sve izmjene.

U drugom slučaju, parametar bloku hvatanja je sam objekt. Vrijednost bačenog objekta će se prekopirati u lokalni objekt koji će se proslijediti bloku hvatanja na obradu. Za to kopiranje koristi se konstruktor kopije klase. Kako sada blok hvatanja barata s kopijom, promjena u vrijednosti objekta neće biti proslijeđena nadređenom bloku. Kopija će se uništiti prilikom izlaska iz bloka hvatanja (pomoću destruktora, ako postoji), a proslijedit će se originalni objekt.

Kada je kao parametar bloku hvatanja navedena referenca, blok hvatanja može promijeniti proslijeđeni mu objekt. Ako on dalje proslijedi objekt, promjena će biti proslijeđena narednim blokovima hvatanja.

Iz gore navedenog je vidljivo da deklaracija objekta kao parametra može rezultirati sporijom obradom iznimke (zbog toga što je potrebno kopiranje i uništavanje objekta). Zbog toga se èesto kao parametar navodi referenca na objekt, iako se ne namjerava mijenjati sama vrijednost objekta.

14.5. Navoðenje liste moguæih iznimaka

U složenim programima èesto nije moguæe pratiti koja se iznimka baca u pojedinoj funkciji. Zbog toga je potrebno uvesti nekakav mehanizam kojim æe svaka funkcija obavijestiti ostatak programa o iznimkama koje ona može baciti. Jezik C++ posjeduje tu moguænost u obliku liste moguæih iznimaka.

Lista mogućih iznimaka navodi se iza liste parametara funkcije tako da se navede ključna riječ `throw` iza koje se u zagradama popišu svi tipovi iznimaka koje funkcija može baciti:

```
void Func1() throw(int, const char*, NizoveIznimke) {
    // definicija funkcije
}

void Func2() throw() {
```

```
        // funkcija koja ne baca nikakvu iznimku
    }
```

U prvom sluèaju specificirana je funkcija koja može baciti iznimku cjelobrojnog tipa te znakovni niz. Ako sluèajno neka iznimka drukèijeg tipa osim navedenih ipak promakne obradi te bude baèena iz funkcije, pozvat æe se predefinirana funkcija `unexpected()` koja æe tipièno izazvati završetak programa. U drugom sluèaju se specificira funkcija koja ne smije baciti nikakvu iznimku.

Važno je uočiti da lista mogućih iznimaka ne definira iznimke koje se smiju pojaviti unutar funkcije, nego iznimke koje mogu biti proslijeđene iz funkcije, bilo da su bačene u samoj funkciji ili su bile bačene u nekoj funkciji pozvanoj iz funkcije. To znači da će se sljedeći kod ispravno izvesti:

```
void Func() throw(int) {
    try {
        // ...
        throw "Iznimka";
        // ...
    }
    catch (char *) {
        throw 0;
    }
}
```

Iako se iznimka tipa `const char *` pojavljuje unutar same funkcije, ona se tamo i obraðuje. Izvan funkcije se baca samo iznimka cjelobrojnog tipa, što udovoljava specifikaciji liste moguæih iznimaka.

Naposljetku, potrebno je uočiti da lista mogućih iznimaka nije dio potpisa funkcije, to jest da deklaracije

```
void Fn() throw();
void Fn() throw(int);
void Fn();
```

sve predstavljaju istu funkciju, a ne preoptereæene varijante iste funkcije. Pokušaj definiranja gornjih funkcija rezultirat æe pogreškom prilikom prevoðenja.

Zadatak. *Poznato je da nije dozvoljeno vaditi logaritam iz negativnog broja. Realizirajte funkciju `sigurni_log()` koja će u slučaju negativnog argumenta ili nule baciti izuzetak definiran klasom*

```
class LogPogreska {
public:
    double argument;
};
```

Podatkovni član `argument` *će sadržavati neispravni argument. Također, u deklaraciju funkcije ubacite naznaku koja će odrediti moguće iznimke koje funkcija može baciti.*

Zadatak. *Modificirajte klasu* `Lista` *iz poglavlja 10 o predlošcima tako da zaštitite listu od mogućih pogrešaka. Tako u slučaju da alokacija objekta* `ElementListe` *ne uspije, baca se objekt klase*

```
class NemaMemorije {};
```

U slučaju da se nekom funkcijskom članu proslijedi neispravan parametar, baca se objekt klase

```
class LosParametar {};
```

14.6. Obrada pogrešaka konstruktora

Prije uvođenja iznimaka u jezik C++, bilo je vrlo teško obraditi eventualne pogreške koje bi se mogle dogoditi prilikom izvođenja konstruktora. Jasno je i zašto: konstruktor se ne poziva eksplicitno i nema povratnu vrijednost koju bismo mogli iskoristiti za signalizaciju pogreške.

Na primjer, u dosadašnjim primjerima koristili smo klasu `ZnakovniNiz` koja je koristila dinamičku alokaciju memorije. U poseban komad memorije smješten je niz koji je proslijeđen konstruktoru kao parametar. No što ako u sistemu nema dovoljno memorije? Naš konstruktor nije provodio nikakvu provjeru – jednostavno je nakon alokacije memorije pomoću standardne funkcije `strcpy()` prepisao parametar u stvoreno memorijsko područje. Ako operator `new` nije uspio alocirati memoriju, on je vratio nul-pokazivač, pa će zbog toga operacija prepisivanja sigurno završiti sistemskom pogreškom (*general protection fault*, *segmentation fault*, *core dump* i sl.). Bilo bi dobro zaštititi klasu od takvih neurotičnih ispada i uvjeriti ju da nije lijepo srušiti program samo zato jer nije bilo dovoljno memorije.

No postavlja se pitanje kako signalizirati pozivajućem programu pogrešku? Jedna od mogućnosti je postavljanje podatkovnog člana objekta u određeno stanje. Nakon kreiranja objekta, bilo deklaracijom unutar bloka, bilo dinamičkom alokacijom, potrebno je ispisati vrijednost tog člana kako bi se ustanovilo je li objekt ispravno stvoren.

No takvo rješenje nije elegantno: time se korisnik prisiljava stalno provjeravati jesu li novi objekti ispravno stvoreni. Ako programer ne provjeri ispravnost, postoji mogućnost da dođe do pogreške prilikom korištenja objekta, a takve pogreške su izuzetno složene za pronalaženje. Osim toga, ako se i detektira pogreška prilikom stvaranja objekta, potrebno je zapisati dodatne informacije koje će pomoći prilikom uništavanja objekta. Naime, memorija za takav objekt s pogreškom će biti zauzeta, pa će ju biti potrebno osloboditi – destruktor mora imati informacije o stanju objekta kako bi ga mogao ispravno uništiti.

Dodatne probleme će prouzročiti eventualni privremeni objekti: ako primjerice vraćamo `ZnakovniNiz` iz neke funkcije i prilikom izvođenja konstruktora kopije dođe do pogreške jer nema dovoljno memorije, nemamo mogućnosti ispisati ispravnost objekta. U takvim slučajevima se jedino možemo nadati da do pogreške neće doći, a kako praksa pokazuje, takva nada je obično uzaludna. Zlatno pravilo programiranja kaže da će memorije nestati upravo u trenutku kada svoj program pokazujete šefu odjela. Bit će vrlo teško tada objasniti, znojeći se i mucajući, da program "u biti radi, no baš sada nema memorije, i ako dođe za dva sata, sve ćete srediti".

Iznimke pružaju pravi odgovor: u slučaju da prilikom izvođenja konstruktora dođe do pogreške, konstruktor će baciti iznimku koja će opisati pogrešku. Objekt neće biti stvoren, a izvođenje će se prenijeti na rutinu za hvatanje pogrešaka koja će eventualno omogućiti neki zaobilazni način rješavanja problema ili će jednostavno ispisati poruku o pogreški i završiti program. Dodajmo kôd za provjeru ispravnosti klasi `ZnakovniNiz`:

```
class ZnakovniNiz {
friend ZnakovniNiz operator +(ZnakovniNiz &a, ZnakovniNiz &b);
private:
    char *pokNiz;
public:
    ZnakovniNiz(char *niz = "");
    ZnakovniNiz(const ZnakovniNiz &ref);
    ~ZnakovniNiz();

    operator const char *() { return pokNiz; }
    ZnakovniNiz &operator =(const ZnakovniNiz &ref);
};

ZnakovniNiz::ZnakovniNiz(char *niz) :
                pokNiz(new char[strlen(niz) + 1]) {
    if (pokNiz == NULL) throw "Nema dovoljno memorije.";
    strcpy(pokNiz,niz);
}

ZnakovniNiz::ZnakovniNiz(const ZnakovniNiz &ref) :
                pokNiz(new char[strlen(ref.pokNiz) + 1]) {
    if (pokNiz == NULL) throw "Nema dovoljno memorije.";
    strcpy(pokNiz, ref.pokNiz);
}

ZnakovniNiz::~ZnakovniNiz() {
    delete [] pokNiz;
}

ZnakovniNiz &ZnakovniNiz::operator =(const ZnakovniNiz &ref) {
    if (this != &ref) {
        delete [] pokNiz;
        pokNiz = new char[strlen(ref.pokNiz) + 1];
        strcpy(pokNiz, ref.pokNiz);
    }
```

```
    return *this;
}

ZnakovniNiz operator +(ZnakovniNiz &a, ZnakovniNiz &b) {
    char *pok = new char[strlen(a.pokNiz) +
                         strlen(b.pokNiz) +1];
    strcpy(pok, a.pokNiz);
    strcat(pok, b.pokNiz);
    ZnakovniNiz rez(pok);
    delete [] pok;
    return rez;
}
```

U našem sluèaju, iznimka koja se podiže je tipa `const char *`, jer nismo htjeli komplicirati primjer uvođenjem posebne klase koja æe opisivati tip iznimke. U stvarnosti bi se taj problem riješio uvođenjem posebne klase, primjerice, `ZnakovnePogreske`, koja æe signalizirati sve pogreške prouzroèene akcijama u klasi `ZnakovniNiz`.

Ako sada želimo sa sigurnošću koristiti znakovne nizove, sve naredbe koje ih stvaraju, prosljeđuju funkcijama ili na bilo koji drugi način njima manipuliraju moraju biti smještene u blok pokušaja. Blok hvatanja mora biti podešen da hvata iznimke tipa `const char *`:

```
int main() {
    try {
        ZnakovniNiz ime("Salvador ");
        ZnakovniNiz prezime("Dalí");
        ZnakovniNiz slikar;
        slikar = ime + prezime;
        cout << (const char *)slikar << endl;
    }
    catch (const char *pogr) {
        cout << pogr << endl;
        return 1;
    }
    return 0;
}
```

15. Identifikacija tipa tijekom izvođenja

Vi vidite stvari i pitate se: “Zašto?”
No ja sanjam stvari koje nikad
nisu bile i pitam se: “Zašto ne?”

George Bernard Shaw (1856-1950)

Kako bi se koncept objektnog programiranja dosljedno proveo, u C++ jezik je ubaèeno svojstvo koje je svojevrsna nadgradnja polimorfizma. Pomoæu polimorfizma je moguæe definirati operacije koje su ovisne o tipu i na taj naèin uže povezati tip i njegovo ponašanje, no samo pomoæu njega nije moguæe doznati o kojem se tipu stvarno radi. Identifikacija tipa u toku izvođenja omoguæava upravo to: bez obzira na pokazivaè ili referencu kojom rukujemo s nekim objektom, u bilo kojem trenutku je moguæe doznati koje klase jest objekt na koji oni pokazuju. Osim prepoznavanja, uvedena je i moguænost usporedbe tipova.

Također, kako bi se povećala sigurnost dobivenih programa, uveden je niz novih operatora za dodjelu tipa koji eksplicitno izražavaju smisao dodjele. Najvažniji od njih je operator dinamičke dodjele, koji prije dodjele provjerava ima li dodjela uopće smisla.

15.1. Statièki i dinamièki tipovi

Jedno od najznaèajnijih svojstava koje C++ jezik èini moænim i prikladnim za razna podruèja primjene je polimorfizam. Prisjetimo se ukratko o èemu se radi. Osnovna ideja je da tip objekta ne specificiramo strogo prilikom prevođenja, veæ da prilikom izvođenja objekt sam obavi svoje operacije na sebi svojstven naèin. To je postignuto nasljeđivanjem i korištenjem virtualnih funkcija. Na primjer:

```
class Linija {
// ...
public:
    virtual void Crtaj();
    // ...
};

class LinijaSaStrelicama : public Linija {
// ...
public:
    virtual void Crtaj();
```

```
    // ...
};
```

U gornjem primjeru klasa `Linija` definira liniju na zaslonu raèunala. Tu klasu smo naslijedili i iz nje izveli klasu `LinijaSaStrelicama` koja opisuje liniju zakljuèenu strelicama na obje strane. Svaka od tih dviju klasa ima funkcijski èlan `Crtaj()` koji obavlja operaciju crtanja objekta na ekranu. No taj èlan je definiran kao virtualan, što znaèi da se njegov poziv obavlja dinamièki – analizira se tip objekta za koji se traži poziv, te se poziva ispravan èlan:

```
Linija *ln = new Linija;
ln->Crtaj();        // pozvat će Linija::Crtaj i iscrtat će
                    // običnu liniju na ekranu
delete ln;
ln = new LinijaSaStrelicama;
ln->Crtaj();        // pozvat će LinijaSaStrelicama::Crtaj
                    // i iscrtat će liniju sa strelicama
delete ln;
```

U drugom sluèaju stvara se objekt `LinijaSaStrelicama`, ali se njegova adresa pridodijeljuje pokazivaèu na tip `Linija`. Ako bi se poziv

```
ln->Crtaj();
```

obavio po statièkom tipu pokazivaèa `ln`, odnosno ako bi se išlo po logici “`ln` je pokazivaè na objekt `Linija` pa æe se zato pozvati èlan `Crtaj()` iz klase `Linija`”, rezultat ne bi bio korektan, jer `ln` zapravo pokazuje na tip `LinijaSaStralicama`. U tome i jest bit polimorfizma: pomoæu pokazivaèa na osnovnu klasu može se korektno baratati i objektima izvedenih klasa.

No ponekad nam može biti vrlo interesantno ustvrditi na koji objekt svarno pokazivač pokazuje. Na primjer, zamislimo da imamo niz pokazivača koji pokazuju na objekte klase `Linija` (ili objekte izvedene iz te klase). Neka je sada potrebno napisati program koji će nacrtati samo linije, dok će se linije sa strelicama namjerno preskočiti (primjerice radi lakšeg pregleda crteža na ekranu). Problem je u tome što mi ne možemo sa sigurnošću odrediti točan tip na koji pokazuje pojedini pokazivač niza. Jedno od mogućih rješenja je uvođenje virtualne funkcije `Tip` koja će vraćati cijeli broj koji će na jedinstven način identificirati klasu. Na primjer:

```
#define CLS_LINIJA                  1
#define CLS_LINIJASASTRELICAMA      2

class Linija {
// ...
public:
    virtual int Tip() { return CLS_LINIJA; }
    // ...
};
```

```
class LinijaSaStrelicama : public Linija {
// ...
public:
    virtual int Tip() { return CLS_LINIJASASTRELICAMA; }
    // ...
};
```

Sada se pomoæu mehanizma virtualnih funkcija pojedini tipovi mogu razlikovati. Gornji kôd se može pozivati ovako:

```
int main() {
    Linija *niz[100];
    int duljNiza = 0;
    // inicijalizacija elemenata niza
    niz[duljNiza++] = new Linija;
    niz[duljNiza++] = new LinijaSaStrelicama;
    // itd...

    // crtanje samo linija
    for (int i = 0; i < duljNiza; i++)
        if (niz[i]->Tip() == CLS_LINIJA)
            niz[i]->Crtaj();
    return 0;
}
```

Izneseno rješenje æe raditi sasvim korektno. No problem je što svaki programer može osigurati tipiziranje objekata na svoj vlastiti naèin. Na primjer, jedan æe funkciju nazvati `Tip()`, drugi `DajTip()` a treæi `ReciMiReciOgledajceMoje_TipMojKojiJe()`. Također, ne postoji nikakav naèin da se osigura jednoznaènost svih tipova. To znaèi da ako koristimo biblioteku koju je razvio netko drugi, može nam se dogoditi da identifikatori tipa iz te biblioteke budu jednaki onima koje smo koristili u drugim dijelovima programa. Zbog takvih i sliènih problema, u C++ jezik je uveden mehanizam za *identifikaciju tipova tijekom izvođenja* (engl. *run-time type identification*).

15.2. Operator `typeid`

Kako bi se omoguæio jednoznaèan naèin dobavljanja informacije o stvarnom tipu nekog objekta, uveden je operator `typeid`. On se primjenjuje tako da se u zagradama navede izraz koji nakon izraèunavanja ima vrijednost pokazivaèa, objekta ili reference na objekt neke klase. Kao rezultat operator æe dati informaciju o stvarnom tipu objekta tako što æe vratiti referencu na konstantan objekt klase `type_info`. Ta klasa je definirana standardom u datoteci zaglavlja `typeinfo.h`, a objekti te klase nose sve relevantne podatke o tipu objekta. Objekte klase `type_info` moguæe je uspoređivati operatorima `==` i `!=` kako bi se ispitala jednakost dvaju tipova. Evo kako je ona deklarirana:

```
class type_info {
public:
    virtual ~type_info();
    bool operator==(const type_info&) const;
    bool operator!=(const type_info&) const;
    bool before(const type_info&) const;
    const char* name() const;
private:
    type_info(const type_info&);
    type_info& operator=(const type_info&);
    // podatkovni članovi
};
```

Naziv klase se može dobiti pomoæu poziva funkcijskog èlana `name()`. Ponekad je potrebno leksièki usporeðivati `type_info` objekte. Tome služi funkcijski èlan `before()` – ako je objekt na koji pokazuje parametar leksièki ispred objekta, onda se vraæa logièka istina. To može biti korisno ako bismo podatke o tipovima smjestili u tablicu koju, radi bržeg pristupa želimo sortirati i primijeniti, primjerice, binarno pretraživanje.

Operator `typeid` se može primijeniti na proizvoljan izraz. Rezultat operatora opisuje tip koji ima rezultirajuća vrijednost izraza. Također, moguće je operator primijeniti i na sam identifikator tipa, te se tada vraća objekt koji opisuje navedeni tip. Naš problem crtanja samo objekata klase `Linija` sada bi se mogao riješiti na ovakav način:

```
    // početak programa je identičan
    for (int i = 0; i < duljNiza; i++)
        if (typeid(niz[i]) == typeid(Linija))
            niz[i]->Crtaj();
```

Više nije potreban virtualni funkcijski èlan `Tip` koji æe jednoznaèno opisivati tip objekta – tipovi se usporeðuju direktno. Operator `typeid` se može primijeniti i na ugraðene tipove. Važno je primijetiti da operator ne razlikuje konstantne i nekonstantne tipove. Na primjer:

```
Linija ln1;
const Linija ln2;

typeid(ln1) == typeid(ln2);             // uvijek je istina
typeid(ln1) == typeid(const Linija);    // također je
                                        // uvijek istina
```

Ako se kao argument `typeid` operatoru proslijedi nul-pokazivaè, operator æe podignuti iznimku `bad_typeid`. To je radi toga što nul-pokazivaè nema tipa, pa operator `typeid` ne može vratiti nikakvu suvislu vrijednost. Jedina ispravna akcija je podiæi iznimku.

Napomenimo da intenzivno korištenje operatora `typeid` vrlo često signalizira da nešto sa strukturom programa nije u redu. Naime, taj operator izričito ruši tipizacijski

sustav jezika C++. U mnogim slučajevima je pametnije umjesto `typeid` operatora dodati određenu virtualnu funkciju u osnovnu klasu koja će obavljati operaciju ovisnu o tipu. To ne znači da `typeid` treba protjerati zajedno s naredbom `goto` na planetu Salsua Secundus[†]. Postoje situacije u kojima je `typeid` koristan: na primjer, u određenim uvjetima htjeli bismo, radi provjere ispravnosti programa, ispisati naziv klase nekog objekta. To možemo učiniti ovako:

```
void MaKojeSiMiTiKlase(GrafObjekt *go) {
    cout << typeid(*go).name() << endl;
}
```

Primjetimo da nas èesto ne interesira koje je toèno klase određeni objekt, veæ je li objekt možda izveden iz neke klase kako bismo sigurno mogli pristupiti nekom njegovom svojstvu. U tom sluèaju definitivno nije preporuèljivo pisati u programu velike `switch` naredbe u kojima se testira tip objekta: ako se pojavi nova klasa u hijerarhiji potrebno je sve takve blokove proširiti. Pametnije je korisiti sigurnu pretvorbu na niže.

15.3. Sigurna pretvorba

U poglavlju o nasljeðivanju opisane su standardne konverzije pokazivaèa. Tamo je navedeno da se svaki pokazivaè na neki objekt može implicitno pretvoriti u pokazivaè na objekt èija klasa je javna osnovna klasa objekta na koji pokazivaè pokazuje. To znaèi da je moguæe sljedeæe:

```
Linija *pokLinija = new LinijaSaStrelicama;
```

Gornje pridruživanje je moguæe zato jer je `LinijaSaStrelicama` izvedena iz klase `Linija` javnim nasljeðivanjem. Svaka linija sa strelicom æe sadržavati sve podatkovne i funkcijske èlanove linije. Takva pretvorba se zove *pretvorba naviše* (engl. *upcast*).

No ponekad je potrebno i suprotno. Na primjer, u gornjem slučaju pokazivač `pokLinija` u biti pokazuje na objekt klase `LinijaSaStrelicama`. Zbog toga ima smisla pretvoriti ga u pokazivač na `LinijaSaStrelicama` te pristupati eventualnim dodatnim podatkovnim i funkcijskim članovima. No prevoditelj ne može znati u toku prevođenja na što će pokazivač pokazivati – to se zna tek prilikom izvođenja. Zbog toga pretvorba iz osnovne u izvedenu klasu nije automatski moguća, nego ju je potrebno eksplicitno zatražiti:

```
Linija *pokLinija = new LinijaSaStrelicama;
LinijaSaStrelicama *lss = (LinijaSaStrelicama*)pokLinija;
```

Ovakva pretvorba se zove *pretvorba naniže* (engl. *downcast*). Ona se ne provodi automatski, nego iskljuèivo na eksplicitan zahtjev programera. Ako bi u gornjem sluèaju

[†] Carski Zatvorski Planet za okorjele kriminalce iz romana “Dune” Franka Herberta.

pokLinija ipak pokazivao na objekt klase Linija, nakon pretvorbe u LinijaSaStrelicama i pokušaja pristupa nekom podatkovnom èlanu može doæi do velikih problema ako taj èlan postoji samo u izvedenoj klasi. Na primjer, ako klasa LinijaSaStrelicama sadrži cjelobrojni èlan tipStrelice koji oznaèava tip strelice, tada æe sljedeæi programski kôd prouzroèiti velike probleme:

```
Linija *pokLinija = new Linija;
LinijaSaStrelicama *lss = (LinijaSaStrelicama *)pokLinija;
pokLinija->tipStrelice = 10;    // opasno (po život)!
```

Objekt na koji pokLinija pokazuje ne sadrži èlan tipStrelice, jer je pomoæu operatora new alociran objekt klase Linija. No mi smo pomoæu eksplicitne pretvorbe prisilili prevoditelj da provede pretvorbu tipa. Time smo omoguæili pristup nepostojeæem èlanu. Pokušaj dodjele æe prepisati neki sluèajan komad memorije koji se nalazi neposredno iza objekta, a to sigurno nije ono što smo željeli (osim ako pišemo novu verziju virusa). Problem je u tome što je cijeli program ispravno preveden: sva odgovornost je na programeru.

Pretvorba naniže je potencijalno opasna ako se ne koristi pažljivo. Ako ju morate koristiti, poslužite se radije sigurnom pretvorbom naniže opisanom u tekstu koji slijedi.

Kada je u C++ jezik uveden podsustav za određivanje tipova prilikom izvođenja, otvorila se moguænost da se uvede sigurna pretvorba naniže. Naime, moguæe je prvo provjeriti koji je to tip na koji pokazivaè pokLinija pokazuje, a zatim obaviti pretvorbu ako je ona dozvoljena. U suprotnom, pretvorba se neæe obaviti. Tu operaciju uvodi novi operator dynamic_cast. Njegova sintaksa je

```
dynamic_cast<T>(izr)
```

gdje izr predstavlja izraz èiju je vrijednost potrebno pretvoriti, a T predstavlja željeni tip. Kaže se da je pretvorba dinamièka zato jer se ne obavlja prilikom prevođenja, nego prilikom izvođenja programa.

Pretvorba će se provesti samo ako je na jednoznačan način moguće od tipa izraza izr doći do tipa T. Ovaj operator će obaviti i običnu pretvorbu u osnovnu klasu. Ako je T pokazivač na osnovnu klasu objekta na koji pokazuje izr, tada je konverziju moguće provesti statički prilikom prevođenja.

Promotrimo slučaj kada je T izvedena klasa u odnosu na klasu na koju pokazuje izr. Napominjemo da je postupak isti bez obzira radi li se o pokazivaču ili referenci na objekt. Najprije će se odrediti tip cijelog objekta na koji izr pokazuje. Ako se za taj tip može pronaći podobjekt T kao jednoznačna javna osnovna klasa, onda će rezultat pretvorbe biti pokazivač (ili referenca) koja pokazuje na taj podobjekt. Jednostavnije rečeno, provjerit će se je li konverzija u izvedenu klasu dozvoljena s obzirom na stvarni objekt koji se pretvara. Nul-pokazivač će se pretvoriti u samoga sebe.

Ponašanje operatora dinamičke konverzije se razlikuje za pokazivače i reference u slučaju da gornji uvjet nije ispunjen. Ako se radi o pretvorbi pokazivača, rezultat konverzije će biti nul-pokazivač. Ako se pak radi o konverziji referenci, bit će podignuta iznimka tipa `bad_cast`. Razmotrimo zbog čega dolazi do takvog ponašanja.

U slučaju pokazivača, neispravna konverzija se jednostavno može dojaviti ostatku programa tako da se vrati nul-pokazivač. No nul-reference ne postoje pa operator konverzije ne može vratiti nikakvu suvislu vrijednost koja bi ukazivala na lošu konverziju. Zbog toga je jedino ispravno rješenje prekinuti program i podići iznimku.

Razmotrimo to na primjeru. Sljedeća konverzija je ispravna jer pokazivač uistinu pokazuje na objekt klase `LinijaSaStrelicama`:

```
Linija *pokLinija = new LinijaSaStrelicama;
LinijaSaStrelicama *lss =
            dynamic_cast<LinijaSaStrelicama *>(pokLinija);
if (lss) {
    // ...
}
```

Kako se radi o pretvorbi pokazivaèa, nakon provedene pretvorbe ispravnost konverzije se može provjeriti tako da se testira je li pokazivaè jednak nul-pokazivaèu. U sljedeæem sluèaju konverzija referenci æe biti neispravna te æe doæi do podizanja iznimke `bad_cast`. Zbog toga je potrebno cijeli postupak staviti u blok pokušaja:

```
Linija &ln = *new Linija;
try {
    dynamic_cast<LinijaSaStrelicama &>(ln).tipStrelice = 10;
}
catch (bad_cast) {
    cerr << "Loša pretvorba." << endl;
}
```

U gornjem primjeru se referenca `ln` na objekt klase `Linija` pokušava pretvoriti u referencu na objekt klase `LinijaSaStrelicama` pomoæu operatora `dymanic_cast`. Ako pretvorba uspije, rezultat je referenca pa se odmah može pristupiti èlanu `tipStrelice` pomoæu operatora `.` za pristup èlanovima klase. Ako pretvorba ne uspije (što æe biti sluèaj u našem primjeru, jer je referenca inicijalizirana objektom klase `Linija`), postupak je potrebno prekinuti i sprijeèiti neispravno pridruživanje. Zbog toga æe operator `dynamic_cast` podignuti iznimku `bad_cast` koja æe se uhvatiti u bloku hvatanja.

Neuspjela dinamièka dodjela tipa æe u sluèaju pokazivaèa vratiti nul-pokazivaè, dok æe u sluèaju referenci podiæi iznimku `bad_cast`.

Dinamièke pretvorbe nisu svojstvo C++ jezika koje treba koristiti èesto. Kao i operator `typeid`, one narušavaju statièke provjere tipa prilikom prevoðenja te zbog toga

jednostavno nisu “u duhu” C++ jezika. Mnoge stvari za koje bi mogli doæi u napast te primijeniti dinamièke pretvorbe, mogu se kvalitetnije riješiti pomoæu predložaka. Intenzivno korištenje dinamièkih pretvorbi u veæini sluèajeva signalizira da korisnik pokušava simulirati pristup programiranju koji se koristi u drugim objektno orijentiranim jezicima, kao što je SmallTalk.

Objasnimo to na primjeru kontejnerskih klasa. U jezicima koji se oslanjaju na dinamičku provjeru tipa kontejnerski objekt se najčešće rješava tako da se stvori klasa koju svi objekti koje želimo smjestiti u kontejner mora naslijediti. Na primjer, to smo i mi učinili u poglavlju o nasljeđivanju: članove koje smo htjeli smještati u listu morali smo naslijediti od klase `Atom`. Kontejnerska klasa `Lista` manipulira s objektima klase `Atom`, a podatak o tipu stvarnog objekta je izgubljen prilikom smještanja objekta u kontejner.

Kako klasa `Lista` ne zna ništa o tipu podataka, prilikom čitanja podataka iz kontejnera potrebno je pomoću operatora za dodjelu tipa pretvoriti vraćeni pokazivač iz pokazivača `Atom *` u pokazivač na stvarni objekt. U poglavlju 9 o nasljeđivanju još nismo prezentirali dinamičku dodjelu tipa, pa smo bili prisiljeni koristiti statičku dodjelu tipa. Iz tamo navedenog primjera se vidi da je kôd dosta nerazumljiv i podložan pogreškama. Bolje je rješenje pomoću predložaka: za svaki mogući tip koji se smješta u kontejner generira se posebna klasa koja je u stanju vratiti ispravan pokazivač. Time se smanjuje zamor programera te se uklanjaju mogući izvori pogrešaka. Primjetite da u SmallTalku to nije problem: tip objekta se ne provjerava prilikom prevođenja, nego prilikom izvođenja. Ako programer pozove neku funkciju na objektu krivog tipa, program će se jednostavno prekinuti uz poruku o pogreški. No za takav luksuz cijena je velika: SmallTalk programi se izvode znatno sporije od C++ ekvivalenata te se mnoge pogreške uočavaju tek kada se program izvede.

Sigurno si postavljate pitanje zašto smo uopće naveli takav primjer. Odgovor je u tome što je opisani način bio dugo vremena i jedini: mnogi prevoditelji nisu podržavali predloške te su korisnici jednostavno kopirali pristup iz drugih programskih jezika. Možda štovani čitatelj ima upravo takav prevoditelj. Čak da to i nije slučaj, osobno smatramo da je korisno upoznati različite stilove programiranja, s njihovim prednostima i manama. Niti jedno znanje nije suvišno!

Odmah ćemo to i dokazati: što ako naš kontejner mora čuvati objekte različitog tipa? U tom slučaju izneseni pristup je i jedini. Zamislimo da korisnimo biblioteku grafičkih elemenata iz poglavlja 9 o nasljeđivanju te želimo dodati novi objekt za crtanje Bezierovih krivulja `Bezier`. Objekti tog tipa će sigurno imati sasvim različite članove za postavljanje pozicije od klase `Linija`. Javno sučelje tih klasa nikako ne možemo objediniti virtualnim funkcijama u korijenu hijerarhije, pa da bi nam bilo svejedno s kojim pokazivačem radimo. Štoviše, nekada niti ne možemo promijeniti korijen hijerarhije, jer se nalazi u biblioteci koju ne možemo mijenjati. U tom slučaju jedino rješenje je nakon dobavljanja objekta iz kontejnera iskoristiti dinamičku pretvorbu naniže te pristupiti specifičnim članovima:

```
Lista lst;
// lista se puni s objektima iz hijerarhije grafičkih objekata
```

```
// pretpostavimo da smo sve grafičke objekte izveli iz klase
// Atom kako bismo omogućili njihovo pohranjivanje u listu
// ...

GrafObjekt *pgo = lst.AmoGlavu();
if (Bezier *pbez = dynamic_cast<Bezier *>(pgo)) {
    // slobodno pristupite Bezierovoj krivulju preko pbez
    // ...
}
```

Ovdje vrijedi isto što smo rekli i za `typeid`: ako kasnije dodamo novi objekt u hijerarhiju, morat æemo promijeniti program tako da na adekvatan naèin podržava nove klase.

15.4. Ostali operatori konverzije

Osim operatora dinamièke dodjele tipa C++ jezik raspolaže s još tri razlièita operatora. Oni su uvedeni da bi se olakšale i uèinile jasnijima neke dodjele potrebne u objektno orijentiranom programiranju. Oni u principu obavljaju sve što može obaviti i obièna dodjela tipa.

15.4.1. Promjena konstantnosti objekta

Jezik C++ raspolaže kljuènom rijeèi `const` kojom se prevoditelju može dati na znanje da mora osigurati nepromjenjivost nekog objekta. Ipak, ponekad se može ukazati potreba da se privremeno konstantnost objekta ukloni, te da mu se promijeni neki podatkovni èlan. Tada je moguæe iskoristiti operator `const_cast`. On je opæeg oblika

```
const_cast<T>(izr)
```

Pri tome izraz `izr` i tip `T` moraju biti jednaki (oba moraju biti pokazivaèi ili reference na objekt), s time da se mogu razlikovati u kvalifikatoru `const` i `volatile`. U tom sluèaju æe se pretvorba provesti te æe se novonastali objekt tretirati kao da mu je uklonjen (ili dodan) `const` odnosno `volatile` kvalifikator. Na primjer:

```
const Vektor v(0, 50);
const_cast<Vektor&>(v).PostaviXY(10, 20);
```

Kako je objekt `v` definiran konstantnim, poziv funkcijskog èlana `PostaviXY()` na njemu ne bi bio dozvoljen. No pomoæu operatora `const_cast` objekt se može uèiniti nekonstantnim te mu se ipak može promijeniti vrijednost.

15.4.2. Statièke dodjele

Postoje dva operatora statièke dodjele: `static_cast` i `reinterpret_cast`. Njihov opæi oblik je

```
static_cast<T>(izr)
reinterpret_cast<T>(izr)
```

Tip `T` oznaèava ciljni tip dodjele te mora biti pokazivaè, referenca, aritmetièki tip ili pobrojenje. Za veæinu tipova gornji operatori æe dati isti rezultat kao i obièan operator dodjele tipa

```
(T)izr
```

Pretvorba æe se obaviti na isti naèin. Postavlja se pitanje èemu su onda uopæe uvedeni ovi novi operatori, te u èemu je njihova razlika.

Osnovna razlika je način na koji se tretiraju klase prilikom nasljeđivanja. Naime, tip `T` kod operatora `static_cast` mora u potpunosti biti poznat prilikom prevođenja. To znači da klasa u koju se pretvara mora biti u cijelosti definirana (a ne samo deklarirana unaprijed). Pogledajmo što se događa prilikom korištenja tog operatora i višestrukog nasljeđivanja:

```
class Osnovna1 {
public:
    int osn1;
};

class Osnovna2 {
public:
    int osn2;
};

class Izvedena : public Osnovna1, public Osnovna2 {
public:
    int izv;
};

int main() {
    Izvedena i;
    i.osn1 = i.osn2 = i.izv = 0;
    Osnovna2 *pok = static_cast<Osnovna2 *>(&i);
    pok->osn2 = 30;
    cout << i.osn1 << endl << i.osn2 << endl;
    return 0;
}
```

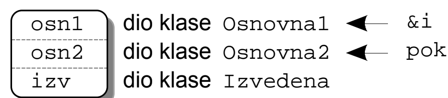
Važno je sjetiti se da se objekt klase `Izvedena` sastoji od dva podobjekta – svaki za jednu osnovnu klasu. U memoriji taj objekt je složen prema slici 15.1.

Pokazivač `&i` pokazuje na prvu memorijsku lokaciju zauzetu objektom. Prilikom pretvorbe u pokazivač na klasu `Osnovna2` vrijednost pokazivača se mora podesiti tako da on pokazuje na početak podobjekta `Osnovna2`. To je moguće zato što operator dodjele tipa tretira klase kao kompletne tipove pa pomoću deklaracije klase može podesiti vrijednost pokazivača. Zbog toga će se gornji kôd izvesti na očekivani način: dodjela preko pokazivača `pok` će stvarno pristupiti članu `osn2` te će se on promijeniti i u objektu `i`.

Operator `reinterpret_cast` radi na drukčiji način. On sve klase tretira kao nekompletne te ne uvažava njihove deklaracije. Promijenimo funkciju `main()` na sljedeći način:

```
int main() {
    Izvedena i;
    i.osn1 = i.osn2 = i.izv = 0;
    Osnovna2 *pok = reinterpret_cast<Osnovna2 *>(&i);
    pok->osn2 = 30;
    cout << i.osn1 << endl << i.osn2 << endl;
    return 0;
}
```

Prilikom pretvorbe operator neæe provesti nikakvu promjenu vrijednosti pokazivaèa. On æe jednostavno interpretirati njegovu vrijednost na novi naèin – kao pokazivaè na objekt `Osnovna2`. Pristup èlanu `osn2` preko pokazivaèa `pok` rezultirat æe promjenom vrijednosti èlana `osn1`, što se može i vidjeti nakon prevođenja i izvođenja programa. Ovakva pretvorba može biti vrlo korisna prilikom pisanja sistemskih programa, kada je potrebno vrlo precizno kontrolirati sve resurse raèunala.



Slika 15.1. Princip smještaja objekta u memoriju

16. Pretprocesorske naredbe

Je li to napredak kada ljudožder koristi nož i vilicu?

Stanislaw Lec (1909–1966), "Nepočešljana razmišljanja"

U ovom poglavlju upoznat æemo se s pretprocesorskim naredbama i njihovom primjenom pri pisanju programa u jeziku C++. Pretprocesorske naredbe te makro definicije i makro funkcije koje se pomoæu pretprocesorskih naredbi ostvaruju èesto se koriste pri pisanju programa u programskom jeziku C. Meðutim, zahvaljujuæi dodatnim ugraðenim svojstvima jezika C++, njihova primjena u jeziku C++ je znaèajno smanjena.

16.1. U poèetku bijaše pretprocesor

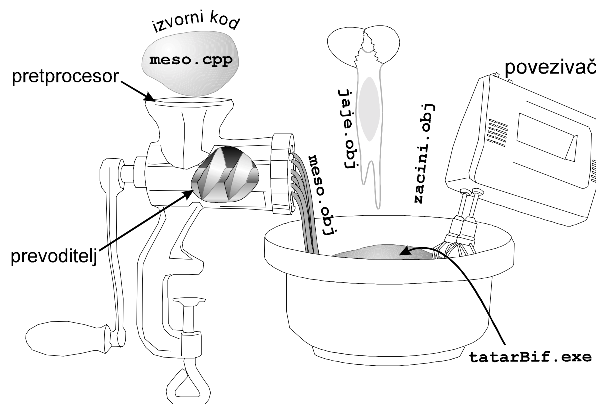
Prilikom pisanja programa, veæinu kôda saèinjavaju C++ naredbe koje se prevoðenjem i povezivanjem pretvaraju u izvedbeni kôd. Meðutim, osim naredbi jezika C++, u nerijetko se koriste i takozvane pretprocesorske naredbe kojima se automatiziraju neke operacije. Pretprocesorske naredbe prepoznaju se po znaku `#` (jugonostalgièari bi rekli "taraba") kojim svaka pretprocesorska naredba (ili *direktiva*) zapoèinje.

Analiza pretprocesorskih naredbi se provodi neposredno prije prevođenja kôda. Pretprocesor će modificirati izvorni kôd, generirajući pritom novu, privremenu datoteku. Ta datoteka, u kojoj više nema pretprocesorskih naredbi, proslijedit će se prevoditelju. Korisnik tu datoteku neće moći vidjeti, osim ako posebno podesi parametre svog prevoditelja. Prevoditelj dakle ne obrađuje originalnu datoteku izvornog kôda, već datoteku koju mu je pretprocesor pripremio. Tijek (pret)procesiranja izvornog kôda može se vidjeti na slici 16.1 na sljedećoj stranici. (Molimo da nam pravi ljubitelji tatarskog bifteka ne zamjere na korištenju flajš-mašine.)

Znak `#` koji označava početak pretprocesorske naredbe mora biti prvi znak u retku iza eventualnih praznina. Naredba se proteže do kraja retka (tj. do znaka za novi redak), ali se može nastaviti i u sljedećem retku, ako se redak zaključi znakom `\`. U tablici 16.1 su navedene sve pretprocesorske naredbe, grupirane u dva stupca: naredbe za uvjetno uključivanje kôda i kontrolne linije.

Tablica 16.1. Pretprocesorske naredbe

`#if`	`#include`
`#ifdef`	`#define`
`#ifndef`	`#undef`
`#elif`	`#line`
`#else`	`#error`
`#endif`	`#pragma`



Slika 16.1. *Pretprocesiranje*, *kompajliranje* i *linkanje* tatarskog bifteka

Pretprocesorske naredbe se u C++ programima uglavnom koriste za uključivanje datoteka s deklaracijama funkcija i klasa te za uvjetno prevođenje dijelova programa. One također omogućavaju definiranje konstanti te makro funkcija, što se obilato koristilo u C programima. Međutim, predlošci funkcija, umetnute funkcije i preopterećenje funkcija čine makro funkcije u C++ programima potpuno izlišnima.

Budući da pretprocesorske naredbe uzrokuju promjenu izvornog kôda neposredno prije prevođenja, one znaju otežati pronalaženje pogrešaka u kôdu. Iz istog razloga one će bitno ograničiti usluge koje vam pružaju pomoćni alati poput programa za simboličko otkrivanje pogrešaka ili programa za praćenje efikasnosti kôda (engl. *profiler*). To ćemo ilustrirati u odjeljku 16.3, na primjeru definicije konstante.

16.2. Naredba `#include`

Naredbu `#include` veæ smo uèestalo koristili za ukljuèivanje deklaracije funkcija iz standardnih biblioteka. Njome se u suštini provodi tekstovno povezivanje razlièitih datoteku u jednu cjelinu. Tako æe naredbu

```
#include "toupet"
```

pretprocesor nadomjestiti sadržajem datoteke `toupet`. Ta datoteka mora biti izvorni kôd pisan u C++ jeziku, buduæi da æe ona uæi u privremenu datoteku koja se prosljeðuje prevoditelju. Ime datoteke koju treba ukljuèiti može biti unutar dvostrukih navodnika `"` ili unutar znakova < i > (“manje od”, odnosno “veæe od”):

```
#include "mojeKlase.h"      // iz tekućeg imenika
#include <iostream.h>       // iz standardnog imenika
```

Postoji suštinska razlika između ova dva naèina omeðivanja imena datoteke. Kada je ime datoteke navedeno unutar znakova < i >, ona se traži u standardnom imeniku za

datoteke zaglavlja, kako je definirano razlièitim opcijama prevoditelja. Ako je definirano više imenika, tada se traženje provodi redoslijedom kako su ti imenici navedeni. Kada je ime omeðeno navodnicima, tada se datoteka prvo traži u tekuæem (radnom) imeniku, a ako se ne pronaðe tamo, datoteka se traži u standardnim imenicima za datoteke zaglavlja.

Prilikom navođenja imena datoteke valja paziti da se ne umeću praznine između imena i navodnika, odnosno znakova < >:

```
#include < string.h >        // nije isto kao <string.h>
```

Naredba `#include` je neizbježna pri ukljuèivanju biblioteka standardnih funkcija, te kod veæih programa u kojima se programski kôd rasprostire kroz nekoliko datoteka, o èemu æe biti detaljno govora u zasebnom poglavlju.

16.3. Naredba #define

Pretprocesorskom naredbom `#define` definira se vrijednost simbola, tzv. *makro imena* (engl. *macro name*). Tako æe naredba

```
#define NEKAJ   NEŠTO DRUGO UMJESTO NEKAJ
```

postaviti vrijednost prvog niza (makro ime `NEKAJ`) na vrijednost navedenu u nastavku naredbe (`NEŠTO DRUGO UMJESTO NEKAJ`). Naiðe li pretprocesor na simbol `NEKAJ` u djelu programa koji slijedi, svaki put æe ga zamijeniti nizom `NEŠTO DRUGO UMJESTO NEKAJ`.

Iako nije nužno, uobičajena je programerska praksa da se, zbog lakše uočljivosti u izvornom kôdu, makro imena pišu velikim slovima.

Radi boljeg razumijevanja prouèit æemo kako bi gornja naredba `#define` djelovala na nekoliko razlièitih pojava niza `NEKAJ` (bez obzira na smisao konaènog kôda). Pretpostavimo da gornjoj naredbi slijede naredbe:

```
a = NEKAJ;              // nadomjestit će
char *b = "NEKAJ";      // neće nadomjestiti
c = PONEKAJ;            // neće nadomjestiti
```

Pretprocesor æe nadomjestiti samo prvu pojavu niza, na što æe prevoditelj prijaviti pogrešku, jer æe naredba koju æe on zaprimiti imati oblik:

```
a = NEŠTO DRUGO UMJESTO NEKAJ;
```

U drugoj naredbi æe sadržaj znakovnog niza ostati nepromijenjen, jer pretprocesor ne pretražuje sadržaje znakovnih nizova. U treæoj naredbi zamjena neæe biti obavljena, jer niz nije u potpunosti jednak traženom.

Naredba `#define` se često rabi u C programima za definiranje konstanti, ali u jeziku C++ valja izbjegavati takvu njenu namjenu. Obrazložimo i zašto. Želimo li definirati nepromjenjivu veličinu, u jeziku C++ ćemo to napraviti pomoću ključne riječi `const`:

```
const double pi = 3.14159;
```

Pomoæu pretprocesorske naredbe to bismo napravili na sljedeæi naèin:

```
#define PI 3.14159
```

Ona æe prouzroèiti da sve pojave znakovnog niza `PI` u izvornom kôdu budu zamijenjene brojem 3.14159. Iako u konaènom ishodu (sasvim vjerojatno) neæe biti nikakve razlike, postoji suštinska razlika između ova dva pristupa. U prvom sluèaju æe u programu postojati (istina nepromjenjiva) varijabla `pi`, koja æe zauzimati određeni memorijski prostor, pa æe biti dostupna programu za otkrivanje pogrešaka. Naprotiv, u sluèaju pretprocesorske definicije æe sve pojave znakovnog niza `PI` biti zamijenjene brojem 3.14159 još prije prevođenja. Stoga neæe postojati zasebna varijabla s tom vrijednošæu i program ju neæe moæi identificirati. “Pa što onda? `PI` je uvijek `PI`, tj. 3.14159”, reæi æe poneki èitatelj. To je toèno. Međutim, poželite li u programu za simbolièko otkrivanje pogreški izraèunati vrijednost koju daje izraz

```
float PovrsinaKruga = r * r * PI;
```

to neæete moæi napraviti tako da napišete `PI`, jer on ne postoji kao podatak u programu. Umjesto toga morat æete napisati broj 3.14159. A tko vam pritom jamèi da je `PI` na tom mjestu zaista nadomješten tim brojem? Ne postoji naèin da to izravno provjerite, osim posredno, preko toènosti konaènog rezultata.

U naredbi `#define` moguće je navesti samo makro ime – u tom slučaju se ono definira na neku neodređenu vrijednost:

```
#define SOLO
```

Ovakva pretprocesorska naredba natjerat æe pretprocesor da ukloni sve pojave niza `SOLO` iz izvornog kôda. Taj oblik naredbe poprima puni smisao u paru s naredbama za uvjetno prevođenje `#ifdef` i `#ifndef`, koje æemo upoznati kasnije u ovom poglavlju.

16.3.1. Trajanje definicije

Vrijednost makro imena može se promijeniti novom naredbom `#define`:

```
#define POZDRAV "Dobar dan!"
cout << POZDRAV << endl;                // Dobar dan!

#define POZDRAV "Hvala, tebi također."
cout << POZDRAV << endl;                // Hvala, tebi...
```

Neki prevoditelji æe odaslati upozorenje o promjeni definicije makro imena.

Jednom definirano makro ime ostaje definirano do kraja tekuće datoteke ili do naredbe `#undef` kojom se on poništava. Zbog toga će, nadovežemo li na gornji kôd sljedeće naredbe:

```
#undef POZDRAV
cout << POZDRAV << endl;    // pogreška: nema više POZDRAV-a
```

Prevoditelj sada javlja pogrešku, jer je naredba ekvivalentna

```
cout <<  << endl;   // pogreška: dva operatora << uzastopce
```

Valja napomenuti da pod datotekom ovdje podrazumijevamo privremenu datoteku koju generira pretprocesor, zajedno s kôdom datoteka ukljuèenih naredbom `#include`.

16.3.2. Rezervirana makro imena

Postoje određena makro imena koja su rezervirana za specifiène namjene, te se ne smiju predefinirati ili uništiti naredbom `#undef`. Navedimo prvo imena koja su određena standardom, s njihovim kratkim opisom. Njihovu primjenu æemo ilustrirati kasnije u ovom poglavlju.

`__LINE__` Broj tekuæeg retka u datoteci izvornog kôda (decimalna konstanta).

`__FILE__` Ime tekuæe datoteke izvornog kôda (znakovni niz).

`__DATE__` Datum prevođenja datoteke izvornog koda. Datum je znakovni niz oblika "Mmm dd gggg", gdje je ime mjeseca troslovna kratica engleskog imena mjeseca.

`__TIME__` Vrijeme prevođenja datoteke izvornog kôda. Vrijeme je znakovni niz oblika "hh:mm:ss".

`__STDC__` Ovo makro ime ne mora biti definirano. Ako jest, vrijednost ovisi o implementaciji, tj. može se razlikovati za pojedine prevoditelje. Ako je ime definirano, ono oznaèava da se prevođenje provodi u skladu s ANSI standardom.

`__cplusplus` Ovo makro ime je definirano prilikom prevođenja datoteke C++ prevoditeljem.

Međutim, veæina prevoditelja definira još i neka dodatna imena, èiji se opis može naæi u pripadajuæim uputama.

Vrijednosti svih makro imena ostaju nepromijenjena kroz cijelu datoteku izvornog kôda, s izuzetkom `__LINE__` i `__FILE__`. Vrijednost makro imena `__LINE__` se automatski mijenja tijekom obrade izvornog kôda, ali se može promijeniti i pretprocesorskom naredbom `#line` (vidi odjeljak 16.5). Vrijednost makro imena `__FILE__` se također može promijeniti naredbom `#line`.

Rezervirana makro imena prvenstveno se koriste tijekom razvoja programa, za otkrivanje i lociranje pogrešaka u kôdu. Želimo li da se prilikom izvođenja izvedbenog kôda nastalog prevođenjem neke naredbe ispiše redni broj retka te naredbe u datoteci izvornog kôda, ime datoteke izvornog kôda, te datum i vrijeme prevođenja (koliko god to bilo besmisleno), iza te naredbe dodat ćemo naredbu:

```
cout << "Redak br. " << __LINE__ - 1 << endl
     << "Datoteka: " << __FILE__ << endl
     << "Datum prevođenja: " << __DATE__ << endl
     << "Vrijeme prevođenja: " << __TIME__ << endl;
```

16.3.3. Makro funkcije

Pomoæu naredbe `#define` mogu se definirati i *makro funkcije*. Makro funkcija je simbol stvoren pomoæu `#define` naredbe, koji može prihvaæati argumente poput funkcija pisanih u jeziku C++. Opæenito se makro funkcija definira kao:

```
#define imeMakroFun( lista_argumenata ) ( tijelo_funkcije )
```

Pretprocesor æe simbolièki poziv makro funkcije u izvornom kôdu nadomjestiti tijelom makro funkcije s umetnutim stvarnim argumentima. Na primjer, pretpostavimo da je definirana makro funkcija

```
#define KVADRAT(x) ((x) * (x))
```

Napišemo li sada negdje u programu:

```
c = KVADRAT(10);            // izvorni kôd
```

nju æe pretprocesor nadomjestiti kôdom:

```
c = ((10) * (10));          // nakon pretprocesora
```

Međutim, makro funkcije znaju vrlo malo o sintaksi C++ jezika, veæ “vrte svoj film”. Stoga æe sljedeæi poziv makro funkcije:

```
d = KVADRAT(++i);           // nešto tu zaudara...
```

pretprocesor razviti u kôd

```
d = ((++i) * (++i));          // nakon pretprocesora
```

Kao što vidimo, argument æe se uveæati dva puta, a rezultat pouzdano neæe biti kvadrat (èak ni tako uveæanog) broja. Konaèni ishod æe se oèito razlikovati od onoga što bi naivni korisnik u prvi mah oèekivao.

Čitatelj je zacijelo primijetio mnoštvo zagrada u definiciji makro funkcije. One su neophodne da bi se sačuvala hijerarhija operacija u slučaju da se kao argument makro funkcije navede neki složeni izraz. Da smo izostavili zagrade u definiciji, na primjer:

```
#define KVADRAT_UPITNI(x) x * x
```

poziv makro funkcije

```
d = KVADRAT_UPITNI(2 + 3);
```

pretprocesor bi razvio u naredbu:

```
d = 2 + 3 * 2 + 3;      // nije ni kvadrat više ono
                        // što je nekoć bio
```

U ovako razvijenom kôdu, zbog razlièitog prioriteta operacija, rezultat neæe odgovarati oèekivanom.

Kao što vidimo, makro funkcije nemaju uvida u sintaksu jezika C++, pa tako ne znaju ništa o pravilima konverzije tipova, dozvoljenim operatorima, hijerarhiji operacija. Budući da prevoditelj ne obrađuje izvornu datoteku, već datoteku koju je generirao pretprocesor (a korisnik u normalnim okolnostima nema uvid u tu međudatoteku), vrlo je teško ući u trag pogreškama koje nastaju zbog nepravilno definiranih makro funkcija. Uostalom, programski jezik C++ pruža pogodnosti poput predložaka funkcija, preopterećenja funkcija i umetnutih funkcija, koje čine makro funkcije potpuno nepotrebnima. Primjena makro funkcija, naprotiv, jest opravdana za traženje pogrešaka u programu, o čemu će biti riječi kasnije u ovom poglavlju.

U C++ programima valja izbjegavati korištenje makro funkcija. Primjena makro funkcija odražava nedosljednost programskog jezika, programa ili programera [Stroustrup91].

16.3.4. Operatori za rukovanje nizovima

Pri pisanju makro funkcija ili makro definicija, korisniku stoje na raspolaganju dva operatora koji omoguæavaju rukovanje simbolièkim imenima (engl. *token*) iz programa: operator `#` za pretvorbu simbolièkog imena u znakovni niz, te operator `##` za stapanje simbolièkih imena.

Operatorom `#` je moguće simboličko ime pretvoriti u znakovni niz. Ilustrirajmo to sljedećim primjerom: prilikom razvoja programa želimo provjeriti vrijednosti pojedinih

varijabli, tako da ih ispisujemo iz programa. Da bismo znali koja ispisana vrijednost odgovara kojoj varijabli, uz vrijednost moramo ispisati i ime varijable. Zdravo-seljački pristup bi bio umetanje naredbe za ispis oblika:

```
cout << "varijabla = " << varijabla << endl;
```

na svakom željenom mjestu u kôdu. Naravno, gore navedeni naziv varijable `varijabla` je proizvoljan i njega treba varirati shodno imenu varijable koju želimo ispisati.

Postupak možemo djelomično pojednostavniti tako da definiramo makro funkciju koja će kao argument prihvaćati ime varijable koju želimo ispisati:

```
#define PROVJERA(varijabla) \
            cout << #varijabla " = " << varijabla << endl
```

Operator `#` ispred simbolièkog imena pretvorit æe ga u znakovni niz, onakav kakvim ga podrazumijeva jezik C++. Za ispis æe sada trebati samo umetnuti naredbe oblika:

```
PROVJERA(a);
```

Gornja naredba se prilikom obrade pretprocesorom pretvara u

```
cout << "a" " = " << a << endl;
```

Operator `##` omoguæava stapanje simbolièkih imena. To se ponekad koristi za generiranje novih simbolièkih imena. Na primjer, definiramo li makro funkciju:

```
#define NESTO_SASVIM_NOVO(i, j) (i ## j)
```

poziv u naredbi

```
int NESTO_SASVIM_NOVO(x, 6);
```

æe generirati novo simbolièko ime, jer æe pretprocesor ovu naredbu razviti u

```
int x6;
```

16.4. Uvjetno prevoðenje

Makro naredbe za uvjetno uključivanje omoguæavaju da se u kôd koji se prevodi uključuju različiti programski odsjeèci, shodno rezultatu nekog logièkog izraza. To nam omoguæava da samo promjenom definicije makro imena koje se koristi u tom logièkom izrazu, generiramo razlièite inaèice istog kôda.

Postoje tri naredbe za ispitivanje uvjeta uvjetnog prevođenja: `#if`, `#ifdef` i `#ifndef`. Blok naredbi koji započinje s bilo kojom od te tri naredbe mora biti zaključen

`#endif` naredbom. Unutar tog bloka može postojati više neovisnih grana, odvojenih s jednom ili više naredbi `#elif` ili (maksimalno) jednom naredbom `#else`. Kao što vidimo, blokovi za uvjetno uključivanje kôda podsjećaju strukturom na `if`-blokove u jeziku C++, s time da se blokovi ne ograđuju vitičastim zagradama, već pretprocesorskim naredbama.

Naredbom `#if` moguće je na makro ime primijeniti neki poredbeni operator. Na primjer, želimo li napraviti više inačica našeg programa s ispisom poruka na različitim jezicima, možemo na početku programa definirati makro ime `JEZIK` i postaviti ga na određenu vrijednost. Unutar `#if-#elif/#else-#endif` blokova definirat ćemo poruke za pojedine jezike:

```
#define JEZIK HRVATSKI
//...
#if    JEZIK == ENGLISH
char *pozdrav = "Do you speak English?";
#elif  JEZIK == DEUTSCH
char *pozdrav = "Sprechen Sie Deutsch?";
#elif  JEZIK == HRVATSKI
char *pozdrav = "Zborite li rvatski?";
#else       // umjesto esperanta:
char *pozdrav = "Говориш ли ти језиком који цео свет разуме?";
#endif
//...
cout << pozdrav << endl;
```

Svaki puta kada želimo generirati inaèicu programa za drugi jezik, dovoljno je promijeniti definiciju niza `JEZIK` i ponovo prevesti i povezati program – prevoditelj æe sam ukljuèiti poruke na odgovarajuæem jeziku.

Kada ne bismo koristili gornji pristup, trebali bismo generirati zasebne datoteke izvornog kôda za svaki jezik ili bismo, prilikom promjene jezika, trebali prepisivati sadržaje svih znakovnih nizova za taj jezik. Osim toga je i proširenje za nove jezike (posebice ako se tekstovi svih poruka grupiraju) vrlo jednostavno.

U ispitivanjima uvjeta mogu se koristiti svi logički i poredbeni operatori koji su dozvoljeni i u jeziku C++ (`!`, `&&`, `||`, `==`, `<`, `>`, `!=`).

Pretprocesorskom naredbom `#ifdef` ispituje se je li neko makro ime definirano. Ako jest, naredbe koje slijede se prevode; u protivnom se kôd do pripadajuće `#elif`, `#else` ili `#endif` naredbe ne uključuje u prevođenje.

Pretpostavimo da razvijamo neki program za koji želimo prirediti i pokaznu (*demo*) inačicu, koja će se od radne razlikovati po tome što neće omogućavati ispis rezultata na pisač. Pomoću pretprocesorskih naredbi možemo to riješiti na sljedeći način:

```
#define DEMO
//...
#ifdef DEMO
cout << "Ovo je pokazna inačica mog programa za rješavanje "
        "Besselove diferencijalne jednadžbe koja ne "
```

```
        "dozvoljava ispis na vašem pisaču" << endl;
#else
IspisiNaPisacu(rezultat);
#endif
```

Uz ovakav `#define` generirat æe se demo verzija programa; izbacivanjem naredbe `#define`, makro ime `DEMO` æe biti nedefinirano, te æe se generirati radna verzija.

Komplementarna pretprocesorskoj naredbi `#ifdef` jest naredba `#ifndef` koja ispituje je li makro ime koje slijedi nedefinirano: ako je nedefinirano, prevode se naredbe koje slijede; u protivnom se preskaču sve naredbe do pripadajuće `#endif` direktive.

Sklopovi naredbi `#if`, `#ifdef` i `#ifndef` redovito se koriste u programima u kojima se izvorni kôd rasprostire kroz više datoteka, pa ćemo se s njihovom praktičnom primjenom još podrobnije upoznati u poglavlju 15.

16.4.1. Primjena uvjetnog prevođenja za pronalaženje pogrešaka

Uvjetno prevođenje u kombinaciji s makro funkcijama se èesto koriste za olakšavanje pronalaženja pogrešaka prilikom razvoja programa. Na primjer, moguæe je definirati makro funkciju `KONTROLA()` koja æe ispisivati vrijednost parametra na standardni izlaz za pogreške koji kasnije možemo analizirati i zakljuèiti zašto neki program ne radi. Pri tome æemo definirati funkciju tako da se jednim potezom svi ispisi mogu iskljuèiti, i ponovnim prevođenjem izbaciti iz kôda. Na taj naèin dobili smo kôd koji æe s jedne strane omoguæavati kontrolu programa prilikom razvoja, dok s druge strane komercijalna verzija neæe biti optereæena nepotrebnim kôdom (za koji se èesto pokaže da i nije baš tako "nepotreban").

Prisutnost ili odsutnost kontrolirajućih ispisa ćemo nadzirati makro imenom `NDEBUG`. To ime se koristi u datoteci zaglavlja `assert.h`. Ta datoteka sadržava makro funkciju `assert()` koja, ako izraz koji je naveden kao parametar funkciji nakon izračunavanja daje vrijednost `false`, ispisuje na ekran poruku "Assertion failed:". Ako, pak, definiramo simbol `NDEBUG`, sve `assert()` makro funkcije će se prilikom prevođenja zamijeniti praznim naredbama čime će se provjere izbaciti iz programa. Evo mogućeg kôda makro funkcije `KONTROLA()`:

```
#ifdef NDEBUG
#   define KONTROLA(izraz) ((void)0)
#else
#   define KONTROLA(izraz) cerr << izraz << endl
#endif
```

Sada ako na određenom mjestu u programu želimo provjeriti vrijednost varijable `a`, to možemo uèiniti ovako:

```
KONTROLA(a);
```

Ako je na određenom mjestu u programu smislena vrijednost varijable `a` razlièita od nule, umjesto ispisa koji bi mogao samo dodatno opteretiti ekran pogrešaka, možemo koristiti makro funkciju `assert()`:

```
assert(a != 0);
```

16.5. Ostale pretprocesorske naredbe

Pretprocesorska naredba `#error` ima opæeniti oblik:

```
#error poruka
```

gdje je *`poruka`* tekst koji želimo ispisati. Ona æe prilikom prevođenja prouzroèiti ispis poruke oblika:

```
Error: ImeDatoteke BrLinije : Error directive: poruka
```

Ova naredba se uglavnom koristi za prekid prevođenja ako se u nekom ispitivanju utvrdi da nisu zadovoljeni svi neophodni uvjeti. Naredbu treba u tom sluèaju umetnuti unutar bloka za uvjetno prevođenje koji ispituje tražene uvjete.

Za ilustraciju, osvrnimo se na primjer našeg multi-jezičnog programa na stranici 482. Želimo li se osigurati da makro ime `JEZIK` uvijek bude definirano, ubacit ćemo sljedeće naredbe:

```
#ifndef JEZIK
#error Makro ime JEZIK mora biti definirano.
#endif
```

Zaboravimo li definirati `JEZIK`, pretprocesor æe onemoguæiti prevođenje, uz ispis imena datoteke i broja linije u kojoj se nalazi navedena `#error` naredba te poruke navedene u naredbi.

Naredbom `#pragma` se podešavaju parametri pretprocesora ili prevoditelja. Općeniti oblik naredbe je:

```
#pragma parametar
```

`parametar` je niz koji podešava određeno svojstvo prevoditelja. Ti nizovi su određeni implementacijom, pa zainteresiranog èitatelja upuæujemo na priruènik s uputama za prevoditelj koji koristi.

Pretprocesorskom naredbom `#line` može se promijeniti vrijednost makro imena `__LINE__` i `__FILE__`, što se ponekad može iskoristiti za referenciranja pri traženju pogrešaka. Naredba ima općeniti oblik:

```
#line broj_retka "ime_datoteke"
```

Ime datoteke se može izostaviti – u tom sluèaju `__FILE__` ostaje nepromijenjen.

Pretpostavimo da smo pri pisanju nekog programa dio kôda neke datoteke jednostavno preslikali u naš novi kôd. Da bismo pri provjeri rada programa znali da se radi o kôdu umetnutom iz druge datoteke, ispred umetnutih naredbi ubacit ćemo naredbu kojom ćemo redefinirati vrijednosti imena `__LINE__` i `__FILE__`. Naravno da iza umetnutog odsječka treba opet vratiti makro imena na stare vrijednosti:

```
// ...

#line 1 "umetnuto.cpp"

// slijedi umetnuti kôd

#line 134 "novikod.cpp"
// slijedi novi kôd
```

16.6. *Ma èa æe meni pretprocesor?*

Kao što smo vidjeli, upotreba pretprocesorskih naredbi je kod pisanja programa u jeziku C++ èesto problematièna. Sam jezik C++ sadrži mehanizme koji su u C-u bili ostvarivi iskljuèivo preko makro funkcija. U suštini, primjena pretprocesorskih naredbi trebala bi se ogranièiti na sljedeæa tri sluèaja:

- Ukljuèivanje datoteka `#include` naredbom.
- Uvjetno ukljuèivanje dijelova kodova za kontrolne ispise prilikom razvoja programa.
- Uvjetno ukljuèivanje pri prevoðenju razlièitih izvedbi programa (na primjer s porukama na razlièitim jezicima, na razlièitim operacijskim sustavima).

17. Organizacija kôda u složenim programima

Tiranija je bolje organizirana nego sloboda.

Charles Péguy 1873–1914, "Rat i mir"

Gotovo svaki "ozbiljniji" program se sastoji od nekoliko tisuæa ili desetaka tisuæa redaka izvornog kôda. Zbog složenosti i duljine, takve je programe neophodno pisati u odvojenim modulima. Pojedini modul se može razvijati nezavisno, pri èemu se programer koncentrira samo na problem koji taj modul rješava, a ne brine o složenosti cijelog sustava.

Također, takve programe nerijetko piše više programera ili čak cijeli timovi istovremeno, pa je i zbog efikasnosti posla neophodno rascjepkati kôd. U ovom poglavlju bit će dani upute i pravila kako dobro organizirati izvorni kôd u slučaju da ga je potrebno rasporediti u nekoliko datoteka. Također će biti govora o povezivanju kôda pisanog u C++ jeziku s programskim kôdom u drugim jezicima.

17.1. Zašto u više datoteka?

Primjeri koji su korišteni u ovoj knjizi su bili najjednostavniji moguæi (iako toga možda u danom trenutku niste bili svjesni) i za njih nije bilo potrebe kôd pohranjivati u odvojene datoteke. U veæini kraæih programa od nekoliko stotina redaka kôda, cijeli program se sasvim lijepo dade smjestiti u jednu datoteku i kao takav obraðivati. Meðutim, kod kompleksnijih problema se redovito ukazuje potreba za razbijanjem kôda u nekoliko odvojenih datoteka. Navedimo glavne razloge za razbijanje izvornog kôda u više datoteka.

Zamislimo da smo napisali program od desetak tisuća redaka izvornog kôda. Promijenimo li samo jednu naredbu (na primjer ime neke varijable ili vrijednost neke konstante), trebat će prevesti cjelokupni izvorni kôd i povezati dobiveni objektni kôd. Proces prevođenja cjelokupnog kôda u ekstremnim slučajevima može trajati čak i više od sat vremena. Naprotiv, ako se izvorni kôd rasprostire kroz nekoliko datoteka, tada treba prevesti samo datoteku u kojoj je ispravka napravljena, te pripadajući objektni kôd povezati s već gotovim objektnim kôdovima ostalih datoteka. Štoviše, današnji prevoditelji imaju ugrađene mehanizme kojima samostalno provjeravaju koje su datoteke izvornog kôda mijenjane od zadnjeg prevođenja/povezivanja, tako da programer ne treba eksplicitno navoditi koju datoteku treba prevoditi.

Druga stvar koja vam se može dogoditi jest da prevoditelj tijekom prevođenja odbije poslušnost i javi da nema dovoljno memorije za uspješni završetak operacije. Naime, prevoditelj prilikom prevođenja generira simboličku tablicu koja uspostavlja vezu između simboličkih naziva objekata i funkcija u izvornom kôdu i njihove adrese u

objektnom kôdu. Ako je broj tih naziva prevelik, memorijski prostor namijenjen za njihovo pohranjivanje će se prepuniti, što će onemogućiti daljnje prevođenje kôda. Istina, veličina tog prostora može se povećati, ali to rješenje nije vječno.

Treći argument za raspoređivanje izvornog kôda u više datoteka jest preglednost. Navedu li se definicije svih klasa u jednoj datoteci kôd će biti nepregledan – puno je praktičnije ako se definicije pojedinih klasa strpaju u zasebne datoteke. Time ćete si kasnije značajno olakšati možebitno uključivanje pojedinih klasa u neki drugi program.

Četvrti argument je vezan uz timski rad: ako program ne piše jedan programer već čitava ekipa, tada je daleko praktičnije ako svaki programer piše kôd u vlastitoj datoteci. Uostalom, i programeri znaju biti sitne duše koje ne vole da im se brlja po njihovim umotvorinama.

Poneki čitatelj će kao argument protiv navesti činjenicu da je daleko teže kontrolirati veći broj datoteka. Srećom, svi današnji prevoditelji/povezivači omogućavaju objedinjavanje više datoteka u cjeline – *projekte*. Takvi projekti dozvoljavaju čak da su datoteke izvornog kôda razbacane na više mjesta (u različitim imenicima, pa i na različitim diskovima).

Očito je da će prije ili kasnije svakom programeru “datotečna koža postati pretijesna” i da će, unatoč svim pokušajima odgađanja, u sudbonosnom trenutku u afektu otvoriti novu datoteku izvornog kôda. Kako se datoteke izvornog koda prevode zasebno, svakoj datoteci moraju biti dostupne deklaracije objekata i funkcija koje se u njoj koriste. Suštinski gledano, izvorni kôd mora biti jednako konzistentan kao kada je cijeli program napisan u jednoj datoteci. Ako se ta konzistentnost izgubi, povezivač će prijaviti pogrešku prilikom povezivanja.

Budući da postupak naknadnog razbijanja kôda može biti vrlo mukotrpan, od presudnog je značaja pravovremeno sagledati kompleksnost nekog problema i od početka krenuti s pravilnim pristupom.

17.2. Povezivanje

U poglavlju 1 opisano je kako se prevođenje izvornog kôda u izvedbeni sastoji od dva koraka: prevođenja i povezivanja. U prvi mah poèetniku nije jasna razlika izmeðu ta dva koraka. Štoviše, za programe èiji je izvorni kôd u cijelosti smješten u jednu datoteku nema svrhe razluèivati ta dva koraka – slika postaje jasnija u sluèaju kada je program razbijen u više modula.

Tijekom prevođenja provodi se sintaksna i semantička provjera, pri čemu se među ostalim provjerava odgovaraju li pozivi funkcija njihovim deklaracijama. Budući da se prevođenje provodi za svaki čimbeni modul zasebno, svaki od njih mora sadržavati deklaracije funkcija koje se unutar tog modula pozivaju. Na primjer, u datoteci `poziv.cpp` poziva se funkcija `cijaFunk()`, koja je definirana u datoteci `definic.cpp`. Da bi prevoditelj mogao pravilno prevesti kôd datoteke `poziv.cpp`, njemu prije poziva treba na neki način predočiti deklaraciju te funkcije:

```
poziv.cpp
```

```
void cijaFunk(int dajStoDas);       // deklaracija
// ...
    cijaFunk(0);                // poziv
```

Tek se prilikom povezivanja traži odgovarajuæa definicija funkcije i ako se ona ne pronaðe, povezivaè æe prijaviti pogrešku da definicija funkcije "te i te" nije pronaðena. Na primjer, ako je funkcija `cijaFunk()` definirana u modulu `definic.cpp` kao

```
definic.cpp
```

```
int cijaFunk(char *tkoTeSljivi) {   // definicija
    //...
}
```

povezivaè æe prijaviti pogrešku. Oèevidno to znaèi da prevoditelja možemo (èak i nehotice) "prevariti" tako da navedemo u nekom modulu deklaraciju nedefinirane funkcije (ili deklaraciju koja æe se razlikovati od definicije), meðutim na kraju æe povezivaè ipak prozreti naš podmukli pokušaj. (*You may fool all the people some of the time; you can even fool some of the people all the time; but you can't fool all of the people all the time*[†]).

Prevoditelj vidi samo modul koji prevodi pa ne može usporediti deklaraciju objekta ili funkcije sa stvarnom definicijom ako su one u odvojenim datotekama.

No valja napomenuti da prevoditelj neæe uvijek reagirati ovako promptno. U to se možemo uvjeriti ako u definiciji gornje funkcije uskladimo tip argumenta s tipom argumenta u deklaraciji, tj. ako u datoteci `definic.cpp` funkciju definiramo kao:

```
int cijaFunk(int dajStoDas) {   // promijenjena definicija
//...
}
```

Sada se deklaracija i definicija funkcije meðusobno slažu u potpisu (tj. broju i tipovima argumenata funkcije), ali se razlikuju u tipu povratne vrijednosti. Poziv funkcije u datoteci `poziv.cpp` bit æe preveden u skladu s deklaracijom funkcije u toj datoteci, tj. kao `void cijaFunk(int)`, dok æe definicija funkcije u datoteci `definic.cpp` biti prevedena kao `int cijaFunk(int)`. Funkcije su jednake po potpisu i povezivaè neæe uoèiti da se deklaracija i definicija funkcije razlikuju po tipu povratne vrijednosti. Posljedica toga æe biti pogreška u izvoðenju programa, pri pokušaju povratka iz

---

[†] "Možete varati sve ljude neko vrijeme, možete èak varati neke ljude cijelo vrijeme, ali ne možete varati sve ljude cijelo vrijeme" – Abraham Lincoln (1809-1865)

funkcije. Naime, generirani kôd funkcije æe pri izlasku iz nje na stog staviti cjelobrojni povratni broj kojeg, meðutim, pozivajuæi kôd neæe pokupiti buduæi da on za dotiènu funkcija oèekuje da je tipa `void`.

Ako deklaracija funkcije u jednoj datoteci i definicija istoimene funkcije u drugoj datoteci imaju jednake potpise, povezivač će shvatiti da se radi o istoj funkciji. Ako su se one razlikuju po tipu povratnih vrijednosti, nastupit će pogreška pri izvođenju.

Iz dosadašnjih razmatranja možemo naslutiti da je za pravilnu organizaciju kôda neophodno razluèiti doseg imena pojedinih identifikatora. Veæ smo nekoliko puta naglašavali da su objekti deklarirani unutar blokova vidljivi samo unutar tog bloka. To se odnosi kako na objekte deklarirane unutar funkcija, tako i na ugniježðene klase. Identifikatori koji su deklarirani izvan funkcija i klasa uglavnom su (uz èasne izuzetke koje æemo navesti) vidljivi u cijelom kôdu, bez obzira što se taj kôd može rasprostirati kroz više datoteka. Za identifikatore (tj. objekte i funkcije) koji su prilikom povezivanja vidljivi i u drugim modulima kaže se da imaju *vanjsko povezivanje* (engl. *external linkage*). Zbog toga u cijelom programu smije postojati samo jedan objekt, tip ili funkcija s tim imenom.

S druge strane, postoje objekti i funkcije koji su vidljivi samo unutar datoteke. Za njih se kaže da imaju *unutarnje povezivanje* (engl. *internal linkage*). Globalne umetnute (`inline`) funkcije i simboličke konstante imaju unutarnje povezivanje. Zato će umetnuta funkcija `svojaUsvojoj()` i simbolička konstanta `lokalnaKonstanta` u kôdu

```
prima.cpp
```

```
inline void svojaUsvojoj() {
    // tijelo funkcije
}

const int lokalnaKonstanta = 23;
```

biti nevidljivi izvan datoteke u kojoj su definirani. To znaèi da u drugim modulima možemo definirati istoimene umetnute funkcije, odnosno simbolièke konstante, a da se njihova imena ne sukobljavaju s navedenim identifikatorima. Na primjer, u nekoj drugoj datoteci možemo napisati

```
seconda.cpp
```

```
inline int svojaUsvojoj(int n) {    // potpuno nova funkcija
    // ...
}

const float lokalnaKonstanta = 12.; // ... i konstanta
```

Prilikom povezivanja obiju datoteka povezivaè neæe prijaviti pogrešku da se funkcija `svojaUsvojoj()` i objekt `lokalnaKonstanta` pojavljuju dva puta, jer je svaki od njih lokalan za datoteku u kojoj su navedeni.

Zanimljivo je spomenuti da se ime s vanjskim povezivanjem definirano u nekoj datoteci neće sukobiti s imenom objekta ili funkcije s unutarnjim povezivanjem u nekoj drugoj datoteci. To znači da će povezivanjem bilo koje od gornjih datoteka s datotekom `tertima.cpp` u kojoj su definirani funkcija i varijabla s vanjskim povezivanjem, povezivač stvoriti dvije funkcije `svojaUsvojoj()` i dvije varijable `lokalnaKonstanta`:

tertima.cpp

```
void svojaUsvojoj() {           // funkcija s vanjskim
                                // povezivanjem
    // ...
}

double lokalnaKonstanta = 210.; // varijabla s vanjskim pov.
```

Svi pozivi funkcije `svojaUsvojoj()` i sva dohvaæanja varijable `lokalnaKonstanta` iz programa povezivaè æe usmjeriti funkciji, odnosno varijabli iz datoteke `tertima.cpp`, osim ako se to ne èini iz modula u kojima su definirani istoimeni funkcija i objekt s unutarnjim povezivanjem. Naravno da ovakvo svojstvo predstavlja potencijalnu opasnost, koju valja izbjegavati pravilnom organizacijom kôda.

Korisnički tipovi definirani pomoću ključne riječi `typedef` su također vidljivi samo unutar datoteke u kojoj su definirani. Zbog toga se isti sinonim može koristiti u različitim datotekama za različite tipove objekata, odnosno funkcija:

prva.cpp

```
typedef int Tip;
// ...
```

druga.cpp

```
// donja deklaracija može miroljubivo i aktivno
// koegzistirati s onom u datoteci "prva.cpp":
typedef double Tip;
// ...
```

Za imena deklarirana unutar blokova se kaže da nemaju povezivanje. Takva se imena (konkretno imena lokalnih klasa i pobrojenja) ne mogu koristiti za deklaraciju objekata izvan bloka u kojemu su definirani:

```
void funkcija() {
    class LokalnaKlasa {        // klasa bez povezivanja
    //...
    };
}

LokalnaKlasa classaLocale;      // pogreška
```

Ako za neki identifikator želimo povezivanje koje nije podrazumijevano, tip povezivanja možemo promijeniti kljuènim rijeèima `static` i `extern`, o èemu æe biti rijeèi u sljedeæem odjeljku.

17.2.1. Specifikatori `static` i `extern`

Dodavanjem kljuène rijeèi `static` ispred deklaracije objekta, funkcije ili anonimne unije, eksplicitno se pridjeljuje interno povezivanje. To omoguæava da se u pojedinim modulima definiraju razlièiti globalni objekti ili funkcije s jednakim imenima, a da zbog toga ne doðe do sukoba imena prilikom povezivanja.

U sljedećem primjeru će navedene objektne datoteke nakon povezivanja imati svaka svoje inačice funkcije `f()`, varijable `broj` i anonimne unije:

suveren1.cpp

```
static int f(int argument) {
    //...
}
static float broj;
static union {
    int br;
    char ch;
};
```

suveren2.cpp

```
static void f() {
    //...
}
static char broj;
static union {
    int br;
    char ch;
};
```

Pažljiviji èitatelj se zasigurno sjeæa da smo kljuènu rijeè `static` veæ upoznali i to u dva navrata: kod funkcija i kod klasa. U oba sluèaja ona je imala razlièito znaèenje. Kod funkcija kljuèna rijeè `static`, primijenjena na lokalne objekte unutar funkcije, èini te

objekte trajnima – pri izlasku iz funkcije se statièki objekt ne uništava, veæ ostaje nedodirljiv (i nepromijenjen) do ponovnog poziva funkcije. U klasama se kljuènom rijeèi `static` pojedini podatkovni ili funkcijski èlan proglašava jedinstvenim za sve objekte te klase. Sada smo upoznali i treæe znaèenje kljuène rijeèi `static`, a to je da globalne objekte i funkcije èini "privatnima" za pripadnu datoteku.

> Preporučljivo je sve globalne objekte i funkcije deklarirati kao statičke, osim ako postoji očita potreba da se oni dohvaćaju iz drugih modula.

Time se smanjuje vjerojatnost sukoba globalnih imena iz razlièitih modula, do kojeg može lako doæi ako ih pišu razlièiti programeri. Osim toga, deklaracija `static` primijenjena na funkcije može doprinijeti boljoj optimizaciji kôda prilikom prevođenja.

Ključnom riječi `extern` se podrazumijevano unutarnje povezivanje nekog identifikatora pretvara u vanjsko. Ako želimo, primjerice, simboličke konstante učiniti dostupnima i iz druge datoteke, to možemo učiniti na sljedeći način:

svecnst1.cpp

```
extern const float pi = 3.1415926;  // definicija
```

svecnst2.cpp

```
// samo deklaracija - definicija je negdje drugdje:
extern const float pi;
cout << pi << endl;     // ispisuje 3.14159...
```

Umetanjem kljuène rijeèi `extern` ispred deklaracije prevoditelju se daje na znanje da je taj objekt (misli se bilo na objekt neke klase, varijablu ili funkciju) definiran u drugom modulu. Ilustrirajmo to primjerom programa razdvojenim u dvije datoteke:

johann.cpp

```
int bwv = 565;
int toccata() {              // definicija funkcije
    //...
}
```

bastian.cpp

```
extern int bwv;              // samo deklaracija
extern int toccata();        // također

float fuga() {               // deklaracija & definicija
```

```
        bwv = toccata();
    }
```

U drugom modulu koriste se varijabla `bwv` i funkcija `toccata()` definirane u prvom modulu. Da bi prevoditelj mogao potpuno "obraditi" drugi modul, moraju mu biti dostupne deklaracije te varijable, odnosno funkcije kako bi znao njihove tipove. Stoga su deklaracije navedene na poèetku datoteke `bastian.cpp`. Kljuènom rijeèi `extern` ispred deklaracija varijable `bwv` i funkcije `toccata()` eksplicitno se naglašava da je ta varijabla definirana u nekom drugom modulu i da je navedena naredba iskljuèivo deklaracija. Izostavi li se ta kljuèna rijeè, naredba

```
int bwv;
```

æe postati definicijom globalne varijable `bwv`. Poput svih globalnih i statièkih objekata, buduæi da nema eksplicitno pridružene vrijednosti, varijabla `bwv` æe se inicijalizirati na podrazumijevanu vrijednost 0. Kôd u datoteci `bastian.cpp` æe biti preveden bez pogreške, ali æe prilikom povezivanja biti dojavljena pogreška da u programu postoji još jedan objekt s istim imenom, onaj definiran u datoteci `johann.cpp`.

Naglasimo da sama ključna riječ `extern` nije dovoljni jamac da bi se naredba interpretirala kao deklaracija. Naime, ako unatoč ključnoj riječi `extern`, varijabli pridružimo vrijednost:

```
extern int bwv = 538;
```

naredba æe postati definicijom. Prevoditelj æe zbog operatora pridruživanja jednostavno zanemariti kljuènu rijeè `extern`.

U deklaraciji funkcije `toccata()` ključna riječ `extern` nije nužna, jer prevoditelj sam može prepoznati da se radi o deklaraciji, budući da iza zagrada `()` slijedi znak točka-zarez. Stoga se u deklaracijama vanjskih funkcija ključna riječ `extern` redovito izostavlja, no ako se stavi, nije pogrešna.

Zadatak. *Razmislite zašto će povezivač prijaviti pogreške tijekom povezivanja sljedeća dva modula (pretpostavite da se samo ta dva modula čine cijeli program):*

```
modul_1.cpp
```

```
int crniVrag = 1;
int zeleniVrag = 5;
extern float zutiVrag;
```

modul_2.cpp

```
int crniVrag;                // pogreška kod povezivanja
extern double zeleniVrag;    // pogreška kod izvođenja
extern float zutiVrag;       // pogreška kod povezivanja
```

U gornjem zadatku, cjelobrojna varijabla `crniVrag` je inicijalizirana dva puta: u `modul_1.cpp` na vrijednost 1, a u `modul_2.cpp` na vrijednost 0. Naime, kao što je veæ reèeno, globalne varijable bez prethodeæe kljuène rijeèi `extern` se, èak i ako nije eksplicitno specificirana neka vrijednost, inicijaliziraju podrazumijevanom vrijednošæu nula. Varijabla `zeleniVrag` je u modulima deklarirana razlièitog tipa, jednom kao `int`, a drugi puta kao `double`. Veæina povezivaèa neæe uoèiti razliku u tipovima, te æe povezati kôd, što æe onda prouzroèiti pogrešku kod izvođenja. Varijabla `zutiVrag` je u oba modula deklarirana kao vanjska – nedostaje njena definicija.

Ključna riječ `extern` je neophodna i prilikom povezivanja kôda pisanog u jeziku C++ s izvornim kôdom pisanim u nekom drugom programskom jeziku, o čemu će biti više govora u odjeljku 17.6.

17.3. Datoteke zaglavlja

U poglavljima o funkcijama i klasama naglasili smo razliku između deklaracije i definicije funkcije. Deklaracijom funkcije, odnosno klase, stvara se samo prototip, koji se konkretizira tek navođenjem definicije. Iako je svaka definicija ujedno i deklaracija, pod deklaracijom u užem smislu podrazumijevamo samo navođenje prototipa. Deklaracije se smiju ponavljati, ali se one moraju podudarati po tipu (izuzetak su preoptereæene funkcije i operatori). Naprotiv, definicija smije biti samo jedna – u protivnom æe prevoditelj prijaviti pogrešku. Pritom je neophodno da funkcija, odnosno klasa, mora biti deklarirana prije prvog poziva funkcije, odnosno instantacije objekta klase. Ovo je neophodno zato da bi prevoditelj veæ prilikom prvog nailaska na poziv funkcije ili instantaciju klase mogao provjeriti ispravnost poziva te generirati ispravan objektni kôd, provodeæi eventualno potrebne konverzije tipova.

U složenijim programima se često neka funkcija poziva iz različitih datoteka, ili se na osnovu neke klase stvaraju objekti u različitim datotekama. Više nego očito jest da deklaracije funkcije/klase moraju biti izdvojene od definicija i dostupne cjelokupnoj “datotečnoj klijenteli”. Jedna mogućnost jest da se neophodne deklaracije prepišu na početke odgovarajućih datoteka, ali je odmah uočljiv nedostatak ovakvog rješenja: prilikom promjene u deklaraciji neke funkcije ili klase treba pretražiti sve datoteke u kojima se ona ponavlja te provesti ručnu izmjenu. Osim što je ovakvo pretraživanje mukotrpno, ono je podložno i mogućim pogreškama.

Daleko je efikasnije deklaracije izdvojiti u zasebne datoteke zaglavlja, koje će se pretprocesorskom naredbom `#include` uključivati na početku svake datoteke gdje je to neophodno. U ovom slučaju, izmjena u deklaraciji se obavlja samo na jednom mjestu, tako da ne postoji mogućnost pogreške prilikom višekratnog ispravljanja. Radi preglednosti se datotekama zaglavlja daje isto ime kao i datotekama koje sadrže

pripadajuće definicije, ali se imenu datoteka zaglavlja umjesto nastavaka `.cpp`, `.c` ili `.cp` dodaje nastavak `.h` (ponekad `.hpp`).

Što valja staviti u datoteke zaglavlja? Budući da su objekti i funkcije s vanjskim povezivanjem vidljivi u svim modulima, ako želimo omogućiti pravilno prevođenje pojedinih modula, očito je neophodno navesti deklaracije svih objekata i funkcija koji će biti korišteni izvan modula u kojem su definirani. Kako često ne možemo unaprijed znati hoće li neka funkcija ili objekt zatrebati u nekom drugom modulu, nije na odmet u datotekama zaglavlja navesti deklaracije svih (nestatičkih) globalnih funkcija i objekata. Osim toga, navođenjem svih deklaracija u datoteci zaglavlja kôd će biti pregledniji, jer će sve deklaracije (osim za statičke objekte i funkcije) biti na jednom mjestu.

Vjerujemo da je svakom jasno da u datoteke zaglavlja također treba staviti definicije objekata i funkcija s unutarnjim povezivanjem (`inline` funkcije i simboličke konstante), želimo li da oni budu dohvatljivi i iz drugih datoteka. Isto tako, u datotekama zaglavlja treba navesti definicije tipova (deklaracije klasa) i predloške.

Radi preglednosti, nabrojimo po točkama glavne kandidate za navođenje u datotekama zaglavlja:

- Komentari, uključujući osnovne generalije vezane uz datoteku (autori, opis, prepravke):

```
/**********************************************************

Program:    Moj prvi složeni program
Datoteka:   Slozeni.h
Autori:     Boris (bm)
            Julijan (jš)
Izmjene:    31.11.96.   (jš) dodana funkcija xyz()
            29.02.97.   (bm) izbačen SudnjiDan

**********************************************************/
```

- Pretprocesorske naredbe `#include` za uključivanje drugih datoteka zaglavlja, najčešće standardnih biblioteka:

```
#include <iostream.h>
#include "MyString.h"
```

  Najčešće se ipak te naredbe navode u samoj datoteci izvornog kôda.

- Definicije tipova (deklaracije klasa i struktura):

```
class KompleksniBroj {
private:
    double Re;
    double Im;
public:
    //...
};
```

Definicije pojedinih funkcijskih i podatkovnih èlanova klasa navode se u samoj datoteci izvornog kôda. Izuzetak su, naravno, umetnuti (`inline`) funkcijski èlanovi koji se moraju navesti unutar deklaracije klase. Razlog tome je što prevoditelj prilikom prevoðenja mora imati dostupan kôd koji æe umetnuti na mjesto poziva.

- Deklaracija i definicije predložaka (u nekim sluèajevima):

```
template <class Tip>    // deklaracija
class Tablica {
private:
    Tip *pok;
public:
    Tablica(int n);
    // ...
};

template <class Tip>
Tablica<Tip>::Tablica(int n) {
    // ...
}
```

Naime, slièno umetnutim funkcijama, izvorni kôd predloška mora biti dostupan prevoditelju prilikom prevoðenja kako bi mogao instancirati predložak potreban broj puta. Baratanje s predlošcima u više datoteka izvornog koda dosta je složeniji problem, èije rješenje ovisi o razvojnoj okolini koja se koristi. Posljednja verzija C++ standarda nudi moguænost eksplicitne instantacije predložaka, èime se može izbjeæi stavljanje definicije predložaka u datotekama zaglavlja. Ovo æemo potanko objasniti u zasebnom odjeljku.

- Deklaracije funkcija:

```
extern void ReciMiReci(const char *poruka);
```

- Definicije umetnutih (`inline`) funkcija:

```
inline double kvadrat(double x) { return x*x };
```

- Deklaracije globalnih podataka koji moraju biti dohvatljivi iz razlièitih modula:

```
extern char *verzijaPrograma;
```

- Definicije konstanti koje moraju biti dohvatljive iz razlièitih modula:

```
const float e = 2.718282
```

- Pobrojenja:

```
enum neprijatelj { unutarnji, vanjski, VanOvan, Feral };
```

- Makro definicije i makro funkcije:

```
#define PI  3.14159265359
#define abs(a) ((a < 0) ? -a : a)
```

Gore navedene toèke èine samo prijedlog – one nisu odreðene pravilima jezika C++, no do gornjih pravila su došli programeri tokom dugih zimskih veèeri provedenih uz raèunalo. Iako u deklaracijama klasa i funkcija u datotekama zaglavlja nije neophodno navoditi imena argumenata (dovoljno je navesti tipove), zbog preglednosti i razumljivosti deklaracija one se stavljaju. Neki programeri preferiraju da se u deklaraciji navede dulje ime da bi se lakše shvatilo znaèenje pojedinog argumenta, dok u definiciji koriste skraæena imena da bi si skratili "muke" utipkavanja prilikom pisanja tijela funkcije.

Sada nakon što znamo što se obično stavlja u datoteku zaglavlja, navedimo i što ne valja tamo stavljati :

- Definicije funkcija:

```
void IspisStraga(const char *tekst) {
    int i = strlen(tekst);
    while (i-- > 0) cout << *(tekst + i);
    cout << endl;
}
```

- Definicije objekata:

```
int a;
```

  Ako u datoteku zaglavlja stavimo definiciju objekta, onda æemo u svakoj datoteci u koju ukljuèimo datoteku zaglavlja dobiti po jedan globalni simbol navedenog imena (u gornjem primjeru to je varijabla `i`). Takav program æe prouzroèiti pogreške prilikom prevoðenja.

- Definicije konstantnih agregata, na primjer polja:

```
const char abeceda = {'a', 'b', 'c', 'd'};
```

  Argument protiv navoðenja konstantnih argumenata u datotekama zaglavlja je pragmatièan: veæina prevoditelja ne provjerava jesu li generirane redundantne kopije agregata [Stroustrup91].

17.4. Organizacija predložaka

Organizaciju kôda jezik C++ nasljeðuje primarno od jezika C. Predlošci su koncept koji se ne uklapa glatko u opisani model datoteka zaglavlja i datoteka izvornog kôda. Zbog toga je rukovanje predlošcima u veæim programima dosta složeno.

U starijoj varijanti jezika C++ u kojoj nije postojao operator za eksplicitnu instantaciju predloška imali smo nekoliko mogućnosti, od kojih niti jedna nije bila previše sretna. Mogućnost koju su koristili mnogi programeri, a koje su zahtijevale i neke implementacije, jest bila smjestiti definiciju predloška u datoteku zaglavlja. Tako bi u svakoj datoteci koja uključuje to zaglavlje, predlošci bili automatski dostupni, te bi se prilikom prevođenja željeni predlošci instancirali. Ako bi više datoteka izvornog kôda koristilo isti predložak iz neke datoteke zaglavlja, na kraju bismo dobili objektni kôd koji bi sadržavao nekoliko instanci iste funkcije. Povezivač bi zatim morao izbaciti sve definicije osim jedne. Takvo rješenje definitivno nije sretno: osim što ovisimo o dobroj volji povezivača da nam skrati izvedbeni kôd, moguće je zamisliti predložak koji će u različitim datotekama izvornog kôda biti instanciran u različiti izvedbeni kôd. Time se krši osnovno C++ pravilo da svaki predložak mora imati točno jednu definiciju, a dobiveni izvedbeni kôd zasigurno ne bi bio ispravan.

Druga mogućnost, koja dosta ovisi o programskoj okolini u kojoj radimo, jest samo deklarirati predložak u datoteci zaglavlja, a posebnu datoteku u kojoj su definicije staviti prevoditelju na raspolaganje. Prilikom povezivanja, ako bi se ustanovilo da instanca nekog predloška ne postoji za željeni tip, povezivač bi ponovo pozvao prevoditelj koji bi generirao traženi predložak, a postupak povezivanja bi se zatim ponovio. Niti ovakvo rješenje nije baš ugodno: postupak prevođenja sada je bio znatno produljen, jer bi se mnogo vremena gubilo na prebacivanje između prevođenja i povezivanja. Također, mnoge jednostavnije razvojne okoline nisu omogućavale da povezivač poziva prevoditelj.

Programeri su se tada dovijali na razne načine. Jedno od možda najčešće korištenih rješenja sastojalo se u tome da se u datoteku zaglavlja smjesti samo deklaracija predloška, a u zasebnu datoteku definicija predloška, te da se eksplicitno prisili prevoditelj da instancira predložak za sve korištene varijante parametara. To se može učiniti tako da se stvori neki globalni objekt ili da se pozove željena funkcija.

Promotrimo kako to izgleda na jednostavnom primjeru. Zamislimo da ćemo raditi s predloškom klase `Kompleksni` parametrizirane tipom koji određuje preciznost brojeva. Također, imat ćemo preopterećen operator `<<` za ispis na izlazni tok. Deklaraciju klase i operatora ćemo smjestiti u zaglavlje `kompl.h`, a definiciju predloška u datoteku `kompl.cpp`. U toj datoteci ćemo također prisiliti prevoditelj da instancira predloške `Kompleksni<int>` i `Kompleksni<double>` i pripadajuće operatore umetanja.

kompl.h

```
#include <iostream.h>

template <class Prec>
class Kompleksni {
private:
    Prec re, im;
public:
    Kompleksni(Prec i = 0, Prec b = 0) : re(i), im(b) {}
    Prec DajRe();   // ovi članovi namjerno nisu umetnuti
```

```
    Prec DajIm();   // kako bi se pokazala organizacija
    void PostaviReIm(Prec, Prec);
};

template <class Prec>
ostream &operator <<(ostream &, Kompleksni<Prec> &);
```

kompl.cpp

```
#include <iostream.h>
#include "kompl.h"

template <class Prec>
Prec Kompleksni<Prec>::DajRe() {
    return re;
}

template <class Prec>
Prec Kompleksni<Prec>::DajIm() {
    return im;
}

template <class Prec>
void Kompleksni<Prec>::PostaviReIm(Prec r, Prec i) {
    re = r; im = i;
}

template <class Prec>
ostream &operator <<(ostream &os, Kompleksni<Prec> &c) {
    os << c.DajRe();
    if (c.DajIm() >= 0) os << "+";
    os << c.DajIm() << "i";
    return os;
}

// sada idu forsirane instance klase i operatora:
static NekoristenaFunkcija() {
    Kompleksni<double> c1;
    Kompleksni<int> c2;
    cout << c1 << c2;
}
```

U gornjem primjeru imamo funkciju `NekoristenaFunkcija()` koja je uèinjena statièkom kako ne bi bila vidljiva izvan datoteke `kompl.cpp`. U njoj su deklarirana dva objekta klase `Kompleksni` i pozvani su operatori za ispis kako bi se prisilio prevoditelj da instancira predložak. Evo kako bi se navedeni predlošci koristili iz glavnog programa, navedenog u datoteci `pozovi.cpp`:

pozovi.cpp

```
#include <iostream.h>
#include "kompl.h"

int main() {
    Kompleksni<double> c1(5.0, 9.0);
    Kompleksni<int> c2(4, 3);
    cout << "Prvi broj je: " << c1 << ", a drugi je " << c2;
}
```

Ovakvo rješenje radi dobro, ako se ne bunimo na to što smo morali deklarirati nepotrebnu funkciju `NekoristenaFunkcija()`, samo zato da bismo instancirali predložak. Posljednja verzija standarda jezika podržava eksplicitnu instantaciju predložaka, pa programer može sam odrediti koje sve predloške treba generirati. Svaki put kada se naiđe na potrebu za time da se instancira predložak za određeni tip, u datoteku gdje je predložak definiran ubaci se dodatna naredba za instantaciju. Tako bi u našem sluèaju datoteka `kompl.cpp` izgledala ovako:

kompl.cpp

```
// definicija predloška je ista, samo ćemo navesti naredbe
// za eksplicitnu instantaciju predloška

template class Kompleksni<double>;
template class Kompleksni<int>;
template ostream &operator <<(ostream &,Kompleksni<double>);
template ostream &operator <<(ostream &,Kompleksni<int>);
```

Pri tome valja primijetiti da ako je predložak parametriziran korisnièkim tipom, na mjesto gdje se instancira predložak potrebno je ukljuèiti deklaraciju tipa. Kako bi takav pristup mogao dovesti do toga da se u jednu datoteku ukljuèuju mnoga zaglavlja, pogodnije je instancu predloška za svaki poseban tip smjestiti u zasebnu datoteku. Definicija predloška se tada stavlja u posebnu datoteku koja se ukljuèuje u svaku `.cpp` datoteku gdje se obavlja instantacija. U našem sluèaju to bi znaèilo da se datoteka `kompl.h` ne mijenja (jer je to datoteka koja se ukljuèuje u druge datoteke iz kojih se predložak poziva). Iz datoteke `kompl.cpp` bismo tada izbacili naredbe za eksplicitnu instantaciju predložaka i naredbe za ukljuèivanje drugih zaglavlja, a samo datoteku bismo preimenovali u datoteku `kompldef.h`. Instantacije `Kompleksni<double>` i `Kompleksni<int>` bismo smjestili u zasebne datoteke, na primjer `kompldbl.cpp` i `komplint.cpp`:

```
kompldbl.cpp
```

```
#include <iostream.h>   // za instantaciju operatora <<
#include "kompl.h"      // deklaracija predloška klase
#include "kompldef.h"   // definicija predloška

template class Kompleksni<double>;
template ostream &operator <<(ostream &,Kompleksni<double>);
```

```
komplint.cpp
```

```
#include <iostream.h>   // za instantaciju operatora <<
#include "kompl.h"      // deklaracija predloška klase
#include "kompldef.h"   // definicija predloška

template class Kompleksni<int>;
template ostream &operator <<(ostream &,Kompleksni<int>);
```

17.5. Primjer raspodjele deklaracija i definicija u više datoteka

Ilustrirajmo opisani pristup preko datoteka zaglavlja programom u kojem tražimo presjek kružnice i pravca. U programu æemo deklarirati i definirati tri klase: `Tocka`, `Pravac` i `Kruznica`. Klasa `Tocka` poslužit æe nam kao osnovna klasa za izvedenu klasu `Kruznica`, a takoðer æemo ju iskoristiti u konstruktoru klase `Pravac`. Glavnu funkciju æemo pohraniti u datoteku `poglavar.cpp`, dok æemo klase i njihove funkcijske èlanove definirati u zasebnim datotekama zaglavlja (`.h`), odnosno izvornog kôda (`.cpp`). Funkciju za ispis presjecišta smjestiti æemo u datoteke `presjek.h`, odnosno `presjek.cpp`. Na slici 17.1 shematski je prikazana struktura programa s oznaèenim ukljuèivanjima.

Klasu `Tocka` deklarirat ćemo u datoteci `tocka.h`, a njene članove ćemo definirati u `tocka.cpp`:

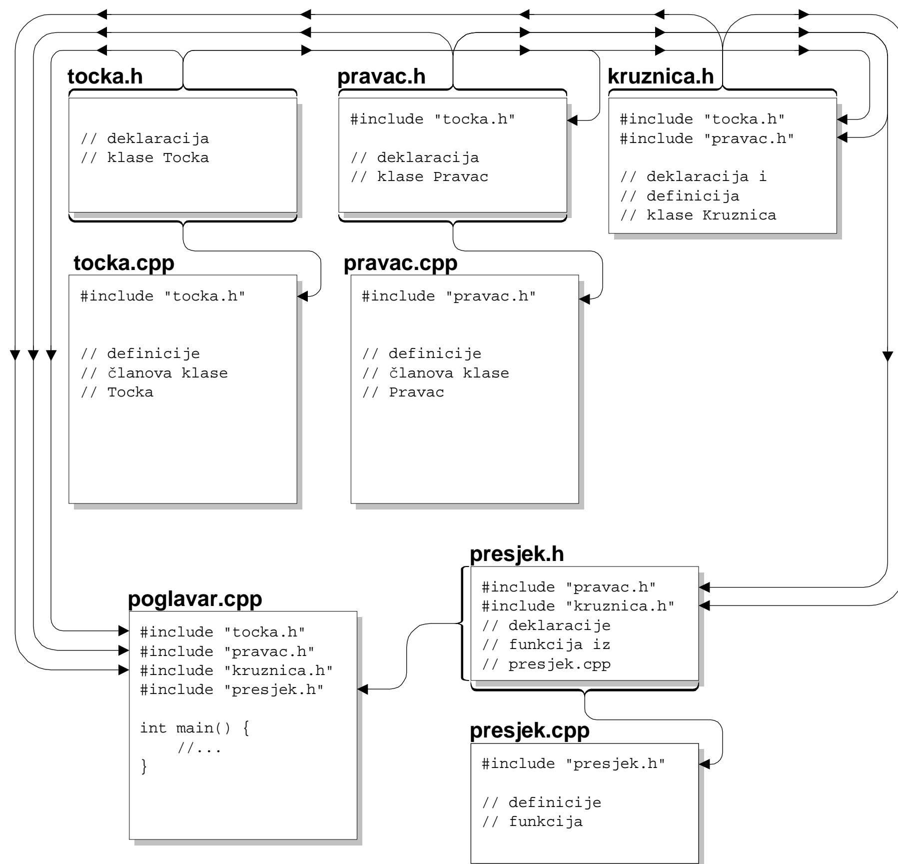
```
tocka.h
```

```
#ifndef TOCKA_H     // smisao ovoga će biti objašnjen
#define TOCKA_H     // iza cjelokupnog kôda

class Tocka {
public:
    Tocka(double x = 0, double y = 0);
    double x() const { return x_koord; }
    double y() const { return y_koord; }
```

```
    void pomakni(double x, double y);
protected:
    double x_koord;
    double y_koord;
};

#endif
```

tocka.cpp

```
#include "tocka.h"

Tocka::Tocka(double x, double y) {
    x_koord = x;
    y_koord = y;
}


void Tocka::pomakni(double x, double y) {
    x_koord = x;
    y_koord = y;
}
```



Slika 17.1. Struktura uključivanja u primjeru složenog programa

Uoèimo kako su podrazumijevani argumenti konstruktora navedeni samo u deklaraciji klase, ali ne i u definiciji konstruktora.

Klasu `Pravac` ćemo deklarirati (zajedno s definicijama umetnutih funkcijskih članova) u datoteci `pravac.h`, a njene članove ćemo definirati u datoteci `pravac.cpp`. Konstruktor klase `Pravac` kao argumente prihvaća točku i prirast u smjeru osi x, odnosno osi y. Stoga je u deklaraciji klase neophodno uključiti i deklaraciju klase `Tocka` iz datoteke `pravac.h`.

`pravac.h`

```
#ifndef PRAVAC_H
#define PRAVAC_H
```

```
#include "tocka.h"

class Pravac {
public:
    Pravac(Tocka t, double dx, double dy);
    double ka() const { return a; }
    double kb() const { return b; }
    double kc() const { return c; }
private:
    double a;
    double b;
    double c;
};

#endif
```

```
pravac.cpp
```

```
#include "pravac.h"

Pravac::Pravac(Tocka t, double dx, double dy) {
    a = dy;
    b = -dx;
    c = dx * t.y() - dy * t.x();
}
```

Klasa `Kruznica` opisuje kružnicu pomoæu središta kružnice i njenog polumjera. Klasa sadrži konstruktor, koji inicijalizira središte i polumjer, te èlanove za dohvaæanje koordinata središta i polumjera. Kako je klasa `Kruznica` jednostavna te su se svi èlanovi mogli ostvariti umetnutima, cijela je smještena u datoteku `kruznica.h` – uopæe nema potrebe za posebnom datotekom definicije.

```
kruznica.h
```

```
#ifndef KRUZNICA_H
#define KRUZNICA_H

#include "tocka.h"
#include "pravac.h"

class Kruznica {
public:
    Kruznica(Tocka s, double r) : srediste(s), polumjer(r) {}
    double r() const { return polumjer; }
    double x0() const { return srediste.x(); }
    double y0() const { return srediste.y(); }
private:
    Tocka srediste;
```

```
    double polumjer;
};

#endif
```

Funkciju `PresjecistaKruzniceIPravca` za raèunanje presjecišta kružnice i pravca definirat æemo u datoteci `presjek.cpp`, a njenu æemo deklaraciju pohraniti u datoteku `presjek.h`:

presjek.h

```
#ifndef PRESJEK_H
#define PRESJEK_H

#include "pravac.h"
#include "kruznica.h"

enum VrstaPresjeka {NijednoPresjeciste,
                    PravacDiraKruznicu,
                    DvaPresjecista};

VrstaPresjeka PresjecistaKruzniceIPravca(const Kruznica &k,
                    const Pravac &p, Tocka &t1, Tocka &t2);

#endif
```

Pobrojenje `VrstaPresjeka` se koristi za definiciju povratnog tipa funkcije za raèunanje presjeka. Ta vrijednost oznaèava sijeèe li uopæe pravac kružnicu, da li ju samo (nježno) dira ili ju sijeèe u dvije toèke.

presjek.cpp

```
#include "presjek.h"
#include <math.h>

VrstaPresjeka PresjecistaKruzniceIPravca(const Kruznica &k,
                    const Pravac &p, Tocka &t1, Tocka &t2) {
    if (p.kb()) {       // ako pravac nije okomit
        double a_b = p.ka() / p.kb();
        double c_b = p.kc() / p.kb();
        double A = 1. + a_b * a_b;
        double B = 2. * (a_b * c_b + a_b * k.y0() - k.x0());
        double C = k.x0() *- k.x0() + k.y0() * k.y0() -
                   k.r() * k.r() + c_b * (c_b + 2 * k.y0());
        double diskr = B * B - 4 * A * C;
        if (diskr < 0)      // nema presjecista
          return NijednoPresjeciste;
        else if (diskr == 0) {      // pravac dira kruznicu
```

```
            double x1 = -B / 2. / A;
            double y1 = -a_b * x1 - c_b;
            t1.pomakni(x1, y1);     // obje tocke jednake
            t2.pomakni(x1, y1);
            return PravacDiraKruznicu;
        }
        double x1 = (-B + sqrt(diskr)) / 2. / A;
        double y1 = -a_b * x1 - c_b;
        t1.pomakni(x1, y1);
        x1 = (-B - sqrt(diskr)) / 2. / A;
        y1 = -a_b * x1 - c_b;
        t2.pomakni(x1, y1);
        return DvaPresjecista;
    }
    else {
        // kôd za vertikalni pravac ćete,
        // naravno, napisati sami!
    }
}
```

Konaèno evo i datoteke `poglavar.cpp`. Funkcija `main()` æe stvoriti kružnicu zadanu koordinatom središta i polumjerom i pravac zadan toèkom i prirastima u *x* i *y* smjeru. Nakon toga æe se pozvati funkciju `PozoviPaIspisi()` za izraèunavanje i ispis presjecišta pravca i kružnice:

poglavar.cpp

```
#include <iostream.h>
#include "tocka.h"
#include "pravac.h"
#include "kruznica.h"
#include "presjek.h"

static void PozoviPaIspisi(const Kruznica &k,
                           const Pravac &p) {
    Tocka t_pr1;        // točke za pohranjivanje
    Tocka t_pr2;        // koordinata presjecišta
    switch (PresjecistaKruzniceIPravca(k, p, t_pr1, t_pr2)) {
      case 0:
        cout << "Nema presjecista!" << endl;
        break;
      case 1:
        cout << "Pravac dira kruznicu u tocki T("
             << t_pr1.x() << "," << t_pr1.y() << ")";
        break;
      case 2:
        cout << "Pravac sijece kruznicu u tockama T1("
             << t_pr1.x() << "," << t_pr1.y() << ") i T2("
             << t_pr2.x() << "," << t_pr2.y() << ")" << endl;
        break;
```

```
    }
}

int main() {
    Tocka t0(1., 1.);        // postavljamo točku
    double r = 5.;           // polumjer kružnice
    Kruznica k(t0, r);       // definiramo kružnicu
    Pravac p(t0, 1., 1.);    // pravac kroz središte kružnice
                             // pod 45 stupnjeva
    PozoviPaIspisi(k, p);
    return 0;
}
```

Funkcija `PozoviPaIspisi()` je statièka, što znaèi da ona nije vidljiva izvan datoteke `poglavar.cpp`. To i ima smisla, jer je ta funkcija uvedena samo zato da se dio kôda koji poziva funkciju za raèunanje presjeka i ispisuje rezultate, izdvoji zasebno i na taj naèin istakne. Poželjno je da ime takve funkcije ne “zagaðuje” globalno podruèje imena.

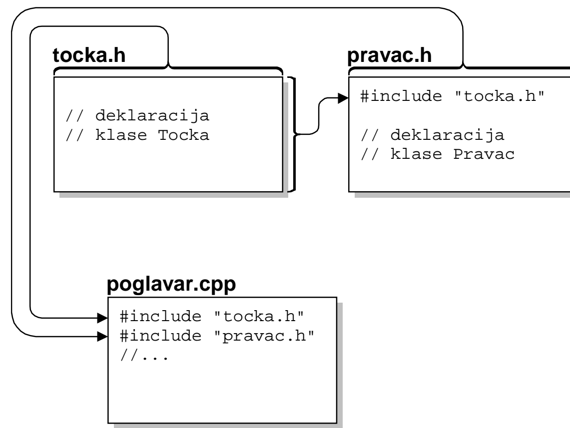
U svim datotekama zaglavlja pažljivi čitatelj je sigurno primijetio pretprocesorske naredbe `#ifndef-#define-#endif`. Na primjer, u datoteci `tocka.h` su sve deklaracije i definicije smještene unutar naredbi

```
#ifndef TOCKA_H
#define TOCKA_H
    // deklaracija i definicije
#endif
```

Pomoæu ovakvog ispitivanja izbjegava se moguæa višekratna deklaracija klase. Na primjer, u gornjem programu deklaracija klase `Tocka` ukljuèena je modul `poglavar.cpp`, jer se u glavnoj funkciji generira objekt te klase. Istovremeno, klasa `Tocka` mora biti poznata deklaraciji klase `Pravac`, jer se objekt klase `Tocka` prenosi kao argument njegovom konstruktoru. Kako je deklaracija klase `Pravac` također ukljuèena u glavni modul, a naredba `#include` obavlja jednostavno umetanje teksta, u glavnom modulu æe se deklaracija klase `Tocka` pojaviti dva puta: jednom izravno, ukljuèivanjem datoteke `tocka.h`, a drugi puta posredno preko ukljuèivanja datoteke `pravac.h` (slika 17.2). Na ovo æe prevoditelj prijaviti kao pogrešku, bez obzira što su one potpuno identiène). Slièna situacija je i kod klase `Kruznica`. Pažljivim odabirom rasporeda ukljuèivanja bi se ova pogreška mogla ukloniti, međutim to iziskuje preveliki intelektualni napor i moæ kombiniranja – daleko elegantnije rješenje jest uvjetno prevođenje pomoæu gornjeg sklopa naredbi.

Kako funkcionira navedeni sklop? Pri prvom nailasku na pretprocesorsku naredbu za uključivanju datoteke `tocka.h`, makro ime `TOCKA_H` nije definiran, pa se prevodi kôd između naredbi `#ifndef` i `#endif`. Unutar bloka `#ifndef-#endif` nalazi se naredba

```
#define TOCKA_H
```



Slika 17.2. Dvokratna deklaracija klase **`Tocka`** u glavnom modulu (izdvojeni detalj sa slike 17.1)

kojom se makro ime `TOCKA_H` definira (na neku neodreðenu vrijednost). Zbog toga æe kod sljedeæeg ukljuèivanja datoteke `tocka.h` (na primjer, prilikom ukljuèivanja datoteke `pravac.h`) makro ime `TOCKA_H` biti definirano pa æe se ponovno prevoðenje deklaracije klase `Tocka`.

Makro ime može biti proizvoljno, ali je uobièajeno da se uskladi s imenom datoteke, osim što se toèka ispred nastavka `.h` ili `.hpp` zamijeni znakom podcrtavanja.

Time se znaèajno smanjuje moguænost da se neko ime upotrebi u razlièitim datotekama zaglavlja.

Postupak izlučivanja datoteka zaglavlja često zna za početnika biti mukotrpan posao. Standard jezika ne specificira niti jedno pravilo glede organizacije kôda koje bi prisiljavalo programera da ga se drži. Stoga taj postupak iziskuje dosta discipline i sva odgovornost leži na programeru; za one “manje marljive”, u [Horstmann96] dan je kôd programa za automatsko generiranje datoteka zaglavlja, *pa ‘ko voli, nek izvoli*.

17.6. Povezivanje s kôdom drugih programskih jezika

Jezik C++ jest dobar, ali ne toliko dobar da bi ga trebalo koristiti pod svaku cijenu za rješavanje svih problema. Na primjer, on ne podržava vektor-operacije na strojevima koji imaju za to posebno optimizirano sklopovlje, no FORTRAN to ima. Zbog toga je prirodno rješenje napisati dio programa koji koristi takve operacije u FORTRAN-u, a zatim to povezati s C++ kosturom koji primjerice, ostvaruje prikladno Windows suèelje.

Također, mnogi programeri posjeduju već znatne megabajte izvornog C kôda. Naposljetku taj je jezik još uvijek (ali ne zadugo, *hehehe...*) najrasprostranjeniji jezik

(osim naravno kineskog, kojeg govori trećina čovječanstva). Kako je jezik C++ nadogradnja jezika C (*C++: As close as possible to C – but no closer*[†]), prevođenje izvornog kôda pisanog u C-u na C++ prevoditelju ne bi trebala predstavljati veći problem. Primijetite naglasak na *ne bi trebala*: stopostotna kompatibilnost s C-om nije ostvarena, te su oko toga bile vođene žučne polemike. Na kraju, utvrđeno je da se ipak dio kompatibilnosti mora žrtvovati kako bi se ostvarile nove mogućnosti.

Dio nekompatibilnosti je uzrokovan time što C++ ima nove ključne riječi, čiji se nazivi ne smiju koristiti za nazive identifikatora. Ako C program deklarira, primjerice, cjelobrojnu varijablu naziva `class`, takav program se neće moći prevesti C++ prevoditeljem. Nadalje, područja pojedinih imena se neznatno razlikuju u jezicima C i C++. Na primjer, u jeziku C se ugniježđene strukture mogu koristiti u globalnom području bez navođenja imena područja u koje je struktura ugniježđena. Zbog takvih i sličnih primjera, ponekad može biti jednostavnije ne prevoditi C kôd C++ prevoditeljem, nego jednostavno C kôd prevesti C prevoditeljem i povezati ga s C++ kôdom. Poteškoće također nastaju kada izvorni kôd u C-u nije dostupan, pa ga se ne može ponovo prevesti C++ prevoditeljem.

Povezivanje objektnih kôdova pisanih u različitim jezicima nije trivijalno. Naime, objektni kôdovi dobiveni različitim prevoditeljima, pa čak i objektni kôdovi dobiveni prevođenjem istog jezika, ali na prevoditeljima raznih proizvođača, međusobno nisu kompatibilni. Primjerice, redoslijed prenošenja argumenata funkciji ili struktura pohranjivanja pojedinih tipova podataka se razlikuju za C i Pascal. Standard jezika C++ podržava povezivanje samo s jezikom C, ali prevoditelji pojedinih proizvođača često podržavaju povezivanje s drugim jezicima – za to morate pogledati upute za prevoditelj/povezivač koji koristite.

17.6.1. Poziv C funkcija iz C++ kôda

Standardom je definiran naèin povezivanja C++ i C kôda. Jednostavna deklaracija C-funkcije specifikatorom `extern` neæe puno pomoæi, jer se njome samo naznaèuje da je funkcija definirana izvan datoteke izvornog kôda, ali se podrazumijeva da ta funkcija odgovara specifikacijama jezika C++. Funkcije u jeziku C++ razlikuju se od C-funkcija po mnogim svojstvima, a jedno između njih je preoptereæenje imena funkcija. Prilikom prevođenja kôda, C++ prevoditelj æe pohraniti imena funkcija u objektni kôd prema posebnim pravilima, da bi mogao pratiti njihovo preoptereæenje. Taj postupak se naziva *kvarenje imena* (engl. *name mangling*). Da bi povezivaè mogao razlikovati razlièite verzije funkcija za razlièite parametre, prevoditelj æe u objektni kôd na naziv funkcije nalijepiti još dodatne znakove kojima æe oznaèiti parametre koje ta funkcije uzima. Naèin na koji to pojedini prevoditelj radi se razlikuje ovisno o proizvoðaèu i nije definiran standardom.

Želi li se u C++ kôd uključiti C-funkciju, iza ključne riječi `extern` treba navesti `"C"` tip povezivanja, čime se prevoditelju daje na znanje da za tu funkciju treba obustaviti kvarenje imena:

[†] “C++: Blizu jeziku C koliko je god moguæe, ali ništa bliže” – naslov èlanka A. Koeniga i B. Stroustrupa u èasopisu *C++ Report.*

```
extern "C" void NekaCFunkcija(char *poslanica);
```

Moguæe je ukljuèiti i nekoliko funkcija istovremeno, tako da se funkcije iza `extern "C"` navedu u vitièastim zagradama:

```
extern "C" {
    float funkcijaCSimplex(int);
    int rand();                     // standardne C-funkcije
    void srand(unsigned int seed);  // za generiranje
                                    // slučajnih brojeva
}
```

Štoviše, sljedeæim ukljuèivanjem mogu se C datoteke zaglavlja pretvoriti u C++ datoteke zaglavlja:

```
extern "C" {
    #include "mojeCfun.h"
}
```

Mnogi C++ prevoditelji se isporuèuju sa standardnim bibliotekama koje u datotekama zaglavlja C biblioteka imaju ukljuèenu ovakvu naredbu koja omoguæava vrlo jednostavan poziv tih funkcija u C++ programima. Pri tome se koristi moguænost uvjetnog prevođenja – pretprocesorski simbol `__cplusplus` je definiran ako se provodi prevođenje u C++ modu, u kojem se sluèaju tada prevodi i `extern "C"` naredba koja deklarira sve funkcije iz biblioteke sa C povezivanjem.

```
#ifdef __cplusplus
extern "C" {
#endif

// slijede deklaracije...

#ifdef _ _cplusplus
}
#endif
```

Valja primijetiti da direktiva `extern "C"` specificira samo konvenciju za povezivanje, ali ne utjeèe na semantiku poziva funkcije. To znaèi da se za funkciju deklariranu kao `extern "C"` i dalje primjenjuju pravila provjere i konverzije tipa kao da se radi o C++ funkciji. Ta su pravila stroža nego pravila za jezik C.

Budući da su tipovi podataka ugrađeni u jezik C++ potpuno jednaki tipovima u jeziku C, argumenti C funkcijama se mogu prenositi izravno, bez pretvaranja zapisa podataka u drugačiji format. S drugim jezicima to nije slučaj: na primjer, u Pascalu se znakovni nizovi pamte tako da se u prvi bajt niza upiše njegova duljina iza koje slijede sami znakovi. Ako se znakovni niz šalje iz C++ programa u Pascal program, potrebno je pretvoriti niz u Pascal format.

Napomenimo još da je moguće eksplicitno deklarirati C++ povezivanje pomoću riječi `extern "C++"`, no kako je to podrazumijevano povezivanje, takva deklaracija se gotovo nikada ne koristi.

17.6.2. Ukljuèivanje asemblerskog kôda

Asemblerski jezik je najniži programski jezik koji omoguæava izravno rukovanje memorijskim lokacijama. On je praktièki jednak strojnim instrukcijama, osim što svaka strojna instrukcija ima svoj mnemonik u obliku kratice na engleskom jeziku, na primjer: `mov` – *move*, `asr` – *arithmetic shift right*, `psh` – *push*. Kako smo vidjeli, jezik C++ pruža neke operacije koje su vrlo bliske ili identiène asemblerskim naredbama, kao što su bitovni operatori `|`, `&`, `<<`. Pa ipak, ponekad niti to nije dovoljno – postoje aplikacije u kojima je vrijeme izvođenja dijela kôda od presudnog znaèaja. Tipièan primjer za to bi mogao biti program koji prima podatke s nekog vanjskog uređaja, te ih odmah mora obraditi. Program sam po sebi ne mora biti brz, ali kada podatak naiđe, obradu tog podatka treba provesti u što kraæem moguæem roku kako bi sistem bio spreman prihvatiti sljedeæi podatak koji æe doæi s vanjskog uređaja. Želimo li dobiti najveæu moguæu brzinu obrade, morat æemo taj odsjeèak napisati u asembleru.

ANSI/ISO standard omogućava jednostavno izravno uključivanje asemblerskog kôda u izvorni kôd pisan u jeziku C++ pomoću ključne riječi `asm`. Sintaksa je oblika

```
asm ( znakovni_niz );
```

pri èemu je *`znakovni_niz`* asemblerska naredba omeđena navodnicima. Sintaksa `asm` naredbe za pojedine prevoditelje èesto odstupa od gornjeg oblika, tako da je neophodno konzultirati pripadajuæi priruènik.

Valja naglasiti da se asemblerske naredbe razlikuju za različite familije procesora: asembler za Intelov Pentium procesor će sadržavati drugačije naredbe nego asembler za Motoroline procesore. Štoviše, čak i procesori iste familije (na primjer Intel 8086, 80286 ili Pentium) ne moraju imati iste asemblerske naredbe, jer se razvojem procesora povećava skup podržanih instrukcija. Ipak, obično postoji kompatibilnost unatrag, što znači da će asemblerski kôd pisan za i8086 “vrtiti” na svim novijim Intelovim procesorima.

Korištenje asemblerskog jezika iziskuje dosta iskustva i opreza, budući da on omogućava izravno dohvaćanje memorijskih lokacija što može dovesti do prekida rada računala. Stoga početniku ne preporučujemo riskantna “zaletavanja” u to područje. Zbog navedenih razloga ovdje neće biti dan niti jedan konkretan primjer asemblerskog kôda. Želite li se ipak upustiti u povezivanje C++ i asemblerskog kôda, najzgodnije je potražiti gotove primjere u specijaliziranim knjigama ili časopisima. Također, kao izvor asemblerskog kôda na kojemu se možete učiti mogu poslužiti programi za simboličko pronalaženje pogrešaka koji omogućavaju prikaz izvedbenog kôda u obliku asemblerskih mnemonika. Pregledom takvog kôda, iskusni programer će uočiti redundantne operacije umetnute tijekom prevođenja/povezivanja te će ih znati “skresati” u vremenski kritičnim dijelovima kôda.

18. Ulazni i izlazni tokovi

Teče i teče, teče jedan slap;
što u njem znači moja mala kap?

Dobriša Cesarić: "Slap"

Svaki program ima smisla samo ako može komunicirati s vanjskim svijetom – nas ne zanimaju programi koji hiberniraju sami za sebe (poput *Mog prvog C++ programa*). U najjednostavnijem sluèaju želimo da program koji pokreæemo barem ispiše neku poruku na zaslonu raèunala ("Dobar dan, gazda. Drago mi je da Vas opet vidim. Kako Vaši bubrežni kamenèiæi?").

Ulazni i izlazni tokovi (engl. *input and output streams*) osiguravaju vezu između našeg programa i vanjskih jedinica na računalu: tipkovnice, zaslona, disketne jedinice, diska, CD-jedinice. U ovom poglavlju upoznat ćemo s osnovama tokova: upoznat ćemo se s upisom podataka preko tipkovnice, ispisom na zaslonu računala te s čitanjem i pisanjem datoteke.

18.1. Što su tokovi

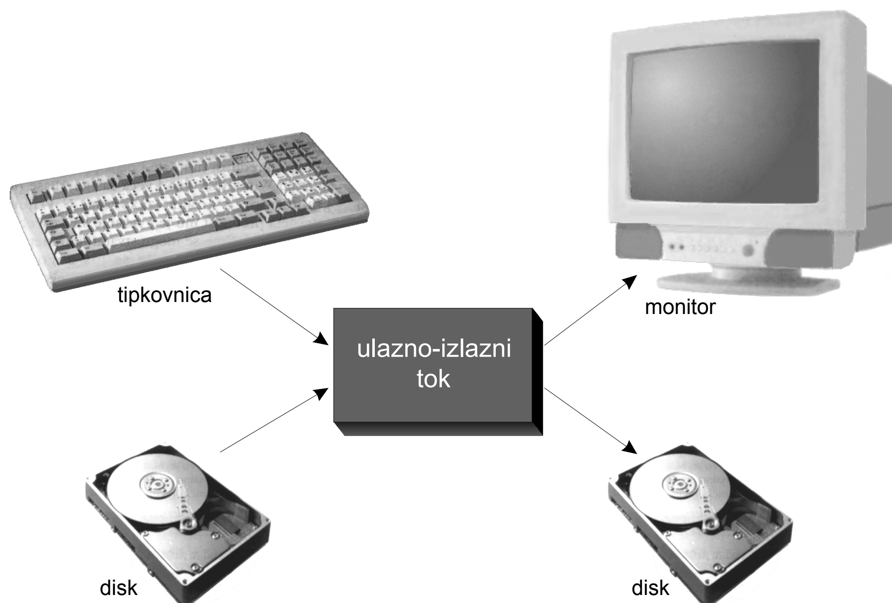
U mnogim dosadašnjim primjerima smo u kôd ukljuèivali `iostream.h` datoteku zaglavlja. U njoj su deklarirani ulazni i izlazni tokovi koji su nam omoguæavali upis podataka s tipkovnice ili ispis na zaslonu raèunala. Međutim, do sada nismo ulazili u suštinu ulazno-izlaznih tokova, buduæi da nam njihovo poznavanje nije bilo neophodno za razumijevanje jezika.

Sam programski jezik C++ ne definira naredbe za upis i ispis podataka. Naime, ulazno-izlazne naredbe u pojedinim jezicima podržavaju samo ugrađene tipove podataka. Međutim, svaki složeniji C++ program sadrži i složenije, korisnički definirane podatke, koji iziskuju njima svojstvenu obradu. Da bi se osigurala fleksibilnost jezika, izbjegnuta je ugradnja ulazno-izlaznih naredbi u jezik.

Za ostvarenje komunikacije programa s okolinom, programer mora na raspolaganju imati rutine koje će podatke svakog pojedinog tipa pretvoriti u računalu u nizove bitova te ih poslati na vanjsku jedinicu, odnosno obrnuto, nizove bitova s vanjske jedinice pretvoriti u podatke u računalu. U jeziku C++ to se ostvaruje pomoću ulaznih i izlaznih tokova. Tokovi su zapravo klase definirane u standardnim bibliotekama koje se isporučuju s prevoditeljem. Sadržaj tih klasa (svi funkcijski i podatkovni članovi) je standardiziran i omogućava gotovo sve osnovne ulazno-izlazne operacije. Osim toga, zahvaljujući svojstvima jezika C++, taj se set operacija po potrebi može proširivati.

Sam koncept tokova se nadovezuje na princip enkapsulacije – način ispisa podataka je neovisan o tipu jedinice. Tako će se pojedini podatak na gotovo isti način ispisati na

zaslon ili na disk (slika 18.1). Također, unos podataka je praktički istovjetan radi li se o unosu s tipkovnice ili diska. Nakon što se stvori određeni tok, program će komunicirati preko tokova, njima ostavljajući glavninu “prljavog” posla.



Slika 18.1. Enkapsulacija ulazno-izlaznog toka

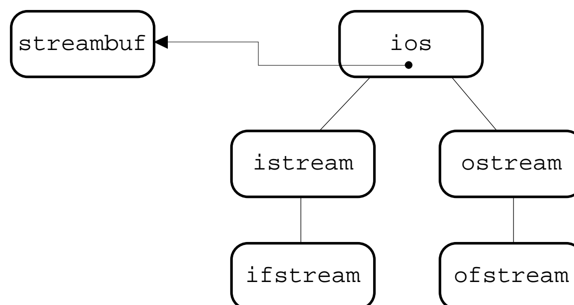
Uz pojam ulazno-izlaznih tokova usko je povezano i *međupohranjivanje* (engl. *buffering*). Naime, na neke vanjske jedinice podaci se ne zapisuju pojedinaèno, bajt po bajt, veæ u blokovima. Tako se podaci koji se šalju na disk prvo pohranjuju u *međuspremnik* (engl. *buffer*). Kada se međuspremnik napuni, cijeli se sadržaj međuspremnika pošalje disku koji ga snimi u odgovarajuæi sektor na disku. Međuspremnik se pri tome isprazni i spreman je za prihvat novog niza podataka. Kod uèitavanja podataka s diska, prvo se cijeli sektor s podacima uèita u međuspremnik, a potom se pojedini podaci èitaju iz međuspremnika.

Postupak pražnjenja međuspremnika je automatiziran, tako da o njemu programer ne mora voditi računa. Međutim, prilikom ispisa s međupohranjivanjem redovito posljednji podaci koji se ispisuju ne popune međuspremnik – bez naknadne intervencije programa oni bi ostali u međuspremniku, čekajući priliku da se međuspremnik eventualno popuni te da i oni budu prebačeni na vanjsku jedinicu. Da bi se osigurao zapis i tih podataka, neophodno je *isprazniti* (engl. *flush*) međuspremnik. Iako se međuspremnik prazni automatski prilikom zatvaranja toka, zbog sigurnosti je ponekad poželjno obaviti ga ručno.

18.2. Biblioteka `iostream`

`iostream` biblioteka sadrži dvije usporedne obitelji klasa (slika **18.2**):

1. Osnovnu klasu `streambuf` i iz nje izvedene klase, kojima je zadatak osigurati međupohranu podataka pri uèitavanju i ispisu podataka.
2. Osnovnu klasu `ios` i iz nje izvedene klase, koje se koriste za formatirano uèitavanje i ispis podataka. Klasa `ios` sadrži i pokazivaè na `streambuf` klasu, što omoguæava međupohranu podataka za objekte klase `ios`, odnosno bilo koje iz nje izvedene klase.



Slika 18.2. Pojednostavljeni prikaz stabla nasljeđivanja klasa u `iostream` biblioteci

Iz klase `ios` (poèetna slova od *Input-Output Stream*) izravno su izvedene klase `istream` i `ostream`. Klasa `istream` se koristi kao osnovna klasa za ulazne tokove, dok se klasa `ostream` koristi kao osnovna klasa za izlazne tokove. One se mogu proširiti operacijama za ispis i uèitavanje korisnièki definiranih tipova. Biblioteka `iostream` takoðer podržava operacije za ispis u datoteku i èitanje iz datoteke. Klasa `ofstream`, izvedena iz klase `ostream`, podržava ispis ugraðenih tipova podataka. Klasa `ifstream`, izvedena iz klase `istream`, podržava uèitavanje ugraðenih tipova podataka iz datoteke. Klasa `fstream` podržava i èitanje i pisanje istodobno. Kako se sve klase za ulazne i izlazne tokove se izvode iz klasa u `iostream` biblioteci, sve one imaju barem jednu od klasa `ios` ili `streambuf` kao osnovnu. Sve ove klase obraðuju podatke tipa `char`, ali imaju svoje pandane za znakove tipa `wchar_t` u klasama `wios`, `wistream`, `wostream`, `wifstream`, `wofstream` i `wstreambuf`.

Valja napomenuti da je stablo nasljeđivanja u `iostream` biblioteci na slici **Error! Reference source not found.** prikazano bitno pojednostavnjeno – tamo su definirane i druge klase, a veza među njima je bitno složenija jer neke od tih klasa višestruko nasljeđuju razne osnovne klase.

Pokretanjem svakog C++ programa u koji je uključena `iostream.h` datoteka zaglavlja, automatski se stvaraju i inicijaliziraju četiri objekta:

- `cin` koji obraðuje unos sa standardne ulazne jedinice – tipkovnice,

- `cout` koji obraðuje ispis na standardnu izlaznu jedinicu – zaslon,
- `cerr` koji obraðuje ispis na standardnu jedinicu za ispis pogrešaka – zaslon,
- `clog` koji osigurava ispis pogrešaka s meðupohranjivanjem. Radi se o istim porukama kao i u `cerr` toku, ali se ovaj ispis obièno preusmjerava u datoteku.

Za podatke tipa `wchar_t` na raspolaganju su odgovarajuæi objekti `win`, `wout`, `werr` i `wlog` – za njih vrijede sva razmatranja koja æemo navesti za objekte `cin`, `cout`, `cerr`, odnosno `clog`.

18.3. Stanje toka

Svaki tok, bilo da se radi o izlaznom ili ulaznom toku, ima pripadajuæe *stanje* (engl. *state*). Pogreške nastale tijekom izluèivanja sa toka ili umetanja podataka na tok mogu se detektirati i prepoznati ispitujuæi stanje toka. Sva stanja su obuhvaæena pobrojenjem definiranim u klasi `ios`, a imaju nazive `goodbit`, `badbit`, `failbit` i `eofbit`. Stanje toka je pohranjeno u bitovnu masku `iostate` u klasi `ios` kao kombinacija gornjih vrijednosti. `iostate` se može oèitati pomoæu funkcijskog èlana `rdstate()`. Meðutim, radi lakše detekcije stanja definirana su èetiri funkcijska èlana (tipa `bool`) za èitanje stanja pojedinih bitova:

1. `eof()` – vraæa `true`, ako je s toka izluèen znak za kraj datoteke.
2. `bad()` – vraæa `true`, ako operacija nije uspjela zbog neke nepopravljive pogreške (npr. ošteæene datoteke).
3. `fail()` – vraæa `true`, ako operacija nije bila uspješno obavljena zbog bilo kojeg razloga.
4. `good()` – vraæa `true` ako niti jedan od gornjih uvjeta nije `true`, tj. ako je operacija potpuno uspješno izvedena.

Razlika između `bad()` i `fail()` (odnosno `badbit` i `failbit`) je suptilna i za veæinu primjena nije neophodno razlikovati ta dva èlana.

Također je definiran i funkcijski član `clear()` koji postavlja stanje toka. Podrazumijevana vrijednost argumenta tog funkcijskog člana je stanje `ios::goodbit`, pa ako se ne navede neki drugi argument, `clear()` će izbrisati stanje pogreške u toku.

Klasa `ios` posjeduje operator konverzije koji konvertira objekt klase `ios` u tip `void *`. Taj operator vraća nul-pokazivač ako je u stanju toka postavljen `failbit`, `badbit` ili `eofbit`. Inače, vraća se neki pokazivač koji nije nul-pokazivač (vrijednost tog pokazivača ne može koristiti ni za što drugo osim za testiranje). To omogućava testiranje toka: ako se tok nađe u uvjetu `if` ili `while` naredbe, pozvat će se taj operator konverzije, te će uvjet biti ispunjen ako je tok u stanju `goodbit`. Na primjer:

```
if (cout)
    cout << "Tečem dalje" << endl;
else
    cout << "Rijeka je presušila, tok više ne teče." << endl;
```

Klasa `ios` definira i operator `!`. On vraæa vrijednost razlièitu od nule ako je tok u stanju pogreške (`failbit`, `badbit`, `eofbit`), odnosno nula ako je tok u ispravnom stanju (`goodbit`). Tako je test toka moguæe i ovako obaviti:

```
if (!cin)
    cerr << "Pukla je brana, tok curi.";
```

Moguæe je dohvatiti stanje bilo kojeg izlaznog ili ulaznog toka (pa tako i za unos s tipkovnice ili ispis na zaslon). Kod ulaznog toka stanje se najèešæe koristi kako bi se detektiralo da je korisnik upisao podatak koji tipom ne odgovara oèekivanom (na primjer, oèekuje se cijeli broj, a korisnik unese znak). Kod tokova vezanih na datoteke stanja su još korisnija, jer se njima može ustanoviti je li pojedina ulazno-izlazna operacija uspjela. Stanje `eofbit` se èesto ispituje prilikom èitanja toka – ono signalizira da se stiglo do kraja datoteke. Stanja `badbit`, odnosno `failbit` se obièno testira prilikom upisa kako bi se provjerilo je li upis uspješno proveden.

18.4. Ispis pomoæu `cout`

`cout` je globalni objekt klase `ostream` koji se automatski stvara izvoðenjem programa u èiji je izvorni kôd ukljuèena datoteka `iostream.h`. On usmjerava ispis podataka na zaslon raèunala pomoæu operatora `<<`. Taj operator je preoptereæen za sve ugraðene tipove podataka, a korisnik ga može po potrebi preopteretiti za korisnièki definirane tipove. Osim toga, moguæe je formatirati ispis podataka, poravnati stupce u ispisu, ispisati podatke u dekadskom ili heksadekadskom obliku, skuhati kavu...

18.4.1. Operator umetanja <<

Ispis podataka ostvaruje se pomoæu izlaznog operatora `<<`. On se obièno zove *operatorom umetanja* (engl. *insertion operator*), jer se njime podatak umeæe na izlazni tok. U suštini se radi o operatoru za bitovni pomak ulijevo koji je preoptereæen tako da s lijeve strane kao operand prihvaæa referencu na objekt tipa `ostream`. S desne strane se može naæi bilo koji tip podatka: `char`, `short`, `int`, `long int`, `char *`, `float`, `double`, `long double`, `void *`. Tako su sljedeæe operacije standardno podržane:

```
cout << 9.3 << endl;                           // float
cout << 'C' << endl;                           // char
cout << 666 << endl;                           // int
cout << "Hvala bogu, još uvijek sam ateist";   // char * †
```

Za ispis pokazivaèa (izuzev `char *` za ispis znakovnih nizova) bit æe pozvan operator `ostream::operator<<(const void *)`, koji æe ispisati heksadekadsku adresu objekta u memoriji. Zato æe izvoðenje naredbi

† Reèenica koju je izrekao Luis Buñuel (1900–1983), “Le Monde”, 1959. godine.

```
int i;
cout << &i;
```

na zaslonu raèunala ispisati nešto poput `0x7f24`. Operator umetanja može se dodatno preopteretiti za korisnièki definirane tipove podataka, što æe biti opisano u sljedeæem odjeljku.

Kao rezultat, operator `<<` vraća referencu na objekt tipa `ostream`, a kako se operator izvodi s lijeva na desno, moguće je ulančavanje više operatora umetanja:

```
cout << 1001 << " noć";
```

Ova naredba æe biti interpretirana kao:

```
(cout.operator<<(1001)).operator<<(" noć");
```

Pritom æe podaci biti ispisani redoslijedom kako su navedeni, s lijeva na desno.

Operator `<<` kotira dovoljno nisko u hijerarhiji operatora, tako da aritmetičke izraze čije rezultate želimo ispisati najčešće nije neophodno pisati u zagradama:

```
cout << 5 + 4 << endl;          // ispisat će 9
cout << ++i << endl;            // ispisat će i uvećan za 1
```

Meðutim, pri ispisu izraza u kojima se koriste logièki, bitovni i operatori pridruživanja je potrebno paziti jer oni imaju niži prioritet. Na primjer, izvoðenje sljedeæih naredbi

```
int prvi = 5;
int drugi = 7;
cout << "Veći od brojeva " << prvi << " i " << drugi << " je "
// problematičan ispis:
cout << (prvi > drugi) ? prvi : drugi;
```

ispisat æe sljedeæi, na prvi pogled neoèekivani rezultat:

```
Veći od brojeva 5 i 7 je 0
```

Razlog ovakvom rezultatu shvatit æemo èim bacimo pogled na tablicu operatora u poglavlju 2 i uoèimo da uvjetni operator ima niži prioritet od operatora umetanja. To znaèi da æe naredbu prevoditelj interpretirati jednako kao da je pisalo:

```
(cout << (prvi > drugi)) ? prvi : drugi;
```

tj. ispisat æe rezultat usporedbe varijabli `prvi` i `drugi`, a tek potom æe pristupiti izvoðenju uvjetnog operatora `?:`. Kako se taj operator zapravo primjenjuje na objekt tipa `ostream &`, poziva se operator za konverziju u `void *`, te se zapravo testira tok.

> Ako se u naredbama za ispis navode izrazi, dobra je navika omeđiti ih zagradama.

Primijenjeno na gornji primjer to znaèi da smo naredbu za ispis trebali napisati kao:

```
cout << ((prvi > drugi) ? prvi : drugi);
```

Neki prevoditelji æe prilikom prevođenja dati upozorenje izostave li se zagrade.

Zadatak. *Pokušajte predvidjeti što će se ispisati na zaslonu izvođenjem sljedećih naredbi:*

```
int i = 7, j = 0;
cout << i * j;
cout << (i && j);
cout << i && j;
```

18.4.2. Ispis korisnièki definiranih tipova

Operator umetanja se može preoptereæivati za korisnièki definirane tipove podataka. Pritom treba kao prvi argument operatorske funkcije navesti referencu na objekt tipa `ostream`, a kao drugi argument navodi se željeni tip. Povratni tip mora biti referenca na objekt tipa `ostream` kako bi se omoguæilo ulanèavanje operacija ispisivanja.

Preopterećenje operatora `<<` ilustrirat ćemo primjerom ispisa kompleksnih brojeva. Pretpostavimo da smo za kompleksne brojeve deklarirali klasu `Kompleksni`:

```
class Kompleksni {
private:
    double realni, imaginarni;
public:
    Kompleksni(double r = 0, double i = 0) {
        realni = r;
        imaginarni = i;
    }
    double Re() {return realni;}
    double Im() {return imaginarni;}
    //...
};
```

Operatorsku funkciju za ispis kompleksnog broja možemo definirati na sljedeæi naèin:

```
ostream& operator<<(ostream &os, Kompleksni &z) {
    os << z.Re();
    if (z.Im() >= 0) os << "+";
```

```
    return os << z.Im() << "i";
}
```

U ispravnost funkcije uvjerit æemo se nakon izvoðenja sljedeæih naredbi:

```
Kompleksni z(2., -8.);
cout << z << endl;
```

Na zaslonu æe se ispisati naš kompleksni broj:

```
2-8i
```

Zadatak. *Deklarirajte klasu* `Tocka` *koja će imati dva privatna podatkovna člana za pohranjivanje koordinate točke u pravokutnim koordinatama, te javne funkcijske članove* `x( )` *i* `y( )` *za dohvaćanje tih članova. Preopteretite operator umetanja* << *tako da ispisuje koordinate točke u obliku:* `(x, y)`.

Zadatak. *Program iz prethodnog zadatka proširite deklaracijom klase* `Krug` *koja će imati dva privatna podatkovna člana: objekt* `srediste` *klase* `Tocka`*, te* `polumjer` *(npr. tipa* `double`*). Preopteretite operator umetanja tako da parametre kruga ispisuje u sljedećem obliku:* `(r = polumjer, t0(x0, y0))`*. Iskoristite pritom već preopterećeni operator za ispis koordinata točke iz prethodnog zadatka.*

18.4.3. Ostali èlanovi klase `ostream`

Ispis pojedinaènih znakova moguæ je i pomoæu funkcijskog èlana `put()`. On kao argument prihvaæa znak tipa `char`, a vraæa referencu na objekt tipa `ostream`. Meðutim, kako operator `<<` sasvim zadovoljava potrebe ispisa znakova, `put()` se rijetko koristi.

Slična je situacija sa funkcijskim članom `write()` namijenjenog za ispis znakovnih nizova. On kao prvi argument prihvaća znakovni niz `char *`, a drugi argument je broj znakova koje treba ispisati. Kao rezultat, i ovaj član vraća referencu na pripadajući objekt tipa `ostream`.

Funkcijski član `flush()` predviđen je za pražnjenje međuspremnika. Njegovim pozivom se svi podaci koji se trenutno nalaze u međuspremniku šalju na vanjsku jedinicu. Može se pozvati standardno, kao svaki funkcijski član:

```
cout.flush();
```

ili preko manipulatora, koji se pomoæu operatora umetanja ubacuje u objekt `cout`:

```
cout << flush;
```

Mehanizam manipulatora je detaljno objašnjen u odsjeèku 18.6.6.

Klasa `ostream` sadrži još dva manipulatora koja se vrlo često koriste: `endl` i `ends`. `endl` smo već dosad koristili u našim primjerima; on umeće znak za novi redak (`'\n'`) u tok i prazni tok funkcijskim članom `flush()`. Manipulator `ends` umeće zaključni nul-znak.

U klasi `ostream` definirana su i dva funkcijska člana koji služe za kontrolu položaja unutar toka. Funkcijski član `tellp()` vraća trenutačnu poziciju u toku, a `seekp()` se postavlja na zadanu poziciju. Ti članovi se koriste primarno za kontrolu pristupa datotekama, jer omogućavaju proizvoljno pomicanje unutar datoteke, pa će biti objašnjeni kasnije u ovom poglavlju, u odsječcima posvećenim radu s datotekama.

18.5. Uèitavanje pomoæu `cin`

Uèitavanje podataka je slièno ispisu podataka. Pokretanjem programa u koji je ukljuèena datoteka zaglavlja `iostream.h` automatski se stvara globalni objekt `cin` klase `istream`. Upis se obavlja pomoæu operatora >>. U klasi `istream` definirana je preoptereæena inaèica operatora za sve ugraðene tipove podataka, a dodatnim preoptereæenjem može se ostvariti uèitavanje korisnièki definiranih tipova.

18.5.1. Uèitavanje pomoæu operatora >>

Uèitavanje podataka ostvaruje se pomoæu ulaznog operatora `>>`. On se obièno zove *operatorom izluèivanja* (engl. *extraction operator*), jer se njime podatak izluèuje s izlaznog toka. U suštini se radi o operatoru za bitovni pomak udesno koji je preoptereæen tako da s lijeve strane kao operand prihvaæa referencu na objekt tipa `istream`. S desne strane se može naæi bilo koji tip podatka za koji je ulazni tok definiran: `char`, `short`, `int`, `long int`, `char *`, `float`, `double`, `long double`. Tako su uèitavanja sljedeæim naredbama standardno podržana:

```
int i;
long l;
float f;
double d;

cin >> i;           // int
cin >> l;           // long int
cin >> f;           // float
cin >> d;           // double
```

Uspješnim uèitavanjem podatka, operator izluèivanja vraæa kao rezultat referencu na objekt tipa `istream`. Zbog toga se naredbe za uèitavanje mogu ulanèavati, pa smo gornja uèitavanja mogli napisati i u jednoj naredbi:

```
cin >> i >> l >> f >> d;
```

Uèitavanje se nastavlja od mjesta gdje je prethodno uèitavanje završilo. Pritom se praznine (znakovi za razmak, tabulator, novi redak, pomak papira) standardno ignoriraju i one služe za odvajanje pojedinih podataka. To znaèi da smo sve ulazne podatke za gornji primjer mogli napisati u jednom retku, razmaknute prazninama:

```
24  102345  34.56   3.4567e95
```

a tek potom pritisnuti tipku za unos (*Enter*).

Ako učitavanje ne uspije, operator će tok postaviti u stanje `failbit`. Takva situacija će nastupiti ako se umjesto očekivanog tipa u ulaznom toku nađe podatak nekog drugog tipa. To se iskorištava kod opetovanog učitavanje podataka u petlji kada broj podataka koje treba učitati nije unaprijed poznat. Donji primjer će učitavati cijele brojeve dok se ne unese znak, kada se tok postavlja u stanje `failbit`. Petlja tada završava:

```
int i;

while ((cin >> i) != 0)
    cout << "Upisan je broj " << i << endl;
cout << "Fajrunat!" << endl;
```

Valja primijetiti da æe izvoðenje petlje prekinuti i decimalna toèka u eventualno upisanom realnom broju. Na primjer, upišemo li sljedeæi niz brojeva:

```
12 27 34.2 5
```

izvoðenjem gornje petlje ispisat æe se:

```
Upisan je broj 12
Upisan je broj 27
Upisan je broj 34
Fajrunat!
```

Kao što vidimo, izvoðenje petlje je prekinuto na decimalnoj toèki. Buduæi da naredba za uèitavanje oèekuje samo cijele brojeve, svaki znak razlièit od neke decimalne znamenke program æe shvatiti kao graniènik. Stoga æe uèitati samo dio treæeg broja lijevo od decimalne toèke. U sljedeæem prolazu petlje pokušat æe uèitati decimalnu toèku, ali æe pokušaj njenog uèitavanja u cjelobrojnu varijablu završiti neuspjehom te æe se izvoðenje petlje prekinuti.

Zadatak. *Isprobajte što će se dogoditi ako u gornjem primjeru broj unesete u heksadekadskom formatu (npr. `0xff`).*

Naravno, da smo u gornjoj petlji uèitavali realne brojeve (`float` ili `double`), problema s decimalnom toèkom ne bi bilo.

18.5.2. Uèitavanje korisnièki definiranih tipova

Poput operatora umetanja, i operator izluèivanja `>>` može se preopteretiti za korisnièki definirane tipove podataka. Kao prvi argument operatorske funkcije se navodi referenca na objekt tipa `istream`, a kao drugi referenca na objekt koji se uèitava – referenca je nužna kako bi operator izluèivanja mogao promijeniti vrijednost objekta koji se uèitava. Povratni tip mora biti referenca na objekt tipa `istream` kako bi se omoguæilo ulanèavanje operacija uèitavanja.

Preopteretimo operator izlučivanja tako da se njime mogu učitavati kompleksni brojevi (klasu smo definirali u primjeru preopterećenja operatora umetanja, na str. 519):

```
istream& operator>>(istream &is, Kompleksni &z) {
    if (!is) return is;     // ako je tok već u stanju
                            // pogreške, operacija se prekida
    double r, i;        // realni i imaginarni dio
    char zn;            // za učitavanje predznaka
    int imPredznak = 1;     // predznak imaginarnog dijela
    is >> r >> zn;
    switch (zn) {
      case '-':
        imPredznak = -1;
      case '+':
        cin >> i >> zn;
        if (zn != 'i') {    // ako je prilikom unosa
                            // detektirana pogreška
                            // tok se postavlja u
                            // stanje pogreške:
            is.clear(ios::failbit)
            return is;      // i operacija se prekida
        }
        break;
    }
    z = Kompleksni(r, i * imPredznak);
    return is;
}
```

Pretpostavlja da se kompleksni broj upisuje toèno u zadanom obliku:

```
realni_dio + imaginarni_dio i
```

gdje *`realni_dio`* i *`imaginarni_dio`* mogu biti bilo kakvi realni brojevi. Praznine oko realnog i imaginarnog dijela su proizvoljne jer je ulazni tok postavljen tako da "jede" praznine oko uèitanih podataka.

Ovo je tek rudimentarna definicija operatorske funkcije, koja služi samo kao pokazni primjer. Funkcija obavlja samo osnovne provjere ispravnosti ulaznih podataka. Ako se detektira neispravnost ulaznih podataka (npr. umjesto `i`, za imaginarnu jedinicu se upiše neki drugi znak), stanje toka se postavlja u `failbit`, a operacija se prekida. Također, uočimo prvu naredbu u funkciji kojom se provjerava stanje toka prije

učitavanja. Ako je, zbog nepravilnih prethodnih operacija učitavanja, tok u stanju pogreške, operacija učitavanja se odmah prekida. Tako će se, u slučaju ulančanog učitavanja podataka s toka, učitavanje prekinuti čim se naiđe na prvu nepravilnost.

18.5.3. Uèitavanje znakovnih nizova

Buduæi da praznine predstavljaju graniènike koji prekidaju uèitavanje podatka, uèitavanje znakovnih nizova nije trivijalno. Sljedeæa petlja neæe tekst uèitati kao jedan znakovni niz, veæ æe svaka rijeè biti niz za sebe:

```
char text[80];

while (cin >> text) cout << text << endl;
```

Za upisani tekst

```
Svakim danom u svakom pogledu napredujemo.
```

gornja petlja æe ispisati niz rijeèi:

```
Svakim
danom
u
svakom
pogledu
napredujemo.
```

Želimo li uèitati znakovni niz zajedno sa prazninama, u klasi `istream` na raspolaganju su funkcijski èlanovi `get()`, `getline()`i `read()`. Funkcijski èlan `get()` definiran je u nekoliko preoptereæenih varijanti:

1. `get(char &znak)` izluèuje jedan znak s ulaznog toka te ga pohranjuje u varijablu `znak`. Funkcijski èlan kao rezultat vraæa referencu na objekt klase `istream`. Uèita li znak za kraj datoteke, tok æe se postaviti u stanje `eofbit`, koje možemo testirati operatorom konverzije. Želimo li uèitati sve znakove utipkanog niza koristeæi ovaj funkcijski èlan, možemo napisati sljedeæi kôd:

   ```
   char niz[80];
   int i = 0;

   while (cin.get(niz[i]))
       if (niz[i++] == '\n')
           break;
   niz[i - 1] = '\0';
   ```

   Nakon pokretanja programa, treba utipkati željeni niz – tek po pritisku na tipku za unos poèet æe se izvoditi petlja koja æe pojedine znakove uèitavati u `niz`. Valja

paziti da duljina upisanog teksta mora biti manja od duljine polja `niz`. Također, uoèimo da nakon završenog uèitavanja znakova, polju `niz` treba dodati zakljuèni nul-znak. Ovaj èlan ima daleko praktièniju primjenu kod uèitavanje podataka iz datoteke.

2. `get()` bez argumenata koji izluèuje znak s ulaznog toka. Èlan je tipa `int`, a vraæa kôd znaka koji je uèitao. Nailaskom na znak `EOF` funkcijski èlan æe vratiti vrijednost `-1`. Pomoæu ovog funkcijskog èlana petlja u prethodnom primjeru bi se mogla napisati na sljedeæi naèin:

```
while ((niz[i] = cin.get()) != -1)
    if (niz[i++] == '\n')
        break;
niz[i - 1] = '\0';
```

3. `get(char *znNiz, int duljina, char granicnik = '\n')` izluèuje niz znakova do najviše `duljina - 1` znakova. Rezultat se smješta na lokaciju koju pokazuje `znNiz`. Izluèivanje se prekida ako naleti na znak jednak argumentu `granicnik` ili ako uèita znak `EOF`. Na kraj uèitanog niza uvijek æe dodati zakljuèni nul-znak, a povratna vrijednost jednaka je pripadajuæem objektu `istream`, osim ako nije uèitan niti jedan znak. Èitatelj sigurno nasluæuje da je ovaj oblik funkcijskog èlana najpogodniji za uèitavanje utipkanog teksta. Riješimo zadatak uèitavanja utipkanog niza pomoæu ovog funkcijskog èlana:

```
char niz[80];

cin.get(niz, sizeof(niz));
```

Buduæi da je u deklaraciji funkcijskog èlana znak za novi redak podrazumijevani graniènik, ne trebamo ga navoditi eksplicitno kao treæi argument funkciji. Uz to, sam funkcijski èlan dodaje zakljuèni nul-znak na kraj uèitanog niza, tako da ga ne trebamo naknadno dodavati. Naravno, i dalje valja paziti na duljinu niza u koji se izluèuju znakovi – duljina niza se može utvrditi pomoæu operatora `sizeof`.

Funkcijski èlan `get(char *, int, char)` ostavlja znak `'\n'` na toku.

To znaèi da ako ulanèimo nekoliko operacija uèitavanja, drugi niz se neæe uspjeti uèitati jer æe sljedeæi `get()` odmah naiæi na `'\n'`. Isto tako æe sljedeæi kôd prouzroèiti beskonaènu petlju:

```
while (cin.get(char niz, sizeof(niz)))
    //...
```

Prvi i treæi oblik funkcijskog èlana `get()` vraæaju kao rezultat pripadajuæi objekt tipa `istream`, što omoguæava njihovo ulanèavanje, jednako kao i kod operatora `<<`, odnosno `>>`.

Funkcijski član `getline()` po potpisu je identičan trećem obliku funkcijskog člana `get()`; deklariran je kao

```
istream &getline(char *znNiz, int duljina,
                 char granicnik = '\n');
```

Stoga se taj funkcijski èlan poziva jednako kao i treæi oblik èlana `get()`, a i djeluje na slièan naèin, s jednom znaèajnom razlikom – `getline()` uklanja završni znak `'\n'` s ulaznog toka. Tako je moguæe ulanèavati operacije uèitavanja pomoæu `getline()`.

Član `getline()`, za razliku o `get()`, uklanja znak `'\n'` s ulaznog toka. Stoga se operacije učitavanja pomoću `getline()` mogu ulančavati, ali pomoću `get()` ne.

Funkcijski èlan `read()` takoðer uèitava niz znakova zadane duljine, ali se njegovo izvoðenje prekida samo u sluèaju nailaska na znak `EOF`. Deklariran je kao:

```
istream &read(char *znNiz, int duljina);
```

Njegova primjena uglavnom je ogranièena na uèitavanje podataka iz datoteka.

Funkcijski član `gcount()` vraća broj neformatiranih znakova učitanih funkcijskim članovima `get()`, `getline()` ili `read()`. Koristi za određivanje duljine učitanog niza ili za provjeru ispravnosti učitavanja.

18.5.4. Ostali èlanovi klase `istream`

Funkcijskim èlanom `ignore()` preskaèu se znakovi na ulaznom toku. On prihvaæa dva argumenta: prvi argument je broj znakova koliko ih treba preskoèiti, a drugi argument je znak kojim se preskakanje znakova može prekinuti. Podrazumijevana vrijednost tog znaka jest `EOF`. Pretpostavimo da prilikom unosa podataka želimo pohraniti samo prvih deset utipkanih znakova u svakom retku. U tom sluèaju æemo uporabiti funkcijski èlan `ignore()`:

```
char podaci[n][11];

for (int i = 0; i < n; i++) {
    cin.get(podaci[i], 11);
    cin.ignore(99, '\n');       // odbaci preostale znakove
}
```

Kada u gornjem kôdu ne bi bilo poziva funkcije `ignore()`, tok bi se nakon prvog prolaza `for`-petlje zaustavio na desetom znaku. Ako podaci u prvom retku imaju više od deset znakova, u sljedeæem prolazu petlje bi se èitanje nastavilo od tog znaka pa bi kao iduæi podatak bio uèitan nastavak prvog retka.

Funkcijski član `peek()` očitava sljedeći znak, ali pritom taj znak i dalje ostaje na toku, a član `putback()` vraća željeni znak na tok. Ilustrirajmo njihovu primjenu jednostavnim primjerom u kojem se učitava tekst utipkan u jednom retku. Pretpostavimo da će tekst biti proizvoljna kombinacija riječi i brojeva, međusobno razmaknutih prazninama. Program će te riječi i brojeve ispisati u zasebnim recima. Da bismo znali koji tip podatka slijedi, trebamo očitati prvi znak niza: ako je slovo ili znak podcrtavanja, tada slijedi riječ; ako je broj, decimalna točka ili predznak, imamo broj. Početni znak, međutim, ne smijemo izlučiti s toka, jer on mora ući u niz koji tek treba izlučiti. U primjeru ćemo koristiti funkcije za razlučivanje tipa znaka, deklarirane u standardnoj datoteci zaglavlja `ctype.h`, tako da to zaglavlje valja uključiti na početku programa.

```
char znak;
char rijec[80];
double broj;

while ((znak = cin.peek()) != '\n') {
    // preskače praznine:
    if (isspace(znak)) cin.get(znak);
    // je li prvi znak niza znamenka, predznak ili dec. točka?
    else if (isdigit(znak) || znak == '+' || znak == '-' ||
             znak == '.') {
        int predznak = (znak == '-') ? -1 : +1;
        // izlučuje predznak s toka
        if (znak == '-' || znak == '+') cin >> znak;
        if (!(isdigit(znak = cin.peek()) || znak == '.')) {
            cout << "Pogreška: iza predznaka mora biti "
                    "broj!" << endl;
            break;
        }
        if (cin.peek() == '.') {
            // izlučuje decimalnu točku da bi provjerio
            // slijedi li iza nje znamenka
            cin >> znak;
            if (!isdigit(cin.peek())) {
                cout << "Pogreška: iza točke mora biti "
                        "broj!" << endl;
                break;
            }
            // ako je OK, vraća decimalnu točku na tok
            cin.putback(znak);
        }
        cin >> broj;
        cout << (broj * predznak) << endl;
        if (!isspace(cin.peek())) {
```

```
            cout << "Pogreška: iza broja mora biti razmak!"
                 << endl;
            break;
        }
    }
    // ako je prvi znak niza slovo ili podcrtavanje:
    else if (isalpha(znak) || znak == '_') {
        int i = 0;
        // izlučuje pojedinačno znakove sve dok je slovo,
        // broj ili podcrtavanje
        do {
            rijec[i++] = cin.get();
        } while(isalnum(znak = cin.peek()) || znak == '_');
        rijec[i] = '\0';
        cout << rijec << endl;
        if (!isspace(cin.peek())) {
            cout << "Pogreška: iza riječi mora biti razmak!"
                 << endl;
            break;
        }
    }
    // sve ostalo "guta":
    else
        cin >> znak;
}
```

Funkcijski èlan `peek()` poziva se u gornjem primjeru na nekoliko mjesta, prilikom ispitivanja sljedeæeg znaka na toku. On nema argumenata, a kao rezultat vraæa kôd (tipa `int`) sljedeæeg znaka na toku.

Funkcijski član `putback()` se poziva samo na jednom mjestu. Naime, nakon što se učita predznak, treba provjeriti slijedi li decimalna točka, a iza nje znamenka. Budući da se funkcijom `peek()` može vidjeti samo jedan znak, da bismo provjerili znamenku iza decimalne točke valja prvo izlučiti decimalnu točku s toka – tek tada možemo baciti pogled na sljedeći znak. Nakon što se uvjerimo da je sljedeći znak znamenka, funkcijom `putback()` vraćamo decimalnu točku na tok, da bismo broj učitali operatorom `>>`. Funkcijski član `putback()` kao argument prihvaća znak (tipa `char`) koji treba vratiti na tok, a vraća referencu na pripadajući objekt tipa `istream`.

Obratimo još pažnju na to kako se provelo razlikovanje pojedinih vrsta znakova. To je provedeno pomoću funkcija deklariranih u zaglavlju `ctype.h`:

- `isspace()` vraæa `true` ako je znak naveden kao argument funkciji praznina (`' '`), tabulator (`'\t'`), znak za povrat (engl. *return*, `'\r'`), za novi redak (`'\n'`) ili pomak papira (`'\f'`).
- `isdigit()` vraæa `true` ako je znak naveden kao argument funkciji decimalna znamenka (`'0'`, `'1'`... `'9'`).

- `isalpha()` vraæa `true` ako je znak naveden kao argument funkciji slovo (`'a'`...`'ž'`, `'A'`...`'Ž'`[†]).
- `isalnum()` vraæa `true` ako je znak naveden kao argument funkciji slovo ili decimalna znamenka (tj. `isalpha() || isdigit()`)

Jednostavnije je (a u veæini sluèajeva i daleko pouzdanije) koristiti ove funkcije nego provoditi ispitivanja oblika:

```
('a' <= znak && znak <= 'z') || ('A' <= znak && znak <= 'Z')
```

Naime, znakovi ne moraju biti na svakom raèunalu složeni po (najèešæem) ASCII slijedu, za koji bi gornji uvjet odgovarao pozivu funkcije `isalpha()`. Dodatni argument za korištenje gornjih funkcija su nacionalno-specifièna slova (`'è'`, `'È'`, `'æ'`...), koja se ne uklapaju niti u jedan međunarodni slijed znakova.

Navedena tri funkcijska člana klase `istream` svoju prvenstvenu primjenu naći će za pisanje jezičnih procesora – programa koji učitavaju tekst (znak po znak) napisan u nekom simboličkom jeziku, provjeravaju sintaksu teksta te interpretiraju tekst pretvarajući ga u interne naredbe.

Funkcijski članovi `seekg()` i `tellg()` omogućavaju kontrolirano pomicanje duž toka. Član `tellg()` vraća trenutnu poziciju u toku, a `seekg()` se postavlja na zadanu poziciju. Njihova primjena bit će prikazana u odsječcima koji slijede, jer se koriste primarno za kontrolu pristupa datotekama.

18.6. Kontrola uèitavanja i ispisa

Ispisi u svim dosadašnjim primjerima su bili neformatirani – podaci su bili pretvoreni u niz znakova prema podrazumijevanim pravilima za pripadajuæi tip podataka te kao takvi ispisani. Međutim, èesto želimo kontrolirati izgled ispisa. Tipièan primjer je tablièni ispis rezultata, gdje želimo poravnati podatke u pojedinim recima i posložiti ih jedan ispod drugoga.

Većina kontrola za unos i ispis podataka deklarirana je u klasi `ios`. Već smo na početku poglavlja spomenuli da je ta klasa osnovna za klase `istream` i `ostream`, tako da su sve kontrole dohvatljive i izvedenih klasa. Osim toga, klasa `ios` sadrži i pokazivač na klasu `streambuf` koja je omogućava međupohranjivanje pri izlučivanju podataka s toka ili umetanju na tok. Pokazivač je moguće dohvatiti te pomoću njega podešavati svojstva međupohranjivanja.

18.6.1. Vezivanje tokova

Vezivanje tokova osigurava da se izlazni tok isprazni prije nego što poène izluèivanje s ulaznog toka (i obrnuto). Ilustrirajmo to sljedeæim primjerom [Stroustup91]:

[†] Hoæe li i koji nacionalni znakovi biti identificirani kao slova ovisi o parametrima definiranima u klasi `locale`.

```
char zaporka[15];
cout << "Zaporka: ";
cin >> zaporka;
```

Izlazni tok `cout` koristi međuspremnik, tako da opæenito ne postoji garancija da æe se poruka `"Zaporka: "` pojaviti na zaslonu prije nego što se eksplicitno (npr. pomoæu `endl`) ili implicitno (završetkom programa) isprazno izlazni tok. Sreæom, između ulaznog toka `cin` i izlaznog toka `cout` uspostavljena je veza koja osigurava da se prije svakog unosa izlazni tok isprazni. Ova veza se automatski uspostavlja prilikom stvaranja globalnih objekata `cin` i `cout`. Takva veza postoji i između izlaznih tokova `cerr`, odnosno `clog` i ulaznog toka `cin`.

Pozivom funkcijskog člana `tie()` može se ulazni tok vezati na bilo koji izlazni tok. Argument u pozivu funkcije mora biti izlazni tok na koji se dotični ulazni tok vezuje, a funkcija vraća pokazivač na izlazni tok na kojeg je ulazni tok bio prethodno vezan ili nul-pokazivač ako ulazni tok prethodno nije bio vezan. Ako se kao argument navede `0`, prekida se veza između ulaznog i izlaznog toka, a ako se `tie()` pozove bez argumenata, vraća se trenutačna veza. Tako su tokovi `cin` i `cout` vezani naredbom `cin.tie(&cout)` koja se izvodi prije ulaska u funkciju `main()`.

Kada je neki `ostream` vezan na `istream`, `ostream` se isprazni čim se pokuša nešto učitati s `istream` toka. Stoga se gornje naredbe za ispis i učitavanje interpretiraju kao:

```
cout << "Zaporka: ";
cout.flush();
cin >> zaporka;
```

18.6.2. Širina ispisa

Èesto želimo da podaci koji se ispisuju zauzimaju toèno određenu širinu na zaslonu, neovisno o broju znamenaka u broju ili znakova u nizu, tako da budu poravnati po stupcima. Širinu polja za ispis ili uèitavanje brojeva ili znakovnih nizova možemo podesiti funkcijskim èlanom `width()`. Cjelobrojni argument određuje broj znakova koji æe ispis ili upis zauzimati. Na primjer:

```
cout << '>';
cout.width(6);                    // postavlja širinu ispisa
cout << 1234 << '<' << endl;
```

Izvođenjem ovih naredbi, na zaslonu æe se ispisati:

```
>  1234<
```

Podešena širina traje samo jedan ispis – nakon toga ona se postavlja na podrazumijevanu širinu. Zbog toga æe izvođenje kôda

```
cout << '>';
cout.width(6);
cout << 1234 << '<' << endl;
cout << '>' << 1234 << '<' << endl;
```

dati sljedeæi ispis:

```
>  1234<
>1234<
```

Ako se navede širina manja od neophodno potrebne za ispis, naredba za širinu æe se ignorirati. Zbog toga æe izvoðenjem naredbi

```
cout << '>';
cout.width(2);
cout << 1234 << '<' << endl;
```

biti ispisan kompletan broj `1234`, unatoè tome što je kao argument funkcijskom èlanu `width()` navedena širina od 2 znaka:

```
>1234<
```

Funkcija `width()` kao rezultat vraæa prethodno zadanu širinu ispisa. Ako se ne navede argument, funkcija `width()` vraæa trenutno važeæu širinu.

Sva navedena razmatranja vrijede i za znakovne nizove. Funkcijski član `width()` se može pozvati i za ulazne tokove – u tom slučaju se njime ograničava broj učitanih znakova, slično kao što smo to mogli postići funkcijskim članom `get()`.

Širine ispisa podataka mogu se podešavati i pomoću manipulatora `setw()`, koji će biti opisan kasnije u ovom poglavlju.

18.6.3. Popunjavanje praznina

Ako je širina polja za ispis zadana `width()` funkcijom, šira od podatka, nedostajuæi znakovi bit æe popunjeni bjelinama. Želimo li umjesto bjelina ispisati neki drugi znak, najjednostavnije æemo to uraditi funkcijskim èlanom `fill()`. On kao argument prihvaæa znak kojim želimo popuniti praznine, a vraæa kôd znaka koji je prethodno bio korišten za popunjavanje.

U sljedećem primjeru funkcijskim članom `fill()` se praznine popunjavaju zvjezdicama:

```
cout << '>';
cout.width(12);
cout.fill('*');
cout << 1234 << '<' << endl;
```

Izvoðenjem gornjih naredbi ispisat æe se:

```
>********1234<
```

Znak za popunjavanje praznina se može podesiti i manipulatorom `setfill()` koji æe biti opisan u odjeljku posveæenom manipulatorima. Postavljeni znak ostaje zapamæen dok ga eventualno ponovo ne promijenimo.

18.6.4. Zastavice za formatiranje

U `ios` klasi definirane su i *zastavice* (engl. *flags*) za formatirani ispis i uèitavanje podataka (tablica 18.1). One omoguæavaju preciznu kontrolu ispisa podataka, na primjer poravnavanje ulijevo, udesno, ispis u heksadekadskom, oktalnom ili dekadskom formatu i sl. Buduæi da je naèin na koji su zastavice pohranjene ovisan o implementaciji, za njihovo dohvaæanje i mijenjanje definirani su funkcijski èlanovi `setf()`, `unsetf()` i `flags()`.

Funkcijski član `flags()` služi za dohvaćanje i promjenu kompletnog seta zastavica. U primjeni su nam zanimljiviji funkcijski članovi `setf()` i `unsetf()` koji služe za postavljanje, odnosno brisanje stanja pojedine zastavice. Ilustrirajmo njihovu primjenu sljedećim primjerom:

Tablica 18.1. Zastavice u **`ios`** klasi

zastavica	grupa	znaèenje
`left`	`adjustfield`	poravnanje ulijevo
`right`	`adjustfield`	poravnanje udesno
`internal`	`adjustfield`	predznak lijevo, ostatak desno
`dec`	`basefield`	pretvorba cjelobrojnih ulaznih podataka ili ispis u dekadskoj bazi
`hex`	`basefield`	pretvorba cjelobrojnih ulaznih podataka ili ispis u heksadekadskoj bazi
`oct`	`basefield`	pretvorba cjelobrojnih ulaznih podataka ili ispis u oktalnoj bazi
`fixed`	`floatfield`	ispis brojeva s fiksnom decimalnom toèkom
`scientific`	`floatfield`	ispis brojeva u znanstvenoj notaciji
`showpos`		ispis predznaka + pozitivnim brojevima
`showpoint`		ispis decimalne toèke za sve realne brojeve
`showbase`		ispis prefiksa koji ukazuje na bazu cijelih brojeva
`uppercase`		zamjena nekih malih slova (npr. `'e'`, `'x'`) velikim slovima (npr. `'E'`, `'X'`)
`skipws`		preskakanje praznina pri uèitavanju
`boolalpha`		umetanje i izluèivanje `bool` tipa u slovèanom obliku
`unitbuf`		pražnjenje izlaznog toka nakon svake izlazne operacije

```
// pohranjuje zatečeno stanje
long zastavice = cout.flags();

cout.setf(ios::showpos);    // '+' ispred pozitivnih brojeva
cout.width(8);
cout << 11 << endl;

// poravnaj desno, predznak lijevo
cout.setf(ios::internal, ios::adjustfield);
cout.width(8);
cout << 12 << endl;

// ispis u heksadekadskom formatu
cout.setf(ios::hex, ios::basefield);
cout << 13 << endl;

// ispiši bazu ispred broja
cout.setf(ios::showbase);
cout << 14 << endl;

// ispis širine 6 ...
```

```
cout.width(6);

// ... ispunjeno nulama ...
cout.fill('0');

// ... između 0x i broja
cout.setf(ios::internal, ios::adjustfield);
cout << 15 << endl;

// oktalni ispis
cout.setf(ios::oct);
cout << 16 << endl;

// natrag na dekadski ispis
cout.setf(ios::dec, ios::basefield);
cout << 17. << endl;

// više ne ispisivati '+'
cout.unsetf(ios::showpos);
cout << 18. << endl;

// natrag na početno stanje
cout.flags(zastavice);
```

Izvoðenjem gornjih naredbi dobit æe se sljedeæi ispis:

```
+11
+     12
d
0xe
0x000f
020
+17
18
```

Prije prvog ispisa postavljena je zastavica za ispis predznaka `'+'` ispred pozitivnih brojeva pomoæu zastavice `setpos`. Unatoè tome što je širina polja za ispis postavljena na 8, prvi broj je ispisan uz predznak, jer je podrazumijevano poravnavanje ispisa ulijevo. Tek nakon postavljanja zastavice `internal` (ili eventualno `right`) broj æe biti pomaknut udesno – ove dvije zastavice oèito imaju smisla samo u kombinaciji s èlanom `width()`.

Postavljanjem zastavica `hex` i `oct`, dobiven je ispis cijelih brojeva u heksadekadskom, odnosno oktalnom formatu, s time da je zastavicom `showbase` uključeno ispisivanje niza `"0x"` ispred heksadekadskog, odnosno znaka `'0'` ispred oktalnog broja. Za ispis realnih brojeva zastavice `hex`, `oct` i `showbase` nemaju značenja.

Na kraju primjera naveden je i jedan poziv funkcije `unsetf()` kojom se briše zastavica za ispis `'+'` ispred pozitivnih brojeva.

Neke zastavice su povezane u srodne grupe. Prilikom postavljanja tih zastavica članom `setf` potrebno je navesti grupu kojoj zastavica pripada. Tako je za postavljanje zastavica iz grupa `adjustfield` (`left`, `right`, `internal`), `basefield` (`dec`, `hex`, `oct`) i `floatfield` (`fixed`, `scientific`) potrebno pozvati preopterećenu inačicu funkcije `setf()` koja prihvaća dva argumenta: prvi argument je zastavica, a drugi je grupa. Brisanje tih zastavica, tj. postavljanje na podrazumijevano stanje postiže se tako da se kao prvi argument navede `0`:

```
setf(0, ios::floatfield);
```

Sve zastavice ostaju aktivne sve do ponovne eksplicitne promjene. Naprotiv, funkcijski èlan `width()` postavlja širinu samo sljedeæe naredbe za ispis, odnosno uèitavanje.

Napomenimo da zastavica `skipws` utjeèe samo na ulazni tok. Podrazumijevano je ta zastavica postavljena pa ju nije potrebno eksplicitno postavljati. Zastavica se može ugasiti ako je to potrebno.

18.6.5. Formatirani prikaz realnih brojeva

U prethodnom odjeljku upoznali smo kako je moguæe pomoæu pojedinih zastavica kontrolirati ispis i upis podataka. Formatiranje ispisa i upisa posebno je važno kod obrade brojèanih podataka. Za formatiranje realnih brojeva, naroèito su korisne zastavice `scientific`, `fixed` i `showpoint` te funkcijski èlan `precision()`. Njihovu primjenu ilustrirat æemo sljedeæim primjerom:

```
cout << 10. << endl;
cout.setf(ios::showpoint);  // uvijek ispisuje decimalnu točku
                            // za realne brojeve
cout << 11. << endl;
cout << 12 << endl;         // za cijele brojeve nema efekta

cout.setf(ios::fixed);      // u fiksnom formatu
cout << 12e3 << endl;
cout << 123456e7 << endl;

cout.setf(ios::scientific, ios::floatfield);
                            // u znanstvenom zapisu
cout << 13e4 << endl;
cout << 134 << endl;

cout.setf(ios::uppercase);  // ... s velikim slovom 'E'
cout << 14. << endl;
```

Izvođenje gornjih naredbi rezultirat æe sljedeæim ispisom:

```
10
11.0000
12
12000.000000
1234560000000.000000
1.300000e+05
134
1.400000E+01
```

Postavljanjem zastavice `showpoint` ispisuje se decimalna toèka i prateæe nule za sve realne brojeve (podrazumijevano se ispisuje ukupno šest znamenki); prije postavljanja zastavice, decimalna toèka se ispisuje samo ako broj ima decimalne znamenke. Kao što vidimo iz treæe naredbe za ispis broja 12, zastavica `showpoint` nema nikakvog utjecaja na ispis cijelih brojeva.

Zastavica `fixed` forsira ispis broja u standardnom formatu s decimalnom točkom i šest znamenki desno od decimalne točke, dok zastavica `scientific` forsira ispis u znanstvenom zapisu. Za cijele brojeve niti ove dvije zastavice nemaju utjecaja.

Zastavica `uppercase` prouzročit će ispis velikog slova `'E'` ispred eksponenta u znanstvenom formatu te velikog slova `'X'` ispred heksadekadskog broja.

Standardno se realni brojevi ispisuju sa do 6 znamenki. Broj koji se ne može prikazati u toliko znamenki bit će ispisan u znanstvenom zapisu. Tako će naredbe

```
cout << 12.34 << endl;
cout << 1234567890123456. << endl;
cout << 0.00000012345 << endl;
```

ispisati

```
12.34
1.23457e+15
1.2345e-07
```

Funkcijski èlan `precision()` fiksira ukupan broj znamenki koji æe se prikazati. Stavimo li ispred prethodne tri naredbe za ispis naredbu

```
cout.precision(2);
```

umjesto prethodnog ispisa, na zaslonu æemo dobiti

```
12
1.2e+15
1.2e-07
```

Ako su postavljene zastavice `fixed` ili `scientific`, tada `precision()` odreðuje broj znamenki koje æe se prikazati desno od decimalne toèke. To znaèi da æe izvoðenje naredbi

```
cout.precision(2);

fout.setf(ios::fixed, ios::floatfield);
fout << 12.34 << endl;
fout << 1234567890123456. << endl;
fout << 0.00000012345 << endl;

fout.setf(ios::scientific, ios::floatfield);
fout << 12.34 << endl;
fout << 1234567890123456. << endl;
fout << 0.00000012345 << endl;
```

ispisati brojeve u sljedeæem formatu:

```
12.34
1234567890123456.00
0.00
1.23e+01
1.23e+15
1.23e-07
```

Treba naglasiti da se prilikom odbacivanja znamenki, zadnja znamenka zaokružuje prema uobièajenim aritmetièkim pravilima.

Zadatak. *Napišite funkciju koja će ispisivati realne brojeve jedan ispod drugoga s poravnatim decimalnim točkama, kao na primjer:*

```
   1.2
1234.
   9.123
```

18.6.6. Manipulatori

U prethodnim odjeljcima smo vidjeli kako možemo pozivima pojedinih funkcijskih èlanova kontrolirati format ispisa. Iako ovakav naèin omoguæava potpunu kontrolu nad formatom, on ima jedan veliki nedostatak: naredbe za postavljanje formata morali smo pisati kao zasebne naredbe. Na primjer:

```
cout.width(12);
cout.setf(ios::showpos);
cout << 1.;
```

Ovakvim pristupom gubi se “nit” kontinuiteta izlazne naredbe – iako su sve tri naredbe vezane za ispis jednog podatka, morali smo ih napisati odvojeno.

Uvođenjem *manipulatora* (engl. *manipulators*) omogućena je bolja povezanost kontrole formata i operatora izlučivanja, odnosno umetanja. Umjesto da se naredba za

formatiranje poziva kao zaseban funkcijski član, ona se jednostavno umeće na tok. Tako gornje naredbe možemo pomoću manipulatora napisati preglednije, na sljedeći način:

```
cout << setw(12) << setiosflag(ios::showpos) << 1.;
```

Standardni manipulatori navedeni su u tablici 18.2. Kao što se iz tablice vidi, postoje manipulatori koji uzimaju argumente i oni bez argumenata. Svi manipulatori koji prihvaæaju argumente definirani su u zaglavlju `iomanip.h` te ako ih želimo koristiti, ne

Tablica 18.2. Standardni ulazno-izlazni manipulatori

manipulator	zaglavlje	znaèenje
`setw(int)`	`iomanip.h`	širina polja za ispis/uèitavanje
`setfill(int)`	`iomanip.h`	znak za popunjavanje praznina
`dec`	`iostream.h`	dekadski prikaz
`hex`	`iostream.h`	heksadekadski prikaz
`oct`	`iostream.h`	oktalni prikaz
`setbase(int)`	`iomanip.h`	postavi bazu
`setprecision(int)`	`iomanip.h`	broj znamenki
`setiosflags(long)`	`iomanip.h`	postavlja zastavicu
`resetiosflag(long)`	`iomanip.h`	briše zastavicu
`flush`	`iostream.h`	prazni izlazni tok
`endl`	`iostream.h`	prazni izlazni tok i umeæe znak za novi redak (`'\n'`)
`ends`	`iostream.h`	umeæe zakljuèni nul-znak na izlazni tok
`ws`	`iostream.h`	ignorira praznine na ulaznom toku

smijemo zaboraviti ukljuèiti to zaglavlje. Vjerujemo da æe pažljiviji èitatelj prepoznati znaèenje manipulatora veæ na temelju sliènosti imena sa zastavicama i funkcijskim èlanovima klase `ios` koje smo opisali u prethodnim odjeljcima. Sve što smo tamo rekli vrijedi i za ove manipulatore. Tako `setw()` manipulator utjeèe samo na sljedeæu ulaznu ili izlaznu operaciju, pa ga po potrebi treba ponavljati. Manipulatori `setiosflag()` i `resetiosflag()` služe za postavljanje i brisanje pojedinih zastavica, slièno kao funkcijski èlanovi `setf()` i `unsetf()` – prihvaæaju jednake argumente (interno definiranog tipa `fmtflag`), ali se mogu umetati u ulazni, odnosno izlazni tok:

```
cout << setiosflag(ios::showpos) << 1.23 << endl;
```

Napomenimo da manipulatori `flush`, `endl` i `ends` djeluju samo za izlazne tokove, dok manipulator `ws` djeluje samo za ulazne tokove.

Zadatak. *Napišite programske odsječke iz prethodna dva odjeljka tako da umjesto funkcijskih članova upotrijebite manipulatore.*

Obratimo pažnju na to kako su manipulatori realizirani. Naime, i mnogim C++ znalcima je trebalo dosta vremena dok su uspjeli shvatiti kojeg je tipa zapravo taj `endl` koji se

upisuje u izlaznu listu. Radi se, naime, o jednom lukavom triku: svi manipulatori bez parametra su realizirani kao funkcija koja kao parametar uzima referencu na tok. Tako u datoteci zaglavlja `iostream.h` postoji deklaracija te funkcije koja ispisuje znak `'\n'`. No kada se u ispisnoj listi navede samo `endl` bez zagrada, to je zapravo pokazivaè na funkciju. Operator `<<` je preoptereæen tako da prihvaæa pokazivaè na funkciju, te on u biti poziva funkciju navodeæi tok kao parametar. Evo kako je taj operator definiran:

```
typedef ostream &(*Omanip)(ostream &);

ostream &operator<<(ostream &os, Omanip f) {
    return f(os);
}
```

Tip `Omanip` je pokazivaè na funkciju koja kao parametar uzima referencu na `ostream` i vraæa referencu na `ostream`. Izraz

```
cout << endl;
```

se interpretira kao

```
operator<<(cout, endl);
```

Na taj naèin moguæe je dodavati vlastite manipulatore. Na primjer, mogli bismo dodati manipulator `cajti` koji ispisuje vrijeme na tok. Napominjemo da nije potrebno dodavati novi operator `<<` za ispis; on je veæ definiran u `iostream.h`. Evo kôda manipulatora:

```
#include <time.h>
#include <iostream.h>

ostream &cajti(ostream &os) {
    time_t vrij;
    time(&vrij);
    tm *v = localtime(&vrij);
    os << v->tm_hour << ":" << v->tm_min << ":" << v->tm_sec;
    return os;
}
```

Manipulator koristi standardno zaglavlje `time.h` za èitanje sistemskog sata. Varijabla `vrij` je tipa `time_t`: u nju se smješta broj sekundi proteklih od 01.01.1970. godine. To se vrijeme pomoæu funkcije `localtime` pretvara u strukturu tipa `tm` koja sadržava sat, minutu, sekundu, datum i još neke podatke. Ta struktura je statièka i definirana je unutar sistemske biblioteke, a funkcija `localtime` vraæa pokazivaè na nju. Evo kako se naš manipulator može pozvati:

```
cout << "Sada je točno " << cajti << endl;
```

Manipulatori također mogu uzimati parametre. U tom sluèaju se mehanizam manipulatora ponešto komplicira. Na primjer, manipulator `width()` uzima cjelobrojni parametar. `width()` je funkcija koja vraæa objekt koji se zatim ispisuje na tok. Kako je dosta nezgodno za svaki manipulator dodavati poseban objekt koji æe obraðivati odreðeni manipulator, u datoteci `iomanip.h` je definirana klasa `omanip` koja u sebi èuva adresu manipulatorske funkcije i sam parametar. Klasa je definirana predloškom, tako da se za svaki tip predloška klasa može instancirati bez potrebe za ponovnim pisanjem kôda. Također, operator `<<` je definiran predloškom tako da prihvaæa bilo koju varijantu objekta `omanip`. Evo kako je sve to realizirano u standardnoj biblioteci:

```
template <class T>
class omanip {
public:
    ostream &(*fn)(ostream &, T);
    T arg;
    omanip(ostream &(*f)(ostream &, T), T a) :
                                        fn(f), arg(a) {}
};

template <class T>
ostream &operator<<(ostream &os, omanip<T> & obj) {
    return obj.fn(os, obj.arg);
}

ostream &postavi_sirinu(ostream &os, int sirina);

omanip<int> width(int sirina) {
    return omanip<int>(postavi_sirinu, sirina);
}
```

Nazivi pojedinih èlanova su izmijenjeni u odnosu na standardnu biblioteku, te su deklaracije uèinjene razumljivijima, no suština je ostala saèuvana. Klasa `omanip` je parametrizirana tipom koji određuje argument manipulatora. Funkcija `width()` prilikom poziva stvara privremeni objekt te klase u koji pakira adresu funkcije `postavi_sirinu` i samu širinu. Taj objekt preuzima operator `<<` koji zauzvrat poziva funkciju iz objekta i prosljeđuje joj zadani parametar. Iskoristimo to kako bismo dodali manipulator `razmaka` koji na tok ispisuje onoliko razmaka koliko mu je zadano parametrom. Evo potrebnih "dodataka" sustavu tokova:

```
#include <iostream.h>
#include <iomanip.h>

ostream &pisi_razmake(ostream &os, int koliko) {
    while (koliko--) os << ' ';
    return os;
}

omanip<int> razmaka(int koliko) {
```

```
    return omanip<int>(pisi_razmake, koliko);
}
```

Evo kako se taj manipulator može pozvati:

```
int main() {
    cout << "Amo" << razmaka(5) << "!" << endl;
    return 0;
}
```

18.7. Datoteèni ispis i uèitavanje

Ispis rezultata i poruka na zaslonu te unos podataka pomoæu tipkovnice dostatni su samo za najjednostavnije aplikacije; u ozbiljnijim programima neophodna je moguænost pohranjivanja podataka u datoteke na disku i moguænost uèitavanja podataka iz datoteka. Iako je princip ulaznih i izlaznih tokova u potpunosti primjenjiv i na ispis podataka u datoteke, odnosno uèitavanje iz datoteka, postoje specifiènosti pri radu s njima. Za ispis na zaslon ili upis preko tipkovnice nismo trebali stvarati nikakve objekte – neophodni globalni objekti `cin`, `cout`, `cerr` i `clog` se stvaraju automatski na poèetku svakog programa u koji je ukljuèena datoteka zaglavlja `iostream.h`. Naprotiv, za rad sa datotekama programer mora sam stvoriti objekt (ulazni ili izlazni tok), vezati ga na neku datoteku, specificirati format zapisa podataka. Tim i sliènim specifiènostima komunikacije s datoteke posvetiti æemo se do kraja ovog poglavlja.

18.7.1. Klase `ifstream` i `ofstream`

Za rad s datotekama standardom su definirane tri klase: `ifstream`, `ofstream` i `fstream`. Klasa `ifstream` je naslijeđena od klase `istream` i namijenjena je prvenstveno za uèitavanje podataka iz datoteke. Klasa `ofstream` je naslijeđena od klase `ostream` i namijenjena u prvom redu za upis podataka u datoteku. Klasa `fstream` omoguæava i upis i uèitavanje podataka u, odnosno iz datoteka. Sve su tri klase deklarirane u datoteci zaglavlja `fstream.h`.

Da bismo mogli koristiti izlazne i ulazne tokove za upis u datoteke, odnosno čitanje iz datoteka, u program treba uključiti datoteku zaglavlja `fstream.h`. Budući da su klase navedene u toj datoteci izvedene iz klasa deklariranih u datoteci `iostream.h`, potonju u tom slučaju ne treba dodatno uključivati.

Pogledajmo za poèetak vrlo jednostavan primjer – program koji æe uèitati sadržaj neke tekstovne datoteke i ispisati ga na zaslonu:

```
#include <fstream.h>
#include <stdlib.h>         // uključeno zbog funkcije exit()
```

```
int main(int argc, char* argv[]) {
    // ako je u pozivu programa naveden samo jedan parametar:
    if (argc == 2) {
        // stvara se ulazni tok datnica klase ifstream i
        // pridružuje datoteci navedenoj kao parametar
        ifstream datnica(argv[1]);
        // ako otvaranje datoteke nije uspjelo:
        if (!datnica) {
            cerr << "Nemrem otpreti potraživanu datoteku "
                 << argv[1] << endl;
            exit(1);
        }
        char zn;
        // znak po znak, sadržaj datoteke:
        while (datnica.get(zn)) cout << zn;
    }
    else if (argc < 2) {
        // ako se program pokrene bez parametra (imena
        // datoteke koju treba ispisati), ispisuje uputu:
        cout << "Program za ispis sadržaja datoteke" << endl;
        cout << "Korištenje: ISPIS <ime_datoteke>" << endl;
    }
    else {
        cerr << "Preveć parametara u pozivu programa" << endl;
        exit(1);
    }
    return 0;
}
```

Ime datoteke èiji sadržaj se želi ispisati navodi se kao parametar pri pokretanju programa. U sluèaju da nije naveden parametar ispisuje se kratka uputa, a ako se navede previše parametara ispisuje se poruka o pogreški.

Usredotočimo se na nama najvažniji dio programa: inicijalizaciju ulaznog toka i učitavanje podataka iz datoteke. Ulazni tok je objekt `datnica` klase `ifstream`. Prilikom inicijalizacije kao argument konstruktoru se prenosi ime datoteke. Zatim se ispituje da je li inicijalizacija uspjela. Ako inicijalizacija nije uspjela (na primjer ako navedena datoteka nije pronađena ili se ne može čitati), program ispisuje poruku o pogreški i prekida daljnje izvođenje.

Prilikom inicijalizacije toka valja provjeriti je li inicijalizacija bila uspješno provedena, tj. je li datoteka uspješno otvorena za uèitavanje ili upisivanje.

U protivnom, daljnje izvoðenje programa neæe imati nikakvog efekta.

Ako je inicijalizacija bila uspješna i datoteka otvorena za čitanje, u `while`-petlji se iščitava sadržaj datoteke. Za čitanje se koristi funkcijski član `get(char &)`, opisan u odjeljku 18.5.3. Budući da je klasa `ifstream` naslijeđena iz `istream`, za učitavanje iz

datoteke na raspolaganju su nam svi članovi klase `istream`, od kojih smo većinu upoznali.

Korištenje izlaznih tokova za upis u datoteku predočit ćemo sljedećim primjerom – programom koji kopira sadržaj tekstovne datoteke, izbacujući pritom sve višestruke praznine i prazne retke. Ime izvorne datoteke neka se navodi kao prvi parametar prilikom pokretanja programa (slično kao u prethodnom primjeru), a ime preslikane datoteke neka je drugi parametar. Zbog sličnosti s prethodnim primjerom, izostavit ćemo provjere parametara i navesti samo "efektivni" dio kôda:

```
//...
ifstream izvornik(argv[1]);
if (!izvornik) {
    cerr << "Ne mogu otvoriti ulaznu datoteku "
         << argv[1] << endl;
        exit(1);
}

ofstream ponornik(argv[2]);
if (!ponornik) {
    cerr << "Ne mogu otvoriti izlaznu datoteku "
         << argv[2] << endl;
        exit(1);
}

char znNiz[80];;
while (izvornik >> znNiz) {
    ponornik << znNiz;
    if (izvornik.peek() == '\n')
        ponornik << endl;
    else if (izvornik.peek() != EOF)
        ponornik << " ";
}

if (!izvornik.eof() || ponornik.bad())
    cerr << "Uuups! Nekaj ne štima" << endl;
//...
```

U ovom primjeru otvaraju se dva toka: `izvornik` klase `ifstream` koji se pridružuje datoteci iz koje se sadržaj išèitava, te `ponornik` klase `ofstream` koji se pridružuje datoteci u koju se upisuje novi sadržaj. Prilikom inicijalizacije oba toka, konstruktorima se kao argumenti prenose imena pripadajuæih datoteka, a potom se provjerava je li inicijalizacija uspješno provedena.

U `while`-petlji se sadržaj izvorne datoteke učitava pomoću operatora `>>` u polje znakova `znNiz`. To znači da će se izvorna datoteka čitati riječ po riječ. Budući da operator `>>` preskače sve praznine između riječi, u izlazni tok treba umetati po jednu bjelinu nakon svake riječi. Ako se iza riječi pojavio znak za novi redak on se upisuje umjesto praznine – u protivnom bi cijela izlazna datoteka bila ispisana u jednom retku. Bjelina se ne smije dodati iza zadnje riječi u datoteci, što se provjerava ispitivanjem

```
if (izvornik.peek() != EOF)
```

`EOF` je simbolièko ime znaka koji oznaèava kraj datoteke (engl. *End-Of-File*), definirano u zaglavlju `stdio.h`.

Na kraju primjera uočimo pozive dva funkcijska člana koji provjeravaju stanje tokova: `eof()` i `bad()`. Tim se ispitivanjem provjerava je li nastupila pogreška u izlaznom toku prije nego što je dosegnut kraj ulaznog toka.

Zadatak. *Napišite program koji kopira sadržaj datoteke u drugu datoteku, znak po znak.*

18.7.2. Otvaranje i zatvaranje datoteke

Da bi se podaci mogli pohraniti u ili uèitati iz neke datoteke, prethodno je potrebno datoteku otvoriti. Ako datoteka u koju se podaci upisuju ne postoji, treba ju stvoriti, a ako datoteka koju želimo èitati nije dostupna, valja prijaviti pogrešku. Nakon obavljenog prijenosa podataka, datoteka se mora zatvoriti. Ovo je posebno važno kod upisa podataka u datoteku, jer se tek prilikom zatvaranja na disku ažuriraju podaci o datoteci (duljina, vrijeme pristupa i slièno). Prema tome, možemo razluèiti tri faze prijenosa podataka:

1. otvaranje datoteke,
2. ispis ili uèitavanje podataka i
3. zatvaranje datoteke.

U dosadašnjim primjerima smo datoteku otvarali prilikom inicijalizacije toka, navodeæi ime datoteke u konstruktoru toka. Datoteka se zatvarala prilikom uništavanja toka, što se automatski dogaða prilikom izlaska iz bloka u kojem je tok definiran. U prethodnim primjerima to se dešavalo pri izlasku iz glavne funkcije naredbom `return` ili funkcijom `exit()`.

Datoteka se može otvoriti i naknadno, pomoću funkcijskog člana `open()`. Naime, konstruktori klasa `ifstream` i `ofstream` definirani su u po dvije preopterećene verzije:

```
ifstream();
ifstream(const char *ime_datoteke, openmode mod = in);

ofstream();
ofstream(const char *ime_datoteke, openmode mod = out);
```

Verzije koje smo dosad koristili kao argument prihvaæaju ime datoteke na koju se tok vezuje. Koristi li se konstruktor bez parametara, tada se tok veže na datoteku naknadno, pomoæu funkcijskog èlana `open()`. Tako smo u primjeru sa stranice 543 mogli pisati:

```
ifstream izvornik;
izvornik.open(argv[1]);

ofstream ponornik;
ponornik.open(argv[2]);
```

Uspjeh otvaranja datoteke ispituje se tek nakon poziva funkcijskog èlana `open()` na potpuno identièan naèin kao i u primjeru na stranici 543.

Ako datoteke treba zatvoriti prije (implicitnog ili eksplicitnog) poziva destruktora, to možemo učiniti funkcijskim članom `close()`:

```
izvornik.close();
ponornik.close();
```

To nam omoguæava da tok vežemo na neku drugu datoteku, iako se to vrlo rijetko koristi.

Kao što se vidi iz gore navedenih deklaracija konstruktora, verzije konstruktora klasa `ifstream` i `ofstream` koje prihvaćaju ime datoteteke, prihvaćaju i drugi argument – *mod*. On ima podrazumijevanu vrijednost za pojedini tip toka, tako da ga nije neophodno uvijek navoditi. Za `ifstream` je podrazumijevani mod `in`, tj. čitavanje iz datoteke, a za `ofstream` podrazumijevani mod je `out`, tj. upis u datoteku. Potpuno isto vrijedi i za funkcijske članove `open()` – ako se datoteka želi otvoriti u modu različitom od podrazumijevanog, uz ime datoteke može se proslijediti mod.

Modovi za otvaranje datoteka definirani su kao bitovna maska u klasi `ios` (tablica 18.3). Više različitih modova može se međusobno kombinirati koristeći operator `|` (bitovni *ili*). Na primjer, želimo li da se prilikom upisivanja u datoteku novi sadržaj

Tablica 18.3. Modovi otvaranja datoteka definirani u klasi **`ios`**

mod	znaèenje
`app`	upis se obavlja na kraju datoteke (*append*)
`ate`	datoteka se otvara i skaèe se na njen kraj (*at-end*)
`binary`	upis i uèitavanje se provode u binarnom modu
`in`	otvaranje za èitanje
`nocreate`	vraæa se pogreška ako datoteka ne postoji
`noreplace`	vraæa se pogreška ako datoteka postoji
`out`	datoteka se otvara za pisanje
`trunc`	sadržaj postojeæe datoteke se briše prilikom otvaranja *(truncate)*

nadoveže na već postojeći, u naredbi za otvaranje toka ćemo kao drugi parametar navesti, uz podrazumijevani mod `out`, i mod `app`. U primjeru na 543. stranici to bi značilo da izlazni tok trebamo inicijalizirati naredbom

```
ofstream ponornik(argv[2], ios::out | ios::app);
```

Ako prevedemo ovakav program i više puta ga uzastopce pozovemo s istim parametrima u komandnoj liniji (npr. `ulazna.dat izlazna.dat`), sadržaj ulazne datoteke æe se prilikom svakog poziva dodati izlaznoj.

Suprotan efekt ima mod `trunc` koji "kreše" datoteku prilikom njenog otvaranja. Budući da je to podrazumijevano ponašanje prilikom otvaranja datoteke za upis, ne treba ga posebno navoditi. Binarni mod će biti opisan u odjeljku 18.7.4.

Maštoviti čitatelj bi se mogao dosjetiti i pokušati promijeniti podrazumijevane modove tokova `ifsream` i `ofstream` (`in`, odnosno `out`). I to je moguće; prevoditelj neće prijaviti pogrešku napišemo li sljedeće naredbe:

```
ifstream izlazniUlaz("kamo.dat", ios::out);
ofstream ulazniIzlaz("oklen.dat", ios::in);
```

Meðutim, od toga neæe biti neke vajde, jer su u klasi `ofstream` (odnosno u klasi `ostream` koju ova nasljeðuje) definirani samo funkcijski èlanovi i operator `<<` za upis u datoteku, dok su u klasi `ifstream` (odnosno njenoj osnovnoj klasi `istream`) definirani samo funkcijski èlanovi i operator `>>` za uèitavanje. Nije potrebno mudrovati: jednostavno koristite ulazne tokove za ulazne operacije, a izlazne tokove za izlazne operacije.

18.7.3. Klasa `fstream`

Ponekad trebamo iz iste datoteke podatke èitati i u nju pohranjivati podatke. Na primjer, program za evidenciju stanja na tekuæem raèunu uèitavat æe iz datoteke zadnje stanje na raèunu, dodati ili oduzeti nove uplate, odnosno izdatke te potom pohraniti novi saldo u datoteku. Kako klase `ifstream` i `ofstream` podržavaju komunikaciju samo u jednom smjeru, njihovo korištenje bi iziskivalo da prvo otvorimo datoteku za uèitavanje, uèitamo podatke te zatvorimo datoteku, a tek potom otvorimo istu datoteku za upis, upišemo novi podatak i na kraju zatvorimo izlazni tok. Takvo programiranje bi bilo dosta zamorno, a rezultiralo bi i vrlo sporim programom. Postupak pojednostavljuje klasa `fstream` izvedena iz klase `iostream`, koja je pak izvedena iz klasa `istream` i `ostream` – ona u sebi ima ukljuèene sve operacije za pisanje i èitanje jedne te iste datoteke[†]. Ilustrirajmo primjenu klase `fstream` primitivnim programom za voðenje evidencije stanja na raèunu:

```
//...
float staroStanje;
fstream strujiStruja("MojaTuga.dat",
                     ios::in | ios::out | ios::nocreate);

if (!strujiStruja) {
    cerr << "Ne postoji datoteka \"MojaTuga.dat\"" << endl
         << "Bit će stvorena nova." << endl;
    strujiStruja.open("MojaTuga.dat", ios::out);
    staroStanje = 0.;
```

[†] Klasa `fstream` nije navedena u standardu, unatoè èinjenici da se spominje u veæini referentnih knjiga. Stoga ne možemo sa sigurnošæu reæi hoæe li ona biti dostupna uz sve biblioteke tokova (no sve su prilike da hoæe).

```
}
else {
    do {
        strujiStruja >> staroStanje;
    } while(strujiStruja);
}
cout << setiosflags(ios::fixed) << setprecision(2)
     << setw(15) << "staro stanje:"
     << setw(8) << staroStanje  << endl;

cout << setw(15) << "uplate:";
float uplate;
cin >> uplate;

cout << setw(15) << "rashodi:";
double rashodi;     // za rashode uvijek koristite double!
cin >> rashodi;
// ispisuje 25 znakova '-' za podcrtavanje
cout << setw(25) << setfill('-') << " " << endl
     << setfill(' ');
float novoStanje = staroStanje + uplate - rashodi;
strujiStruja.clear();
strujiStruja << novoStanje << endl;
if (strujiStruja.fail())
    cerr << "Nekaj ne valja s upisom" << endl;
else {
    cout << setw(15) << "novo stanje:"
         << setw(8) << novoStanje << endl << endl
         << "Novo stanje je s uspjehom pohranjeno" << endl;
}
//...
```

Prilikom inicijalizacije objekta `strujiStruja` klase `fstream` kao prvi argument navodimo ime datoteke. Drugi argument konstruktoru jest mod: definiramo `in` i `out` (uèitavanje i upisivanje), uz eksplicitnu zabranu automatskog generiranja nove datoteke ako datoteka s navedenim imenom ne postoji (mod `noreplace`). Ovo nam treba zato da bi korisnika upozorili da datoteka ne postoji (možda postoji, ali u drugom imeniku), te da bismo inicijalizirali vrijednost varijable `staroStanje`, koju inaèe uèitavamo iz datoteke, na nulu. Ako datoteka ne postoji, nakon ispisa poruke se otvara nova datoteka naredbom

```
strujiStruja.open("MojaTuga.dat", ios::out);
```

Ako datoteka postoji, èita se njen sadržaj, sve do kraja datoteke. Zadnji podatak u datoteci je posljednje upisano stanje. Kraj datoteke provjerava se preko pokazivaèa koji vraæa `strujiStruja` – sve dok je uèitavanje u redu, taj je pokazivaè razlièit od nule. Nailaskom na kraj datoteke postavlja se `eofbit` pa pokazivaè postaje jednak nuli. Uvjet za ponavljanje petlje uèitavanja mogli smo napisati i kao:

```
do {
    //...
} while(!strujiStruja.eof());
```

Po završenom unosu podataka i raèunu, vrijednost varijable `novoStanje` se pohranjuje na kraj datoteke. Uoèimo naredbu

```
strujiStruja.clear();
```

prije upisa novog podatka. Da nema te naredbe, podatak se ne bi mogao upisati! Naime, prilikom uèitavanja, nailazak na kraj datoteke postavio je `eofbit` – sve dok se to stanje ne obriše, upis u datoteku se ne može obaviti.

Ovako napisani program ima jedan očiti nedostatak: pri svakom pohranjivanju novog stanja datoteka se povećava i u njoj su pohranjeni podaci koji korisniku gotovo sigurno neće više nikad trebati. Elegantnije rješenje je da se stari podatak jednostavno prepiše novim podatkom, ali da bismo to mogli napraviti, moramo se nakon učitavanja starog stanja vratiti na početak datoteke i tamo upisati novo stanje. Kako to ostvariti bit će prikazano u sljedećem odjeljku.

18.7.4. Odreðivanje i postavljanje položaja unutar datoteke

Prilikom pisanja i èitanja datoteke, poseban *pokazivaè datoteke* (engl. *file pointer*) pokazuje na poziciju u datoteci na kojoj se operacija obavlja. U dosadašnjim primjerima podaci su se uzastopno upisivali u datoteke ili iz njih uzastopno uèitavali. Pritom se pokazivaè datoteke automatski uveæava za broj upisanih/uèitanih bajtova. Eksplicitnim usmjeravanjem tog pokazivaèa na željeno mjesto moguæe je èitati podatke ili pisati ih na proizvoljnom mjestu unutar datoteke.

Pokazivač datoteke je potrebno razlikovati od običnih C++ pokazivača – on je jednostavno cijeli broj koji pokazuje na koliko se bajtova od početka datoteke upis obavlja. Tip pokazivača je definiran tipom `streampos`. Taj tip je definiran pomoću ključne `typedef` u datoteci zaglavlja `fstream.h` te je njegova definicija implementacijski zavisna. U većini implementacija taj tip je zapravo `long int`.

U klasi `ostream` definirana su dva funkcijska člana kojima se dohvaća pokazivač datoteke, te time kontrolira položaja unutar izlaznog toka:

1. `tellp()` koji kao rezultat vraæa trenutnu vrijednost pokazivaèa datoteke,
2. `seekp()` kojim se pomièe na zadani položaj unutar datoteke.

Slièno su u klasi `istream` definirana dva funkcijska èlana koji omoguæavaju kontrolu položaja u ulaznom toku:

1. `tellg()` koji kao rezultat vraæa trenutnu vrijednost pokazivaèa datoteke,
2. `seekg()` kojim se pomièe na zadani položaj unutar datoteke.

Zbog sliènosti imena, u poèetku æe korisnik teško razlikovati èlanove klase `ostream` od èlanova klase `istream`. Radi lakšeg pamæenja, možemo shvatiti da zadnja slova u

nazivima èlanova (`p`, odnosno `g`) dolaze od naziva funkcijskih èlanova `put()` i `get()` u pripadajuæim klasama.

Funkcijski članovi `seekp()` i `seekg()` definirani su u po dvije preopterećene inačice:

```
ostream &ostream::seekp(streampos &pozicija);
istream &istream::seekg(streampos &pozicija);

ostream &ostream::seekp(streamoff &pomak, ios::seekdir smjer);
istream &istream::seekg(streamoff &pomak, ios::seekdir smjer);
```

Prve dvije inaèice funkcijskih èlanova `seekp()`, odnosno `seekg()`, prihvaæaju kao argument apsolutnu poziciju na koju se valja postaviti unutar datoteke. Kao stvarni argument najèešæe se prenosi vrijednost dohvaæena funkcijama `tellp()`, odnosno `tellg()`.

Druge dvije verzije kao prvi argument prihvaćaju pomak u odnosu na poziciju definiranu drugim argumentom. Tip `streamoff` je definiran u zaglavlju `fstream.h` i služi za specifikaciju pomaka pokazivača. `seekdir` je pobrojenje koje označava u odnosu na koje mjesto se pomak računa, te ima tri moguće vrijednosti:

- `beg` – pomak u odnosu na poèetak datoteke,
- `cur` – pomak u odnosu na trenutaènu poziciju u datoteci, te
- `end` – pomak u odnosu na kraj datoteke.

Èitatelju æe znaèenje gornjih naredbi biti zasigurno jasnije nakon sljedeæeg primjera:

```
ifstream ulTok("slova.dat");
// pomiče pokazivač datoteke na jedno mjesto prije kraja
ulTok.seekg(-1, ios::end);
// pohrani trenutni položaj pokazivača
pos_type pozic = ulTok.tellg();
char zn;
ulTok >> zn;
cout << zn;
// pomiče pokazivač na apsolutni početak datoteke
ulTok.seekg(0);
ulTok >> zn;
cout << zn;
// vraća pokazivač za jedno mjesto prema početku
ulTok.seekg(-1, ios::cur);
ulTok >> zn;
cout << zn;
// vraća pokazivač na jedno mjesto ispred zapamćenog položaja
ulTok.seekg(pozic - 1);
ulTok >> zn;
cout << zn << endl;
```

Ako datoteka `slova.dat` sadrži samo niz `"abcdefgh"`, tada æe izvoðenje gornjih naredbi na zaslonu ispisati:

```
haag
```

Zadatak. *Napišite program koji kopira sadržaj datoteke, znak po znak, od njena kraja prema početku, u novu datoteku.*

18.7.5. Binarni zapis i uèitavanje

Brojevi se u memoriji raèunala pohranjaju u binarnom formatu. Meðutim, tekstovni je format primjereniji ljudima, tako da se na zaslonu brojevi ispisuju u tekstovnom formatu (osim kada brojeve ispisujemo u heksadekadskom ili oktalnom obliku). Isto vrijedi i za unos podataka pomoæu tipkovnice. Izravna posljedica takve "dvoliènosti" jest neophodna pretvorba brojeva iz tekstovnog oblika u binarni oblik prilikom unosa, te obrnuta pretvorba prilikom ispisa.

Iako se podaci u datoteke mogu pohranjivati u tekstovnom obliku i kao takvi čitati (kao što je u dosadašnjim primjerima rađeno), ne postoji suštinski jači argument da se brojevi moraju pohranjivati baš u tom obliku. Jedini argument za tekstovno pohranjivanje jest taj da se tako pohranjeni brojevi mogu pročitati bilo kojim programom za obradu teksta. Međutim, binarno pohranjeni brojevi se brže učitavaju iz datoteke u memoriju računala i upisuju u datoteku, jer nema nikakve pretvorbe – podaci se jednostavno preslikavaju bajt po bajt.

Uz veću brzinu, binarno pohranjivanje brojeva gotovo uvijek rezultira manjom duljinom zapisa, kao posljedica efikasnijeg korištenja znakova u binarnom formatu. U tekstovnom formatu se, primjerice, broj 1066 pamti točno tako, kao niz od četiri znamenke (`'1', '0', '6', '6'`) pri čemu se za svaku znamenku potroši jedan bajt (pamti se kôd znaka). Naprotiv, taj broj se može pohraniti u binarnom formatu kao niz od samo dva bajta: niži bajt je 42, a viši bajt je 4. Razlika u korist binarnog zapisa je više nego očita! Za znakove i znakovne nizove nema razlike između binarnog i tekstovnog zapisa, jer se znakovi uvijek pohranjuju kao (najčešće) ASCII kôdovi pripadajućih znakova.

Binarni zapis i čitanje datoteke pokazat ćemo na sljedećem primjeru. Zamislimo da imamo poduzeće koje se bavi proizvodnjom elektroničkih mišolovki. Želimo napraviti bazu podataka s imenima kupaca kojima periodički (svaka dva mjeseca, a zimi čak jednom mjesečno) šaljemo nove baterije i barutno punjenje. Kako poduzeće posluje vrlo uspješno u uvjetima tržišnog gospodarstva (Jeste li gledali "Izbavitelja"?), imat ćemo vrlo dugački listu, pa ako je želimo pregledavati, pogodno je imati ju sortiranu po imenima kupaca. Kako osim ingenioznosti na području suvremenog uklanjanja glodavaca posjedujemo i novostečeno znanje iz jezika C++ i nesumnjive sklonosti pisanju programa s tokovima (psiholozi će im tek posvetiti mnoge knjige), napisat ćemo kratak program kojim ćemo unositi imena pomoću tipkovnice i upisivati ih u datoteku. Pri tome će se podaci u datoteci pamtiti u vezanoj listi: ispred svakog naziva bit će zapamćen datotečni pokazivač na sljedeći naziv. Prilikom upisa, novododani podatak će

biti smješten na kraj datoteke, ali ćemo pronaći mjesto u listi na koje se podatak dodaje, te ćemo ažurirati pokazivače. Na taj način izbjegavamo učitavanje kompletne datoteke u memoriju, njeno sortiranje i zapisivanje – upis će biti brži. Štoviše, za veliku bazu bi nam sigurno ponestalo memorije za učitavanje cijele baze, pa operaciju niti ne bismo mogli obaviti.

Prvi podatak u datoteci bit će glava liste – pokazivač na poziciju na koju je zapisan prvi član liste. Ako je lista prazna, pokazivač će sadržavati nulu. Svaki se član liste sastoji od pokazivača na položaj sljedećeg člana te od samog sadržaja člana. Član koji je zadnji u listi ima pokazivač jednak nuli. U donjem kôdu je riješeno umetanje novog člana; čitatelju prepuštamo da riješi problem brisanja člana.

```
const int duljImena = 50;   // najveća dozvoljena duljina

pos_type pozSljed = 0;
fstream bazaImena("imenabin.dat", ios::in | ios::out |
                  ios::nocreate | ios::binary);
if (!bazaImena) {
    bazaImena.open("imenabin.dat", ios::in | ios::out |
                   ios::binary);
    bazaImena.write((char *) &pozSljed, sizeof(pos_type));
}

char novoIme[duljImena];
cin.getline(novoIme, duljImena);
// ponavlja unos novih imena sve dok je duljina utipkanog
// imena različita od nule
while (cin.gcount() > 1) {
    pos_type pozPreth = 0;
    pos_type pozTemp = 0;
    // na početak datoteke
    bazaImena.seekg(0);
    // učita poziciju sljedećeg
    bazaImena.read((char *) &pozSljed, sizeof(pos_type));
    // prolazi datoteku sve dok postoji sljedeći
    while (pozSljed) {
        // pomakne na sljedeći zapis
        bazaImena.seekg(pozSljed);
        char imeNaDisku[duljImena];
        // učita pokazivač na sljedeći zapis
        bazaImena.read((char *) &pozTemp, sizeof(pos_type));
        // učita ime u tekućem zapisu
        bazaImena.get(imeNaDisku, duljImena, '\0');
        if (strcmp(imeNaDisku, novoIme) >= 1)
            break;      // ako novo treba doći ispred, prekida
        pozPreth = pozSljed;
        pozSljed = pozTemp;
    }
    // prvo dopisuje novo ime na kraj datoteke
    bazaImena.seekg(0, ios::end);
    pozTemp = bazaImena.tellg();
```

```
        bazaImena.write((char *) &pozSljed, sizeof(pos_type));
        bazaImena.write(novoIme, strlen(novoIme) + 1);
        // postavlja se na prethodni zapis u nizu i usmjerava ga
        // na novododani zapis
        bazaImena.seekp(pozPreth);
        bazaImena.write((char *) &pozTemp, sizeof(pos_type));
        // unos novog imena
        cin.getline(novoIme, sizeof(novoIme));
    }
    cout << "Upisi su završeni." << endl << endl;
    // slijedi kôd za ispis liste...
```

Inicijalizacija toka je identièna kao u primjeru sa strane 546, osim što dodatno navodimo binarni mod. Ako datoteka ne postoji, tada se ona generira i upisuje se glava liste kao nula. Zatim zapoèinje petlja za unos novih podataka.

Podaci se unose u zasebnim recima – pritiskom na tipku za unos izvodi se funkcija `getline()`, koja očitava utipkani redak teksta; ako se tipka za unos pritisne na samom početku retka, petlja će se prekinuti. Zatim se provjerava je li neki podatak uopće unesen. To se može obaviti pomoću funkcije `gcount()` – ona vraća broj učitanih znakova u prethodnoj `get()`, `getline()` ili `read()` funkciji. Budući da `getline()` učitava i znak za novi redak, u slučaju da nije utipkan nikakav tekst, funkcija `gcount()` će vratiti broj 1 te je to signal da korisnik nije unio podatak.

Uočimo način na koji se binarno pohranjuju i učitavaju pokazivači datoteke. Za upis pokazivača korišten je funkcijski član `write()`. On kao prvi argument prihvaća pokazivač `char *` koji pokazuje na niz bajtova koje želimo upisati, a drugi argument je broj bajtova koje treba upisati. Budući da pokazivače želimo upisati binarno, bajt po bajt, pokazivač operatorom dodjele tipa `(char *)` pretvaramo u niz bajtova i kao takav ga zapisujemo. Za upis teksta također koristimo član `write()`, jer on omogućava da jednostavno zapišemo i zaključni nul-znak.

Za usporedbu znakovnih nizova korištena je funkcija `strcmp()`, deklarirana u zaglavlju `string.h`. Navedimo još kako bi izgledao kôd za ispis članova liste:

```
    // ispis članova liste po abecedi
    cout << "Dosad upisana imena:" << endl;
    bazaImena.seekg(0);
    bazaImena.read((char *) &pozSljed, sizeof(pos_type));
    while (pozSljed) {
        bazaImena.seekg(pozSljed);
        bazaImena.read((char *) &pozSljed, sizeof(pos_type));
        bazaImena.get(novoIme, duljImena, '\0');
        cout << novoIme << endl;
    }
    cout << "*** Konac popisa ***" << endl;
```

Za èitanje binarno zapisanog pokazivaèa datoteke koristi se funkcijski èlan `read()`. On je komplementaran èlanu `write()`: prvi argument mu je pokazivaè `char *` na niz

bajtova kamo treba uèitane podatke pohraniti, a drugi argument je broj bajtova koje treba uèitati. Tekstovni niz se uèitava funkcijskim èlanom `get()` kojemu je kao graniènik, kod kojeg prekida uèitavanje, naveden nul-znak.

U svakom slučaju valja uočiti da kod binarno zapisanih podataka, prilikom učitavanja treba unaprijed znati kakvog je tipa podatak koji se učitava. U binarnom obliku svi podaci izgledaju jednako, pa program neće razlikovati učitava li se broj ili znakovni niz.

Zadatak. *Razmislite i pokušajte riješiti gornji zadatak tako da pokazivače zapisujete u tekstovnom obliku. Usporedite duljine datoteka sa deset jednakih nizova podataka za obje izvedbe programa.*

Zadatak. *Napišite program koji pohranjuje i učitava polje cijelih brojeva u binarnom obliku. Duljinu polja (također u binarnom obliku) pohranite kao prvi podatak.*

18.8. Tokovi vezani na znakovne nizove

U prethodnim odjeljcima smo vidjeli kako se tok može vezati na datoteku. Podaci su se prilikom zapisivanja pohranjivali u datoteku kao niz znakova. Prilikom èitanja, taj se niz znakova uèitava sa vanjske jedinice na koju je datoteka pohranjena. Isto tako se tok može vezati i na znakovni niz pohranjen u memoriji raèunala – datoteka na disku je zapravo niz bajtova te se može promatrati kao znakovni niz. Za tokove nema razlike èitaju li se podaci iz datoteke ili znakovnog niza, jer tok zapravo barata s apstraktnim spremnikom podataka u kojemu su podaci organizirani u nizove bajtova (od tuda i dolazi naziv *tok*).

Tokovi vezani na znakovne nizove deklarirani su u zaglavlju `sstream`. Na raspolaganju su nam dvije klase: `istringstream` (naslijeđena od klase `istream`) za čitanje toka te `ostringstream` (naslijeđena od klase `ostream`) za upis na tok. U starijim implementacijama zaglavlje se zvalo `strstream.h`, a klase su se zvale `istrstream`, odnosno `ostrstream`. Zbog kompatibilnosti s postojećim implementacijama, standard predviđa podršku i za te klase.

Korištenje tokova vezanih na znakovne nizove ilustrirat ćemo na primjeru programa koji učitava sadržaj datoteke `text.txt` u znakovni niz [Lippman91]. Program smo realizirali pomoću zaglavlja `strstream.h`, budući da je prevoditelj kojeg smo mi koristili imao samo to zaglavlje u biblioteci:

```
ifstream ulDatoteka("tekst.txt", ios::in);
ostrstream znTok;
char znak;
while (znTok && ulDatoteka.get(znak))
    znTok.put(znak);
char *sadrzaj = znTok.str();
```

Za uèitavanje sadržaja datoteke inicijaliziramo ulazni tok `ulDatoteka` te uèitavamo znakove iz datoteke `tekst.txt` sve do njenog kraja. Uèitane znakove upisujemo u tok

znTok klase ostrstream. Upisivanje u tok se obavlja pomoæu funkcijskog èlana put(). Taj funkcijski èlan ne samo da zapisuje znakove u memoriju, nego istovremeno poveæava memorijski prostor neophodan za niz koji trebamo pohraniti.

Nakon što su učitani svi znakovi, poziva se funkcijski član str() koji vraća pokazivač na niza znakova pridruženih objektu klase ostrstream. Tim pozivom se pokazivač sadrzaj preusmjerava na niz učitanih znakova pa se učitani znakovi mogu obrađivati kao običan znakovni niz (valja jedino napomenuti da taj niz ne mora biti zaključen nul-znakom – to ovisi o sadržaju datoteke).

Osim što vraća pokazivač na toku pridruženi znakovni niz, funkcijski član str() ujedno "zamrzava" sadržaj toka. On jednostavno rečeno, niz učitanih znakova preuzima od ostrstream objekta i prosljeđuje ga okolnom programu. Time svako daljnje upisivanje podataka postaje onemogućeno. Zbog toga član str() treba pozvati nakon što su u niz upisani svi podaci.

Izlaskom iz bloka u kojem je objekt klase ostrstream bio inicijaliziran automatski se uništava cijeli objekt, zajedno sa sadržajem upisanog niza. Međutim, nakon poziva člana str(), taj niz više nije u nadležnosti ostrstream objekta, tako da ga programer mora eksplicitno uništiti. Primjerice, za gornji slučaj to bi se postiglo naredbom:

```
delete sadrzaj;
```

Kako su klase ostrstream i istrstream naslijeðene iz klasa ostream, odnosno istrstream, dostupni su i svi njihovi funkcijski èlanovi. To primjerice znaèi da se po znakovnom nizu možemo "šetati" pomoæu funkcijskih èlanova seekg() i seekp(), kao da se radi o datoteci na disku. Takoðer automatski vrijede operatori umetanja << i izluèivanja >>.

18.9. Ulijeva li se svaki tok u more?

Napomenimo na kraju da u ovom poglavlju nisu spomenuti svi èlanovi opisanih klasa – iznijeli smo samo najinteresantnije èlanove. Osim toga, ispis na zaslon i uèitavanje s tipkovnice ulazno-izlaznim tokovima (onakvima kako su definirani standardom) moguæe je samo pod operacijskim sustavima s tekstovnim suèeljem, kao što su UNIX ili DOS. Operacijski sustavi s grafièkim suèeljem (MS Windows, X-Windows) iziskuju posebne funkcije koje se isporuèuju u bibliotekama s prevoditeljem.

No znanje stečeno u ovom poglavlju je dosta važno i za (X) Windows programere. Naime, mnoge komercijalno dostupne biblioteke (na primjer Microsoftova MFC biblioteka) podržavaju operacije koje vrlo nalikuju tokovima, a obavljaju poslove pohranjivanja objekata biblioteke na disk ili slične operacije. Zbog svoje jednostavnosti i intuitivnosti, model tokova je preuzet za različite ulazno/izlazne poslove te je dosta važno razumjeti ga.

19. Principi objektno orijentiranog dizajna

Ja slikam objekte kako ih mislim,
a ne kako ih vidim.

Pablo Picasso (1881-1973)
U knjizi J. Golding: "Cubism" (1959)

Objektno orijentirano programiranje nije samo novi naèin zapisivanja programa – ono iziskuje potpuno novu koncepciju razmišljanja. Stari proceduralni programi se, doduše, mogu prepisati u objektnom jeziku, no time se ne iskorištavaju puni potencijal jezika.

C++ nije zamišljen zato da bi se stari programi bolje preveli, ili da bismo dobili brži izvedbeni kôd. Njegova osnovna namjena je pomoći programeru da formulira svoje apstraktne ideje na način razumljiv računalu. Njegova uporabna vrijednost posebice dolazi do izražaja pri razvoju novih, složenih programa.

19.1. Zašto uopæe C++?

Ako ste izdržali èitajuæi knjigu do ovog poglavlja, onda ste se upoznali sa svim svojstvima objektno orijentiranog programiranja koje nudi jezik C++. Nauèili ste definirati klase, specificirati funkcijske èlanove, preoptereæivati operatore i još štošta drugo. No ako ste novi brodolomac u vodama objektno orijentiranog programiranja, pa èak ako ste veæ prošli objektno orijentirano vatreno krštenje i izdigli se iznad statusa žutokljunca (engl. *green-horn*), vrlo je vjerojatno da ste si prilikom èitanja prethodnih poglavlja postavili pitanje: "A što je to uopæe objekt? I zašto da se uopæe patim s virtualnim funkcijskim èlanovima kada se to sve može napraviti i u obiènom C-u, ili èak u Pascalu?"

Odgovor na to pitanje nije jednostavan. Napisano je mnogo knjiga o objektno orijentiranom pristupu u kojima su razmatrani teoretski aspekti OOP-a (engl. *Object-Oriented Programming*). Istina je da se sve što se napiše u C++ jeziku može realizirati i u običnom, "bezobjektnom" C ekvivalentu. Uostalom, na kraju balade, kompletan kôd se prevodi u strojni jezik, pa bismo isto mogli postići pišući direktno strojne instrukcije.

Općeniti odgovor na gornje pitanje može se sažeti u dva osnovna pojma: objektno programiranje podržava *apstrakciju podataka* (engl. *data abstraction*) i bolju *ponovnu iskoristivost kôda* (engl. *code reusability*). Ti pojmovi nisu vezani isključivo za C++ jezik, već i za preostale objektno orijentirane jezike, kao što su SmallTalk, Actor i drugi. Tim više, C++ podržava višestruki pristup programiranju: objekte možemo koristiti samo za neke segmente programa ili ih čak uopće ne moramo koristiti. Dapače, moguće

je napisati C++ program koji ne deklarira niti jedan objekt te dosljedno prati proceduralni način pisanja programa. No ipak, s pojavom novih operacijskih sustava koji teže objektnom ustroju, objektno programiranje se jednostavno ne smije zanemariti.

Jezik C++ je jezik opće namjene, za poslove od sistemskog programiranja do razvoja korisničkih sučelja te čisto apstraktnih primjena kao što su matematički proračuni i predstavlja najkorišteniji programski jezik za pisanje komercijalnih aplikacija. To je mjesto stekao dobrim dijelom zahvaljujući svom prethodniku – jeziku C. Svi C programi se mogu prevesti C++ prevoditeljem potpuno ispravno, te se na već postojeće projekte lako može dograditi objektno orijentirani modul. I dok je očuvanje kompatibilnosti s C-om sigurno jedna od velikih prednosti C++ jezika, to mu je istodobno i jedan veliki kamen smutnje. Naime, mnogi C-ovski koncepti, kao što su pokazivači, nizovi te dodjele tipova ugrađeni su u C++ te dozvoljavaju zaobilaženje objektnih svojstava. Tako se C++ nalazi između klasičnih proceduralnih jezika, kao što su C i Pascal, i čistih objektnih jezika, kao što je SmallTalk te objedinjava svojstva i jednih i drugih. Takva dvostruka svojstva ponekad dodaju jeziku nepotrebnu kompliciranost.

U prethodnim poglavljima iznesena su svojstva jezika s naznakama kako se ta svojstva mogu iskoristiti u objektno orijentiranim programima. No do istinskog poznavanja objektno orijentiranog principa programiranja potrebno je prevaliti još dug put. Naime, ako programer poznaje ključnu riječ `virtual`, ne znači da će ju ujedno znati i upotrijebiti na pravom mjestu na pravi način, po principu “Stavi pravi `virtual` na pravo mjesto... (učini to često)”. Dapače, ima dosta primjera nenamjerne “zloupotrebe” slobode koju pruža OOP. Rezultat je loš izvorni kôd koji ima dosta pogrešaka, nečitak je, sporo se izvodi te se dosta teško prilagođava promijenjenim zahtjevima.

Zbog toga ćemo u ovom poglavlju pokušati prikazati način na koji se elementi jezika upoznati u prethodnim poglavljima mogu učinkovito iskoristiti. Ovo poglavlje neće uvesti niti jednu novu ključnu riječ niti proširiti stečeni C++ vokabular. No ispravna i dosljedna primjena principa programiranja je čak važnija od poznavanja sintakse i semantike jezika.

Ovdje će se spomenuti samo osnovni principi modeliranja objekata – na kraju, u području kao što je C++ ne može se očekivati da svo znanje bude kondenzirano na jednom mjestu. Naposljetku, iskustvo je vrlo značajna dimenzija objektnog programiranja koja se stiče isključivo radom. Neke stvari, ma koliko puta ih pročitali, shvatit ćete tek nakon tri dana i noći provedenih analizirajući program koji bez pardona ruši sve teorije o pisanju programa bez pogrešaka.

19.2. Objektna paradigma

U starijim programskim jezicima, kao što su poèetne verzije FORTRAN-a, programeri nisu imali baš velike moguænosti modeliranja podataka. Na raspolaganju su im bile varijable koje su èuvale najèešæe samo jednu brojèanu vrijednost i polja koja su mogla èuvati nekoliko istovrsnih podataka. Oni sretniji su raspolagali prevoditeljem koji je podržavao znakovne nizove. Takvo što se smatralo veæ rasipništvom – raèunala su

imala vrlo usko podruèje primjene, primarno za numerièke proraèune. Takoðer, programi pisani u tim jezicima bili su bez ikakve strukture te se programer morao brinuti o svim aspektima rada raèunala, gotovo kao da je sâm program pisan u strojnom jeziku.

No jezici Pascal i C unijeli su značajnu novost u programiranje. Umjesto programera, mnoge poslove organizacije podataka u tim jezicima obavlja prevoditelj. To je postignuto strukturiranjem programa. Cijeli monolitni kôd se razbija u više zasebnih funkcionalnih cjelina, koje se zatim međusobno povezuju. Uvedene su lokalne varijable koje postoje samo u nekom segmentu programa, te se vanjski program o njima ne mora brinuti.

U tim jezicima po prvi put se javlja mogućnost strukturiranja podataka. U Pascalu se to čini ključnom riječi `record`, a u C-u riječi `struct`. Takve strukture mogu sadržavati nekoliko podataka različitih tipova koji zajedno čine smislenu cjelinu. Program koji je koristio takve strukture u cijelosti je poznavao svaki član strukture te je pristupao svakom članu strukture direktno. Opisani pristup je u mnogome pojednostavnio način pisanja složenih programa. Programer nije promatrao sve elemente strukture nezavisno, već povezano u logičku cjelinu. Programiranje na taj način je proceduralno, no podaci su skupljeni u agregate.

Od agregata do objekata je vrlo malen korak. Naime, kada smo već povezali srodne podatke i omotali ih celofanom, zar ne bi bilo prikladno odmah definirati i dozvoljene operacije nad njima te na taj način dati smisao podacima i tako na njih staviti *mašnicu*? Za svaki tip podataka s kojim radimo postoji određeni smisleni skup operacija, dok neke druge operacije za taj tip jednostavno nemaju smisla. Na primjer, nema smisla množiti dva znakovna niza, ali ih ima smisla ispisivati ili zbrajati. Zbog toga, prije nego što se uputimo u izradu tipa podataka koji će prikazivati znakovni niz, prvo ćemo se upitati što zapravo s nizom želimo učiniti. Ako dobro shvatimo čemu nam neki podatak služi, bit će jednostavnije napisati adekvatnu implementaciju sa svim željenim svojstvima. Podaci i radnje nad njima se više neće tretirati odvojeno i neovisno jedni od drugih. Takav model podataka se ponekad naziva i model *apstraktnih tipova podataka* (engl. *abstract data types*).

Ključna riječ u gornjem terminu je *apstrakcija* podataka. To je postupak kojim se unutrašnja složenost objekta skriva od okolnog svijeta. To je i logično: vanjski program ne mora znati kako je znakovni niz interno predstavljen – njega interesira isključivo sadržaj. Vanjskom programu su interesantna svojstva objekta te akcije koje on može učiniti, a ne način na koji će se to provesti. Interna reprezentacija objekta je implementacijski detalj koji je skriven od vanjskog svijeta. On se može i promijeniti ako se pokaže da postojeća implementacija nije dobra, no važno je da objekt u smislu svojih svojstava ostane nepromijenjen. Okolni programi pristupaju objektu preko njegovog javnog sučelja koje definira izgled objekta prema okolini.

Ovakav postupak prikrivanja unutarnje strukture objekta naziva se *skrivanje podataka* (engl. *data hiding*). Ono ima značaj koji debelo prelazi granice objektno orijentiranog programiranja. Složeni računarski sustavi se danas modeliraju upravo po tom načelu. Tako na primjer operacijski sustav Windows skriva svoje strukture u kojima se čuvaju podaci o stanju računala od prosječnog korisnika. Korisnički program pristupa

pojedinim uslugama sustava preko definiranog programerskog sučelja, često označenog skraćenicom *API* (engl. *application programming interface*).

Jezik C++ omogućava primjenu koncepcije skrivanja informacija pomoću klasa – tipova podataka koji sadržavaju i podatke i funkcije. Pomoću dodjele prava pristupa pojedinim segmentima objekta, moguće je razgraničiti javno sučelje od implementacije.

19.3. Ponovna iskoristivost

Drugi pojam vezan za objektno programiranje je *ponovna iskoristivost* (engl. *reusability*) kôda. To je svojstvo koje omoguæava da se jednom napravljeni dijelovi programa ponovo iskoriste u nekom drugom projektu uz minimalne izmjene. Time se može znatno skratiti vrijeme potrebno za razvoj novih programa.

“No što je tu čudno?”, upitat ćete se: “Zar je takvim bombastičnim i kompliciranim nazivom potrebno označavati jednostavno kopiranje segmenata kôda? Naposljetku, u običnim proceduralnim jezicima moguće je stvarati biblioteke funkcija koje se mogu pozivati iz bilo kojeg drugog programa.”

Gornja konstatacija je točna što se tiče stvaranja običnih biblioteka funkcija, ali pod pojmom ponovne iskoristivosti se danas podrazumijeva znatno više. Naime, kako smo već zaključili da je pristup programiranju pomoću apstraktnih podataka znatno jednostavniji i manje sklon pogreškama, ponovna iskoristivost se ne može ograničiti isključivo na stvaranje biblioteke funkcija. Umjesto toga, potrebno je stvoriti biblioteku objekata koji se zatim koriste prema potrebi.

No niti to nije osnovni problem ponovne iskoristivosti. Ponovna iskoristivost znači da se jednom napisani kôd unutar istog programa koristi više puta ili čak na više načina. Na primjer, česte su potrebe za definiranje lista različitih objekata. U tu svrhu može se definirati objekt `Lista` koji će definirati javno sučelje (operacije `Dodaj()`, `Ukloni()`, `Trazi()` i slične) i mehanizam liste. Sam objekt koji će se čuvati u listi specificirat će se naknadno. Također, ako je mehanizam liste potrebno proširiti novim svojstvima, kao primjerice brisanje cijele liste, neće biti potrebno pisati cijeli kôd iznova nego će se jednostavno dodati samo željena promjena. Osnovni mehanizmi koji to omogućavaju su nasljeđivanje i polimorfizam. Time se model programiranja pomoću apstraktnih tipova podataka izdiže na nivo pravog objektnog programiranja.

Nasljeđivanje je postupak kojim se iz postojeće klase izvodi nova klasa u kojoj se navode samo promjene u odnosu na osnovnu klasu. Polimorfizam je svojstvo koje omogućava rukovanje podacima čiji tip prilikom pisanja programa nije poznat. To je zapravo samo još daljnji korak u procesu enkapsulacije: tip objekta je upakiran sa samim objektom te se koristi prilikom izvođenja kako bi se odredilo ponašanje objekta.

Jezik C++ podržava sva tri pristupa programiranju: proceduralno programiranje prošireno agregatima podataka, programiranje pomoću apstraktnih tipova te pravo objektno orijentirano programiranje. Na korisniku je da izabere pristup najprikladniji problemu koji se rješava. U nastavku će se razmotriti važniji koraci u pisanju objektno orijentiranih programa.

Principe objektno orijentiranog dizajna prikazat ćemo na primjeru izrade biblioteke koja će opisivati interaktivno korisničko sučelje. Ona će sadržavati objekte koji će omogućavati prikazivanje prozora na ekranu računala. Prozori će se moći pomicati pomoću miša. Također će biti moguće svakom prozoru dodijeliti izbornik. Taj primjer je izabran zato jer su relacije između elemenata korisničkog sučelja (prozori, izbornici) i objekata koji modeliraju te elemente vrlo očite i razumljive.

Cilj ovog poglavlja nije ispisivanje izvornog kôda biblioteke koja će stvarno omogućavati otvaranje prozora. To i nije moguće, jer je za to potrebno mnogo podataka o stvarnom računalu za koje je biblioteka namijenjena. Također, iluzorno je za očekivati da je moguće izraditi kvalitetnu biblioteku te njenu strukturu prikazati na desetak stranica koje sačinjavaju ovo poglavlje. Ono što je moguće, jest prikazati osnovnu ideju bez specificiranja kôda koji stvarno crta prozor ili izbornik. Biblioteka neće biti vezana niti za jedno računalo, pa čak neće specificirati da li se koristi u grafičkom ili tekstovnom načinu.

19.4. Korak 1: Pronalaženje odgovarajuæe apstrakcije

Da bismo principe objektnog programiranja pravilno primijenili u C++ jeziku, potrebno je prvo pronaæi odgovarajuæi model za opisivanje naših podataka. Aplikacija koju izraðujemo sastoji se od niza razlièito organiziranih struktura podataka. Te podatke je potrebno prikazati objektnom apstrakcijom. Da bismo to uèinili, neophodno je detaljno istražiti koje su to apstrakcije i koja su njihova svojstva. Prije nego što poènemo pisati klase, potrebno je ustanoviti s kojim æemo klasama uopæe raditi.

Dobra je praksa najprije ispisati na komad papira što sažetije tekst zadatka kojeg smo si postavili te ga zatim nekoliko puta pročitati i razmotriti. U slučaju korisničkog sučelja zadatak bi mogao biti ovako zadan:

Potrebno je napisati biblioteku u C++ jeziku koja će omogućavati prikazivanje prozora i izbornika na ekranu računala.

Iako je ovo vrlo površan opis samog problema, dovoljno je intuitivan za sve koji su ikad imali ikakav kontakt s grafièkim korisnièkim suèeljima, kao što su primjerice MS Windows ili X-Windows. Iz gornjeg opisa potrebno je odrediti objekte koji su kandidati za apstrakciju. Dobro je ispisati sve imenice iz teksta zadatka kako bi se pronašli kandidati za apstrakciju:

- *biblioteka*
- *jezik*
- *prozor*
- *izbornik*
- *ekran*
- *računalo*

Potrebno je razmotriti listu i sa stanovišta zadatka ocijeniti koji su objekti pogodni za apstrakciju pomoæu klase prilikom rješavanja problema.

Iako *biblioteka* ima veze s rješavanjem problema, ona je objekt koji će objedinjavati sve preostale objekte. Nije potrebno uvoditi posebnu apstrakciju za biblioteku, jer će sam izvedbeni kôd dobiven nakon prevođenja predstavljati biblioteku.

Jezik nema direktne veze s rješavanjem problema. On je samo sredstvo koje koristimo za formulaciju rješenja, a ne dio rješenja.

Prozor, *izbornik* i *ekran* su objekti koji su direktno pogodni za prikazivanje pomoću klase. Na primjer, ekran je fizički dio računala koji omogućava prikaz podataka u vidljivom obliku. Njegova svojstva, kao na primjer razlučivost u horizontalnom i vertikalnom smjeru, broj boja koji se odjednom može prikazati, frekvencija osvježavanja i slično, mogu se vrlo efikasno upakirati u objekt klase `Ekran` koji će biti apstrakcija stvarnog ekrana.

Prozor i *izbornik* su elementi korisničkog sučelja koji su već sami po sebi apstrakcije. Oni ne postoje u stvarnom svijetu, već su dio apstrakcije u komunikaciji između računala i korisnika.

Iako se možda na prvi pogled čini da je *računalo* objekt koji nema veze s rješenjem problema, pažljivo razmatranje nam može reći da to nije tako. Računalo je objekt koji se sastoji od mnogo različitih cjelina: tipkovnice, memorije, miša, ekrana, diskova i slično. Neki od njih služe za interakciju između programa i korisnika, poput miša, tipkovnice i ekrana. Na sličan način na koji ćemo u klasu `Ekran` upakirati svojstva ekrana u računalu, moguće je definirati i klase `Mis` i `Tipkovnica` koji će predstavljati apstraktni model tih uređaja. Dakle, nije nam potrebno direktno modelirati računalo, no modelirat ćemo neke njegove dijelove. Modeliranje računala koje objedinjuje sve druge objekte za svaki dio računala bio bi posao operacijskog sustava.

Primjenom gornjeg postupka dobili smo već nekoliko klasa koje ćemo primijeniti u rješavanju problema. Time popis nije nipošto zaključen – daljnjom analizom dobivenih objekata vidjet ćemo da ćemo morati uvesti nove klase za nove apstrakcije.

19.5. Korak 2: Definicija apstrakcije

U prethodnom koraku razluèili smo nekoliko objekata koje bismo željeli implementirati pomoæu klasa. No još smo vrlo daleko od samog pisanja kljuène rijeèi `class`. Za sada znamo premalo o pojedinom objektu da bismo ga mogli direktno opisati. Zbog toga æemo nadalje pokušati preciznije definirati svaku apstrakciju. Pri tome je vrlo važno usredotoèiti se na identifikaciju apstrakcije bez razmišljanja o implementaciji.

19.5.1. Definicija ekrana

Napisat æemo kratku reèenicu koja æe dati definiciju ekrana:

> *Ekran je element računala koji koristi katodnu cijev za prikaz slike u razlučivosti 640 točaka horizontalno i 480 točaka vertikalno, s maksimumom prikazivanja od 16 boja istodobno.*

Nakon što smo napisali gornju definiciju, pokušajmo je ponovo proèitati i razmotriti da li ona dobro opisuje željenu apstrakciju.

Već se na prvi pogled može ustanoviti da je gornja definicija zapravo vrlo loša. Umjesto da kaže što ekran radi i s kojim objektima surađuje, ona odmah uvodi niz implementacijskih detalja koji nisu važni prilikom pokušaja opće definicije ekrana. Nadalje, rečeno je da ekran koristi katodnu cijev. To ne samo što ne mora uvijek biti točno (sve se više koriste ekrani s tekućim kristalima ili aktivnim elementima), nego uopće nije važno za naš program. Zbog toga ćemo odbaciti gornju definiciju te pokušati specificirati čemu ekran služi:

Ekran je element računala koji omogućava prikaz promjenjive slike.

Ovo je znatno bolje, ali nije dovoljno specifièno. Sada veæ imamo intuitivnu viziju èemu nam ekran služi, no moramo biti još malo više specifièni kako bismo opisali naèin rada ekrana.

Prikaz slike na ekranu se odvija tako da se slika rastvori u niz točaka koje se postave u horizontalnu i vertikalnu križaljku. Broj točaka u horizontalnom smjeru se naziva horizontalna razlučivost, a broj točaka u vertikalnom smjeru vertikalna razlučivost.

Sada veæ znamo mnogo više: ekran služi za prikaz slike. Slika je nova apstrakcija koja se sastoji od toèaka, a ima i odreðene atribute: razluèivost u horizontalnom i vertikalnom smjeru. Toèka je takoðer nova apstrakcija, a evo i njene definicije:

Točka je osnovni element slike. Njeno glavno svojstvo je boja. Svakoj točki se može neovisno postaviti neka željena boja.

Gornja definicija kaže da toèka zapravo modelira boju na odreðenom dijelu slike. Takoðer, definicija naznaèava da se atribut boje za pojedinu toèku mora moæi postaviti. Iako u definiciji nije navedeno, bilo bi poželjno da se boja može i proèitati. Definicija takoðer kaže da se boja za svaku toèku može postavljati neovisno. Za mnoge vrste ekrana to nije toèno. Naime, zbog svoje sklopovske graðe veæina današnjih video-podsustava omoguæava prikaz samo nekog odreðenog broja boja odjednom (16, 256, i slièno). Gornju definiciju treba promijeniti tako da se to svojstvo ugradi u definiciju toèke:

Svakoj točki se može postaviti boja neovisno sve dok ukupan broj različitih boja svih točaka na ekranu ne premaši ukupan broj boja koje se mogu prikazati na ekranu.

Taj problem veæina stvarnih uređaja za prikaz slike rješava tako da se uvede takozvana paleta boja. To je zapravo jedan popis trenutno aktivnih boja. Svaka boja predstavlja jednu stavku u paleti dok toèka umjesto da direktno specificira boju predstavlja samo indeks u paletu. Na ovom mjestu bi sada bilo potrebno dati definiciju palete te zatim iterirati kroz sve do sada dane definicije. To bi nas vrlo brzo odvelo duboko u detalje hardverske implementacije pojedinih uređaja pa æemo se zato zadovoljiti s ovakvim pojednostavljenim modelom.

Napomenimo na kraju što podrazumijevamo pod ekranom. U užem smislu ekran je zaslon računala ili monitor. No kako ne bismo otišli predaleko i cjepidlačili oko toga da

za prikaz slike treba monitor i grafička kartica, pod nazivom “ekran” podrazumijevat ćemo cjelokupni podsustav računala za prikaz slike.

19.5.2. Definicija prozora

Po gore navedenom principu provest æemo definiciju apstrakcije prozora kako bismo mogli bolje definirati naš objektni model:

> *Prozor je područje slike na ekranu koje prikazuje stanje izvođenja pojedine aplikacije u računalu.*

Gornja definicija, iako možda na prvi pogled najoèitije prikazuje ulogu prozora, nije sasvim precizna. Naime, reæi da je primarna uloga prozora prikazivanje stanja određene aplikacije je vrlo usko određenje. U stvarnosti prozori se koriste za mnogo raznih drugih poslova, kao što je prikazivanje dijaloških prozora ili èak modeliranje pojedinih kontrola unutar prozora. Također, operacijski sustav MS Windows implementira padajuæe izbornike pomoæu prozora. Zbog toga treba promijeniti gornju definiciju:

> *Prozor je pravokutno područje slike na ekranu za prikaz određenog skupa logički povezanih podataka.*

Takva definicija znatno bolje opisuje što je prozor. Pri tome ništa nije reèeno o svrsi prozora; on samo prikazuje podatke. Takav prozor može poslužiti za prikaz podataka aplikacije koja se izvodi na raèunalu, ali također može poslužiti i za prikaz izbornika. Nadalje, prozora ima razlièitih vrsta: osim prozora koji predstavljaju neko podruèje na ekranu, postoje prozori koji mogu imati okvir, polja za poveæavanje, smanjenje i zatvaranje pa i pridijeljen izbornik. Nazovimo takve prozore uokvirenim prozorima.

19.5.3. Definicija izbornika

Treæi važan element naše biblioteke elemenata grafièkog suèelja jesu izbornici (*meniji*). Pokušajmo izreæi kratku definiciju izbornika.

> *Izbornik je područje na ekranu koje omogućava izbor neke od ponuđenih opcija.*

Ova definicija dosta dobro ocrtava što je izbornik i èemu služi, ali ona ništa ne kaže o detaljima kako to izbornik radi. Zbog toga je potrebno definiciju proširiti i opisati što su opcije te navesti naèin na koji su opcije prikazane.

> *Pojedina opcija izbornika je kratak tekst koji ukratko opisuje akciju koju će program poduzeti nakon njenog izbora. Također, opcija izbornika može voditi u novi podizbornik.*

Sada veæ znamo dosta o strukturi izbornika. No još mnoga pitanja ostaju otvorena: kako izbornik izgleda kada se prikaže na ekranu, kako izgledaju pojedine opcije, kako se izabire opcija mišem i slièno. Dapaèe, razvoj klase izbornika koja æe se moæi koristiti je vrlo složen i obuhvaæat æe razne parametre. U razvoj takve klase osim inženjera

informatike mogu biti ukljuèeni primjerice i dizajneri koji æe odrediti vizualni identitet izbornika. Zbog toga æemo preskoèiti daljnju analizu ove klase.

Postoji tehnika koju su razvili američki stručnjaci K. Beck i W. Cunningham koja pomaže razvoju apstrakcije u velikim složenim projektima. Ta tehnika koristi takozvane CRC kartice (kratica od engl. *class, responsibility, collaboration – klasa, odgovornost, suradnja*). Ona se sastoji u tome da se za svaku klasu koja se pojavljuje u projektu formira jedna kartonska kartica točno određene veličine (kako se ne bi pretjerivalo s glomaznim definicijama) na koju se napiše naziv klase i njena glavna svojstva. Ljudi koji rade na projektu se zatim okupe na sastanku te svaki dobije jednu ili više kartica. Oni zatim uzimaju ulogu pojedinih klasa te pokušavaju odigrati više scenarija interakcije između objekata. Pri tome često postaje jasno da pojedine apstrakcije nisu adekvatne te ih se zatim mijenja u skladu sa zahtjevima. Sve te promjene se unose na kartice kako bi se na kraju mogla izvući bit svake klase. Također, pojedini sudionici mogu sugerirati da su neke apstrakcije previše komplicirane i zbunjujuće te predložiti nove. Male kartice prisiljavaju sudionike na jednostavne i razumljive apstrakcije. Pojedine kartice mogu biti smještene u grupe čime se označava pripadnost pojedinoj funkcionalnoj cjelini.

Iako takav postupak može ponekad izmaknuti kontroli i pretvoriti se u poslijepodnevno ludilo isfrustriranih i premalo plaćenih informatičara, pokazao se vrlo korisnim u početnom postupku razvoja apstrakcija nekog složenog sustava. Ovakvim neformalnim sastankom mnogi ljudi često izraze svoje zamisli koje inače nikada ne bi ispričali na sličnom sastanku formalnog karaktera.

19.6. Korak 3: Definicija odnosa i veza između klasa

Ovaj korak se redovito ne može promatrati zasebno, veæ zajedno s prethodnim korakom. No mi smo ga istakli zasebno zato jer je prilikom odreðivanja odnosa i veza potrebno biti vrlo pažljiv kako bi se izbjegle klasiène zamke.

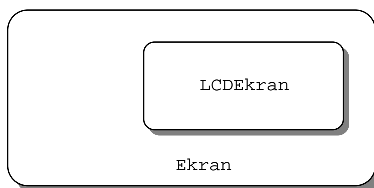
Već iz primjera iznesenog u prethodnom koraku vidljivo je da rijetko koja klasa postoji sama za sebe. Većina klasa na neki način dolazi u interakciju s drugim klasama. Zbog toga je za ispravan dizajn sustava vrlo važno ispravno definirati odnose između klasa. Postoje tri osnova tipa odnosa klasa:

- Odnos *biti* u kojemu je apstrakcija opisana nekom klasom `A` istodobno i apstrakcija opisana nekom drugom klasom `B`. Pri tome se može reći da je `A` podskup od `B`.
- Odnos *posjedovati* u kojemu neka apstrakcija opisana klasom `A` sadržava (posjeduje) neku drugu apstrakciju opisanu klasom `B`.
- Odnos *koristiti* u kojemu neka apstrakcija opisana klasom `A` koristi, ali ne posjeduje neku apstrakciju opisanu klasom `B`.

Vrlo je važno razumjeti svojstva svakog od ovih tipova odnosa te æemo ih razmotriti pojedinaèno.

Odnos *biti* je odnos u kojem se neki objekt `A` istodobno može promatrati i kao objekt nadklase `B`. Na našem primjeru, LCD ekran opisan klasom `LCDEkran` *jest* istodobno i običan ekran opisan klasom `Ekran`. Klasa `LCDEkran` je izvedena iz klase

`Ekran`, odnosno klasa `Ekran` je naslijeđena te je dobivena klasa `LCDEkran`. Takav je odnos prikazan na slici 19.1.



Slika 19.1. Relacija *biti*

Relacija *biti* oznaèava definiciju novog tipa na osnovi veæ postojeæeg tipa. Njena najèešæa implementacija u C++ jeziku je pomoæu javnog nasljeðivanja. Prilikom nasljeðivanja objekt klase `LCDEkran` æe sadržavati podobjekt klase `Ekran` i ta se èinjenica vrlo èesto interpretira na krivi naèin pomoæu relacije *posjedovati*.

Relacija *posjedovati* opisuje odnos kada neki objekt kao dio svog internog ustroja posjeduje ili sadrži neki drugi objekt. Na primjer, `Ekran` će sadržavati objekt `Paleta` koji će definirati listu boja koje se trenutno mogu prikazati na ekranu. `Ekran` nije podtip klase `Paleta` (niti obrnuto), on jednostavno sadržava paletu. `Paleta` će biti implementirana kao podatkovni član objekta `Ekran`.

Vidjeli smo da svaki objekt izvedene klase sadržava podobjekt osnovne klase. Neiskusan korisnik objektno orijentirane tehnologije bi mogao zbog toga zaključiti da se `Ekran` jednostavno može implementirati tako da se naslijedi klasa `Paleta`. Iako takvo rješenje može korektno raditi, ono je nekorektno sa stanovišta dizajna sustava. Svrha nasljeđivanja je opisati podtipizaciju, a ne sadržavanje objekata. Takvu manu u dizajnu sustava moglo bi se pokušati ispraviti tako da se `Ekran` izvede zaštićeno ili privatno, no niti to nije korektno. Takav program može raditi ispravno, ali će sigurno biti nerazumljiv. Programeri koji rade na razvoju takvog sustava bit će zasigurno zbunjeni navedenim pristupom rješavanju problema. Zbog toga će biti vrlo jednostavno prilikom pisanja programa načiniti pogrešku, te će dobiveni kôd gotovo sigurno biti neispravan. Također, ako se prilikom razvoja sustava ustanovi da dotična apstrakcija nije sasvim prikladna, bit će teško provesti izmjenu. Pomicanje pojedine klase u hijerarhiji prema gore ili dolje će također biti vrlo teško.

Potrebno je biti oprezan prilikom određivanja odnosa između pojedinih klasa te pokušati vrlo precizno odrediti je li neki odnos tipa *biti* ili *posjedovati*.

Relacija *koristiti* je najopæenitija – ona specificira da jedan objekt u svome radu samo koristi neki drugi objekt. Pri tome ta dva objekta koji ulaze u interakciju nisu niti na jedan drugi naèin meðusobno povezani (primjerice tako da je jedan objekt podatkovni èlan drugoga ili da postoje hijerarhijski odnosi između njih). Ta dva objekta mogu komunicirati putem njihovog javnog suèelja, ili možda èak pomoæu privatnog suèelja u

sluèaju da je jedan deklariran kao prijatelj drugog. Ovakav odnos se u C++ jeziku najèešæe implementira pokazivaèima i referencama između objekata. U našem primjeru možemo reæi da æe svaki `Prozor` koristiti `Ekran` za ispisivanje na njemu.

Osim tih osnovnih tipova odnosa između objekata postoji niz podtipova tih odnosa. Nakon identifikacije pojedinih odnosa moramo se odmah zatim zapitati je li taj odnos jednosmjeran ili dvosmjeran. Jednosmjerni odnosi se lakše implementiraju i zahtijevaju manje izvornog kôda za implementaciju. Zbog toga se oni i izvode znatno brže. No dvosmjerni odnosi su mnogo fleksibilniji, pa je zato posao korisnika objekata znatno pojednostavljen.

U našem slučaju odnos između `Ekrana` i `Prozora` će gotovo sigurno biti implementiran kao jednosmjeran – `Prozor` će koristiti `Ekran` za ispis svojih podataka, dok `Ekran` neće imati potrebe koristiti `Prozor` (barem na sadašnjem stupnju razvoja projekta toga nismo svjesni).

Gornji primjer odmah nas vodi na dodatna četiri podtipa odnosa: *jedan na jedan*, *jedan na više*, *više na jedan* i *više na više*. U našem slučaju odnos između prozora i ekrana je *više na jedan*: u nekom trenutku imat ćemo više aktivnih prozora koji će željeti crtati na ekranu. Također, možemo se zapitati koliko je to *više*: je li to neki fiksni broj (na primjer dva) ili neki promjenjivi broj.

Važan je i način na koji se odnosi mijenjaju tijekom vremena. Pojedine veze vrijede samo dok postoje povezani objekti. Na primjer, prozori se mogu u toku rada računala stvarati i uništavati. Veza prozora s ekranom postoji samo dok postoji prozor, kada prozor biva uništen, veza također prestaje. Nadalje, veza prozora s ekranom nije potrebna cijelo vrijeme; prozor pristupa ekranu dok želi obaviti nekakav ispis. Nakon toga veza do sljedećeg ispisa nije potrebna. Svaki prozor može prepustiti ekran za vrijeme perioda dok ne obavlja ispis nekom drugom prozoru. Na taj način se može odrediti protokol po kojem u svakom trenutku samo jedan prozor ima mogućnost ispisa. Da li će se takva implementacija primijeniti ovisi o nizu čimbenika, kao što su fizička svojstva stvarnog ekrana s kojim raspolažemo, operacijskim sustavom koji se koristi u računalu i slično. Radi jednostavnije implementacije nećemo uvoditi kontrolu pristupa, nego ćemo se zadržati na originalnom rješenju kod kojeg više prozora može pristupati ekranu odjednom.

19.6.1. Odnosi objekata u korisnièkom suèelju

U nastavku æemo pokušati primijeniti navedena pravila kako bismo precizno odredili odnose između objekata koje smo izveli u prethodnom poglavlju.

Počet ćemo s klasom `Ekran`. Ona opisuje fizičku izlaznu jedinicu koja služi za komunikaciju s korisnikom. Svakom ekranu je pridružena slika koja se sastoji od niza točaka. Možemo definirati klasu `Slika` koja će opisivati sliku na ekranu. Ona će omogućavati radnje kao što su postavljanje boje određene točke na ekranu, čitanje boje neke točke i slično. Također će sadržavati mehanizme za podržavanje različitih razlučivosti.

Iz navedenog se može lako zaključiti da će svaki ekran posjedovati točno jednu sliku (i ta slika će biti pridružena samo tom jednom ekranu). Dakle, veza je tipa *jedan na jedan* te dok postoji ekran postoji i slika i obrnuto.

Kada smo pokušali razmotriti način na koji funkcionira većina ekrana, rekli smo da mnogi ekrani ne mogu prikazati sve boje odjednom. Zbog toga svaki ekran održava paletu boja – tablicu u kojoj su navedene pojedine boje. Ustanovili smo da je paleta vrlo pogodna za apstrakciju pa smo uveli klasu `Paleta` koja će omogućavati operacije kao što su unos boje u paletu i iščitavanje boje u pojedinom registru palete. Svaki `Ekran` će sadržavati točno jednu klasu `Paleta`. Ta će veza također biti *jedan na jedan*. Također, dok postoji ekran postoji i paleta i obrnuto, pa je i ova veza slična prethodnoj.

Slika se sastoji od niza točaka. Točka je bila definirana kao najmanja jedinica prikaza te će se prikazati klasom `Tocka`. Možemo reći da će svaka slika posjedovati određen broj točaka. Pokušajmo ustanoviti malo detaljnije tip te veze.

Broj točaka u pojedinoj slici ovisi o rezoluciji u kojoj grafički podsustav računala trenutno radi. Zbog toga će i broj točaka koji će slika posjedovati varirati od slučaja do slučaja. Klasa `Slika` mora ostvariti mehanizme kojima će prilagoditi svoje ponašanje pojedinom modu rada računala. Veza između točaka i slike će biti *više na jednu*.

Na ovom mjestu valja naglasiti smisao gornje veze. Mi nipošto ne pokušavamo sugerirati da se problem grafičkog sučelja treba riješiti tako da se u memoriji alocira prostor za broj točaka jednak broju točaka kojim raspolaže grafički podsustav te da se crtanje provodi postavljanjem boja pojedinih točaka. Takvo rješenje će biti vrlo daleko od idealnog: osim što će zauzimati veliku količinu memorije (na primjer za razlučivost od 800×600 točaka u 16 boja potrebno bi bilo odvojiti najmanje $800 \times 600 \times 0.5 = 240000$ bajtova), bit će iznimno sporo. Nema smisla odvajati dodatnu memoriju kada to već čini grafički adapter za nas. Ovdje govorimo o apstrakcijama: `Slika` se veže za $800 \times 600 = 480000$ objekata klase `Tocka`. No ništa se ne kaže o implementaciji te veze. U gornjim definicijama nije nigdje specificirano da će se slika realizirati tako da će odvojiti zaseban memorijski prostor za te točke. Ta veza je posve apstraktna.

Klasa `Slika` mora omogućiti sučelje kojim će se korisnicima klase (primjerice klasama `Ekran` ili `Prozor`) mehanizam prikaza slike prezentirati na opisani način. Ako klasa `Slika` omogućava da se funkcijskim članom pristupi određenoj točki, onda ona mora svojim unutarnjim mehanizmom to osigurati. Željeno se može postići na više načina i odvajanje memorije za svaku točku je sigurno jedan od njih, no on je daleko od idealnog. Jedna znatno bolja mogućnost je da klasa `Slika`, nakon upita za određenom točkom na nekom mjestu na slici, pročita vrijednost pohranjenu u memorijskom spremniku uređaja za prikaz, tu vrijednost upakira u objekt klase `Tocka` te taj objekt vrati pozivatelju. Takva implementacija će biti znatno kvalitetnija, jer neće tražiti bilo kakvu dodatnu dodjelu memorije. Pri tome je integritet apstrakcije očuvan: svaka slika se može promatrati kao niz određenog broja točaka. Stvar je implementacije, a ne apstrakcije, kako će se takav model realizirati.

Klasa `Tocka` će biti razmjerno jednostavna: ona modelira indeks boje iz palete te ne sadrži niti jednu drugu klasu.

Vratimo se na klasu `Paleta`. Ako pomnije razmotrimo njeno ustrojstvo, možemo zaključiti da se svaka paleta sastoji od niza boja. Zbog toga je prikladno uvesti klasu `Boja` koja će opisivati pojedinu stavku u paleti. Takva klasa mora omogućavati jednoznačno definiranje pojedine boje. Najjednostavnije se čini uvesti klasičan RGB koordinatni sustav boja te reći da se `Boja` sastoji od tri podatkovna člana koji specificiraju udio crvene, zelene i plave boje. No takva definicija previše zadire u implementaciju. Iako je RGB sustav boja vjerojatno najprikladniji za korištenje na ekranima računala, to ne mora biti jedini izbor. Ponekad nam može biti vrlo prikladno da se boja prikaže u HSB sustavu (gdje se svaka boja prikazuje pomoću tona, zasićenja i svjetline) ili čak CMY sustavu (gdje se boja prikazuje pomoću udjela modrozelene, purpurne i žute). To može biti korisno želimo li boje na ekranu uskladiti s bojama na pisaču. Ako se držimo gornje definicije, bit će teško uvesti novi koordinatni sustav boja zato jer će implementacija biti poznata okolnom svijetu. Uvijek će postojati opasnost da neki objekt pristupi direktno implementaciji, narušavajući integritet objekta. Bolje je reći da klasa `Boja` osigurava mehanizme kojima se pojedina boja može prikazati u bilo kojem koordinatnom sustavu: RGB, HSB ili CMY. Njena implementacija pri tome može biti potpuno skrivena od okolnog svijeta.

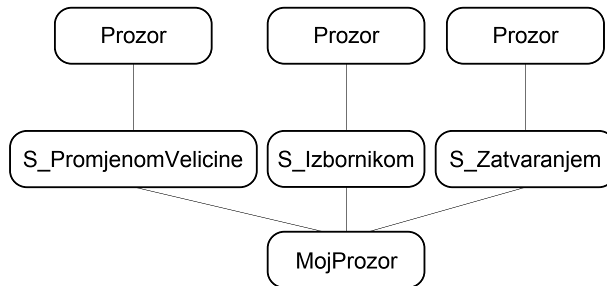
Klasa `Prozor` će definirati osnovna svojstva prozora. No ona će istodobno služiti i kao osnovna klasa pomoću koje će biti moguće izvoditi nove klase za opis prozora s pojedinim svojstvima.

U prethodnom smo odjeljku napomenuli da će klasa `Prozor` označavati općeniti prozor bez okvira i elemenata koji omogućavaju posebne akcije, kao što su povećavanje i smanjenje, zatvaranje i slično. Također, klasa `Prozor` ne podržava izbornike. No rekli smo da ćemo taj problem riješiti tako što ćemo općeniti prozor dodatno tipizirati. Uvest ćemo podklasu `UokvireniProzor` koja će dodati željenu funkcionalnost.

Odnos klase `UokvireniProzor` i klase `Prozor` je tipičan primjer relacije *biti*. Svaki uokvireni prozor istodobno je i prozor, dok obrat ne vrijedi. `UokvireniProzor` će se najbolje implementirati tako da se naslijedi klasa `Prozor` i doda željena funkcionalnost.

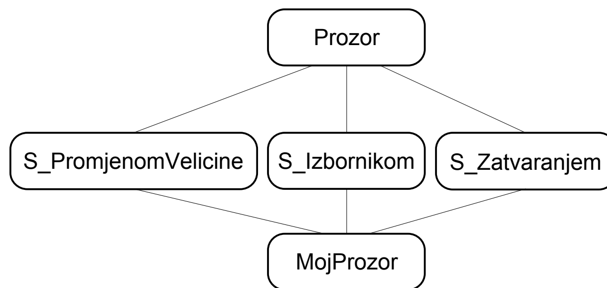
Svaki prozor ima niz dodatnih elemenata koji omogućavaju posebnu manipulaciju prozorima: zatvaranje, povećanje, maksimizaciju, minimizaciju, sistemski izbornik i slično. Zbog toga bi se moglo postaviti pitanje je li možda bilo pogodnije za svaki od tih funkcionalnih elemenata izvesti zasebnu klasu koja će općem prozoru dodati željeno svojstvo. Prozor koji ima više tih elemenata (primjerice i element za zatvaranje i element za promjenu veličine) mogao bi se realizirati višestrukim nasljeđivanjem pojedinih klasa.

Postoji više razloga zašto takve implementacije treba izbjegavati. Gornjim postupkom uvodi se velika konfuzija u hijerarhijsko stablo te se dodatno komplicira dizajn korisničkog sučelja. Možda najočitija pogreška je u tome što ako naslijedimo klase koje uvode tri elementa, na primjer `S_PromjenomVelicine`, `S_Izbornikom` i `S_Zatvaranjem`, rezultirajući objekt će imati u sebi tri podobjekta klase `Prozor`. Takvo hijerarhijsko stablo se može vidjeti na slici **Error! Reference source not found.**.



Slika 19.2. Pogreška prilikom izvođenja klase `MojProzor`

Ta mana bi se mogla ispraviti tako da se klasa `Prozor` uèini virtualnom prilikom izvoðenja klase `S_PromjenomVelicine`, `S_Izbornikom` i `S_Zatvaranjem`. Time bi sve klase izvedene iz tih klasa imale samo po jedan podobjekt `Prozor`, kako je prikazano hijerarhijskim stablom na slici 19.3.



Slika 19.3. Popravak pogreške korištenjem virtualnog nasljeđivanja

Gornje rješenje je korektno, ali još uvijek uvodi dodatne komplikacije u hijerarhiju. Kao i uvijek prilikom višestrukog nasljeðivanja, tri klase koje uvode elemente za kontrolu prozora mogu definirati èlan istog naziva. Zbog toga æe za pristup pojedinim èlanovima klase `MojProzor` biti neophodno navesti osnovnu klasu u kojoj je èlan definiran.

Osnovnu dizajnersku pogrešku smo napravili već pri izradi apstraktnog modela. Prozor s elementom za zatvaranje ili za promjenu veličine nije suštinski različit od bilo kojeg drugog prozora. Element za zatvaranje se može promatrati kao objekt koji prozor posjeduje, ili jednostavno kao modifikator koji mijenja način na koji prozor funkcionira. U gornjem rješenju smo relaciju *posjedovati* pokušali realizirati pomoću relacije *biti*.

Prvo bi rješenje bilo generirati dodatne klase izvan hijerarhije prozora. Zatim bi klasa `UokvireniProzor` morala uvesti metode za manipulaciju pojedinih elemenata (dodavanje, uklanjanje, promjenu položaja i slično) te za komunikaciju između njih. Takvo rješenje je znatno korektnije od simulacije posjedovanja pomoću nasljeđivanja.

Još se jednostavnije rješenje može postići ako kažemo da je nazočnost ili odsutnost pojedinog elementa na prozoru jednostavno atribut prozora. Prilikom stvaranja prozora potrebno je odrediti koji su elementi prisutni (na primjer, pomoću dodatnog parametra konstruktoru klase), a zatim se sam prozor brine za njihovo ispravno funkcioniranje. Na taj način smo različito ponašanje pojedinih objekata klase riješili parametrizacijom objekata.

Preostaje nam još razmotriti strukturu i veze između objekata koji će sačinjavati izbornike. Izbornik ćemo predstaviti pomoću klase `Izbornik`. Prvo moguće pitanje jest kakva je veza između izbornika i prozora. Ako se malo bolje razmisli, može se vidjeti da izbornik nije ništa drugo nego prozor s posebnom namjenom. Uloga izbornika je prikazivanje izbora u točno određenom području na ekranu. Dakle, moglo bi se reći da je `Izbornik` tip izveden iz tipa `Prozor` – realizacija pomoću javnog nasljeđivanja odmah pada napamet.

No javno nasljeđivanje pretpostavlja da se cijelo sučelje osnovne klase uključi u izvedenu klasu. Javno sučelje izbornika će sigurno biti drukčije nego javno sučelje prozora. Pravilnije je i jednostavnije za korištenje odvojiti sučelje izbornika od sučelja prozora, jer se ta dva objekta dosta razlikuju po načinu korištenja. Zbog toga je bolje `Izbornik` izvesti pomoću privatnog nasljeđivanja. Sučelje prozora se time skriva, a korištenje izbornika se omogućava pomoću novog, prikladnijeg sučelja.

Nadalje, izbornik nikada ne postoji sam za sebe. On je uvijek dodijeljen nekom prozoru koji vodi računa o prikazivanju izbornika i njegovom povezivanju s aplikacijom. Već smo ranije naveli da neće svaki prozor imati izbornik, nego samo prozori definirani klasom `UokvireniProzor`. Pri tome prozor može imati izbornik, ali i ne mora. Konceptualno, prozor sadrži izbornik, no to sadržavanje se neće rješavati tako da se definira podatkovni član tipa `Izbornik`. Bolje je vezu s izbornikom riješiti pokazivačem na izbornik. Nul-vrijednost pokazivača može označavati da prozor nema izbornika.

Kao pouku gornjeg primjera važno je razumjeti da koncepcijske veze između objekata nisu niti na koji način strogo vezane s načinom implementacije. Javno nasljeđivanje najčešće označava tipizaciju objekata, no to ne mora biti isključivo pravilo. Također, posjedovanje objekta se u principu rješava pomoću podatkovnih članova, no može se riješiti i na bilo koji drugi način. Bitno je ispravno postaviti veze između objekata te na kraju napraviti implementaciju koja će odgovarati apstraktnom modelu.

Naposljetku, tu su nam još klase `Mis` i `Tipkovnica`. Slično klasi `Ekran`, oni modeliraju stvarne uređaje u računalu. Pomoću njih preostali objekti mogu doznati trenutnu poziciju miša ili čitati koje se tipke pritišću.

19.7. Korak 4: Definicija implementacijski zavisnih apstrakcija

Apstrakcije koje smo do sada izveli i veze između njih dosta dobro opisuju suèelje naše biblioteke klasa prema korisniku. No još ništa ne znamo o naèinu na koji se postiže

harmonièan rad svih komponenti. Prilikom razmatranja moguænosti implementacije može se doæi do novih apstrakcija. Da bi se pojasnio model koje te apstrakcije zastupaju, potrebno je ponovo proæi korake od 1 do 3.

U prethodnom odsječku smo rekli da klase `Mis` i `Tipkovnica` omogućavaju pojedinim objektima pristup do stvarnog uređaja u računalu. Međutim, ništa nije rečeno o tome kako će se to napraviti. Prvo moguće rješenje koje pada na pamet može biti da se jednostavno kaže da klasa `Prozor` koristi objekt klase `Mis` i objekt klase `Tipkovnica` za čitanje ulaznih podataka. No stvari nisu tako jednostavne.

Miš i tipkovnica su uređaji koji su zajednički za sve programe koji se izvode na računalu. Zbog toga se u sustavima koji koriste prozore uvodi konvencija po kojoj se tipkovnica privremeno dodjeljuje trenutno aktivnom prozoru. Prozor se može postaviti aktivnim pomoću miša tako da se klikne tipkom miša unutar prozora. Također, pomaci miša i pritisci na tipke miša se uvijek prosljeđuju prozoru iznad kojeg se miš trenutno nalazi.

Zbog navedenog je potrebno uvesti podsustav koji će obavljati gore opisane poslove. On će održavati listu postojećih prozora te će stalno čitati pomake miša. Ako se klikne mišem u području nekog prozora koji nije aktivan, on će automatski deaktivirati do tog trenutka aktivan prozor i aktivirat će prozor iznad kojeg je tipka miša pritisnuta. Također, podsustav će čitati tipkovnicu te će automatski usmjeravati sve pritiske na tipke u trenutno aktivan prozor. Dakle, radi se o nekoj vrsti posrednika između stvarnih uređaja i prozora.

Takav podsustav će se opet prikazati jednom klasom koja će se instancirati točno jednom nakon pokretanja programa. Idealno bi zapravo bilo definirati takav podsustav na razini operacijskog sustava, no radi jednostavnosti nećemo se upuštati u vode složenog sistemskog programiranja. Nazvat ćemo tu klasu `Prometnik` – objekti te klase se brinu za promet podataka unutar računala.

Pokušajmo odrediti s kojim objektima klasa `Prometnik` dolazi u interakciju, te na koji način. Prvo i osnovno, prozori ne smiju imati direktan pristup do miša i tipkovnice. Klasa `Prometnik` će imati isključivo pravo čitanja tih podataka. Nadalje, ta klasa će biti u relaciji *koristiti* sa svim prozorima na zaslonu. Svaki prozor se mora prilikom svog stvaranja prijaviti u prometničku listu. Ta veza traje samo do trenutka kada se prozor uništava – tada je potrebno ukloniti prozor iz liste. U suprotnom, može doći do velikih komplikacija pokuša li `Prometnik` obavijest o pritisnutoj tipki proslijediti nepostojećem prozoru. Veza između prometnika i prozora je, dakle, dvosmjerna.

Postavlja se pitanje na koji način će se ostvariti veza između prometnika i prozora. Jedan od najprikladnijih načina za primjenu u C++ jeziku jest uvesti u klasu `Prozor` po jedan funkcijski član za svaki tip događaja koji prometnik može signalizirati prozoru. Tako ćemo imati članove koji signaliziraju pokretanja miša, pritisak lijeve tipke miša, pritisak desne tipke miša te pritisak na tipku. Po potrebi bi se prometnik mogao proširiti na obrađivanje dodatnih događaja kao što su protjecanje određenog vremenskog intervala ili neki sistemski događaj kao što je primjerice ubacivanje nove diskete u disketnu jedinicu. Za svaki od tih događaja bilo bi potrebno uvesti novi funkcijski član u klasu `Prozor` kojeg će `Prometnik` pozivati po nastupu događaja.

19.8. Korak 5: Definicija suèelja

Do sada smo ugrubo opisali hijerarhiju objekata koji æe saèinjavati biblioteku korisnièkog suèelja. Razmotrili smo skup apstrakcija koje æemo koristiti za naš sustav te smo pokušali što preciznije definirati svrhu pojedine apstrakcije. Sada smo došli do toèke kada je potrebno precizno definirati suèelje pojedine klase te definirati naèin na koji æe objekti meðusobno komunicirati.

Ako bismo se željeli strogo držati pravila objektnog programiranja, morali bismo sučelje definirati bez razmišljanja o implementaciji. No to često nije sasvim moguće. Dosljedno provođenje takvog pristupa rezultiralo bi nepotrebnim konverzijama podataka između unutarnjeg formata definiranog implementacijom i formata kojeg koristi sučelje. Nije uvijek dobar princip žrtvovati učinkovitost programa (pogotovo ako je brzina obrade podataka vrlo važna, kao na primjer kod aplikacije za obradu podataka u stvarnom vremenu) da bi se poštivalo objektno ustrojstvo programa. Također, prilikom razmatranja o vezama između objekata, neminovno se automatski razmišlja i o načinu implementacije i o elementima sučelja.

Dijelove sučelja smo dotakli u neformalnom obliku već u prethodnim poglavljima. Naime, pojedini koraci u razvoju objektnog modela se ne mogu provoditi izolirano, već su isprepleteni. Kada se govori o odnosima između klasa, većina programera odmah razmišlja i o sučelju i o implementaciji. Svrha postupka kojeg smo provodili do sada nije bili sasvim odvojiti pojedine korake, već naglasiti važne točke u razvoju.

Za uspješnu primjenu objektno orijentiranog pristupa modeliranju sustava vrlo je važno što dosljednije provoditi gornji postupak kojim se implementacija razdvaja od sučelja. Taj postupak se naziva *skrivanje podataka* (engl. *data hiding*): svaki objekt pruža okolini upravo onoliko informacija koliko je potrebno da bi okolina mogla uspješno komunicirati s njime. Implementacijski detalji su skriveni, jer okolina o njima niti ne mora voditi računa.

Programi pisani tako da se strogo pridržavaju principa skrivanja informacija vrlo rijetko omogućavaju direktan pristup podatkovnim članovima. Podatkovni članovi su implementacija, a sučelje se najčešće osigurava pomoću funkcijskih članova. Klasa `Boja` bi se tada mogla implementirati ovako:

```
class Boja {
private:
    unsigned short udioR, udioG, udioB;
public:

    Boja(unsigned short R = 0, unsigned short G = 0,
            unsigned short B = 0) : udioR(R), udioG(G),
                                    udioB(B) {}

    // sučelje za RGB sustav
    unsigned short dajR() { return udioR; }
    unsigned short dajG() { return udioG; }
    unsigned short dajB() { return udioB; }
    void postaviRBG(unsigned short R, unsigned short G,
```

```
                    unsigned short B) {
            udioR = R; udioG = G; udioB = B;
        }

        // sučelje za HSB sustav
        unsigned short dajH();
        unsigned short dajS();
        unsigned short dajBr();      // dajB je već deklariran,
                                     // pa moramo koristiti naziv
                                     // dajBr
        void postaviHSB(unsigned short H, unsigned short S,
                        unsigned short B);

        // sučelje za CMY sustav
        unsigned short dajC();
        unsigned short dajM();
        unsigned short dajY();
        void postaviCMY(unsigned short C, unsigned short M,
                        unsigned short Y);

    };
```

U gornjem primjeru za implementaciju je odabran RGB sustav boja, no pristup podatkovnim èlanovima nije omoguæen. Naprotiv, bojama se pristupa pomoæu funkcijskih èlanova. Kako su ti èlanovi definirani kao umetnuti, dobar prevoditelj æe generirati kôd koji se neæe izvoditi sporije nego kôd koji direktno pristupa podatkovnim èlanovima. Ako je potrebno boju proèitati u nekom drugom sustavu, pozvat æe se funkcijski èlan koji æe provesti pretvorbu u taj sustav. Ti èlanovi nisu ovdje definirani te nisu umetnuti, jer je pretvorba sustava boja vrlo složen postupak koji nema smisla umetati na svako mjesto gdje se traži pristup primjerice H komponenti.

Prednost ovakvog pisanja programa je u tome što, ako se zbog nekog razloga implementacija promijeni, sučelje ostaje isto. Svi objekti koji koriste ovu klasu moći će pristupati R, G i B članovima neovisno i na isti način. Stvar je implementacije da ostvari način da takve podatke učini dostupnima.

Uzmimo, na primjer, da pišemo program koji vrlo intenzivno barata s bojama u drugim koordinatnim sustavima. Zbog toga će česte pretvorbe sustava imati utjecaja na brzinu izvođenja programa. Programer koji razvija klasu `Boja` može promijeniti implementaciju tako da doda još šest podatkovnih članova u kojima će se pamtiti koordinate boje u svim sustavima. Prilikom postavljanja boje u bilo kojem sustavu, automatski će se zadana boja preračunati u sve ostale te se tako učiniti brzo dostupnim bez potrebe za naknadnim preračunavanjem. Ako korisnik klase sada pokuša pristupati bojama u drugom sustavu, primijetit će da se program brže izvodi. No ključno je da se bojama pristupa na isti način. Da je pristup podatkovnim članovima bio omogućen korisnicima klase, ovakvo poboljšanje ne bi bilo izvedivo jer bi nakon svakog postavljanja podatkovnih članova u RGB, korisnici morali sami pretvarati sustave i voditi brigu o takvim "kućanskim" poslovima (*Pa da, cijeli dan samo pretvaram sustave, a na kraju me niti van ne izvodiš...*).

Obratite pažnju na konstruktor iz gornjeg primjera: on automatski postavlja boju u RGB sustavu. Korištenjem takvog konstruktora omogućena je sintaksa po kojoj se mogu zadavati konstante boja, kao u primjeru:

```
Boja nizBoja[10];
nizBoja[0] = Boja(10, 250, 70);
```

Ovo možemo protumaèiti kao da je `Boja(10, 250, 70)` konstanta koja se nalazi s desne strane operatora pridruživanja, te se pridružuje nekom elementu niza `nizBoja`. Valja uoèiti da zbog implementacije klase nije potrebno uvoditi posebne operatore pridruživanja ili konstruktore kopije.

Prilikom definicije sučelja vrlo je važno ispravno odrediti atribute pojedinih elemenata sučelja. U C++ jeziku pod atributima se primarno misli na virtualnost, statičnost, konstantnost i javnost. Pozabavimo se malo pravilima kojima se određuju ti atributi.

Vrlo je važno ispravno odrediti je li neki funkcijski član klase virtualan ili ne. Pravilo "od palca" (engl. *rule of the thumb*) koje se nalazi u većini knjiga o objektnom programiranju kaže da sve funkcijske članove, za koje se očekuje da će biti promijenjeni prilikom nasljeđivanja, treba obavezno učiniti virtualnima. Lijepo rečeno, no ne baš previše korisno! Naime, sada je odgovor prebačen na drugo pitanje: "A koji će funkcijski članovi biti promijenjeni prilikom nasljeđivanja?!"

Odgovor na gornje pitanje može se dobiti ako se shvati bit virtualnosti. Virtualan funkcijski član specificira akciju koja je vezana uz tip podataka na kojemu se obavlja. Ako se akcija za različite tipove obavlja na isti način, tada ona nije vezana uz tip te ju nije potrebno učiniti virtualnom. Štoviše, preporuka je da se virtualni funkcijski članovi koriste samo kada je to nužno.

Poziv virtualnog člana najčešće se realizira pomoću indirekcije (poziva preko pokazivača) koji se izvodi sporije od direktnog poziva. Virtualan član se ne može učiniti umetnutim, čime se još više degradira učinkovitost programa.

Vratimo se na primjer klase `Boja` i promotrimo jesmo li ispravno odredili tip funkcijskih èlanova. Prvo je pitanje da li uopæe oèekujemo da æe ikad biti potrebno naslijediti klasu `Boja`. Ta je klasa razvijena iskljuèivo kao pomoæna klasa za klasu `Paleta`. Teško je zamisliti koji bi bio smisao u nasljeðivanju klase `Boja`. Zbog toga nema niti smisla govoriti o virtualnim funkcijskim èlanovima –njima bismo samo znatno usporili izvoðenje programa.

Razmotrimo sučelje klase `Paleta`. Velika je sličnost između palete i niza – paleta zapravo i jest niz boja. Zbog toga su operacije nad paletom slične operacijama s nizovima: potrebno je definirati veličinu palete te osigurati pristup pojedinim članovima.

Veličina palete se može definirati već prilikom njenog stvaranja, dakle u konstruktoru. Također, korisnik prilikom rada s računalom može htjeti promijeniti broj

boja, pa je veličinu palete potrebno mijenjati i u toku rada. Uvest ćemo stoga funkcijski član `BrojBoja()` koji će podesiti veličinu palete na željeni broj boja. Član `DajBrojBoja()` će omogućiti čitanje broja boja u paleti. Pristup pojedinoj boji može se realizirati na razne načine. Jedan od vrlo prikladnih je adresiranje palete pomoću operatora `[]` – time se naglašava čitatelju programskog kôda da je paleta neka vrsta niza. Napisat ćemo kako bi izgledala deklaracija klase `Paleta`:

```
class Paleta {
public:

    Paleta(int brBoja);

    void BrojBoja(int brBoja);
    int DajBrojBoja();

    ?? operator [](int indeks);      // nismo još odredili
                                     // povratni tip

};
```

Razmotrimo koji bi tip bilo najprikladnije vratiti iz operatora `[]`. Jedno od moguæih rješenja koje se nameæe samo po sebi veæ je opisano u poglavlju o preoptereæenju operatora kod definicije klase `Matrica`, a to je vratiti referencu na objekt `Boja`:

```
class Paleta {
    // ...
    Boja &operator [](int indeks);
    // ...
};
```

Što smo dobili ovakvim rješenjem? Vraæanjem reference na objekt smo automatski omoguæili da se operator `[]` nađe s lijeve strane znaka pridruživanja. Time bi bilo omoguæeno postavljanje boja u paletu po sljedeæem principu:

```
Paleta p;
p[0] = Boja(127, 127, 127);
```

No takvo rješenje ne mora uvijek odgovarati. Ono zapravo pretpostavlja toèno određenu implementaciju koja ne mora uvijek biti ispravna. Naime, u gornjem sluèaju najjednostavnija implementacije bi bila kada bi klasa `Paleta` sadržavala niz objekata klase `Boja`. Operator `[]` jednostavno vraæa referencu na određeni element niza. Gornje pridruživanje se tada provodi tako da se pozove operator `=` za klasu `Boja`. Klasa `Paleta` više nema nikakvog nadzora nad pridruživanjem svojim èlanovima – nakon pridruživanja kontrola se ne vraæa više u klasu pa dodatna obrada podataka nije moguæa.

Međutim, u našem bi slučaju upravo to bilo potrebno: nakon dodjele određenom elementu palete potrebno je uskladiti sistemske registre na grafičkoj kartici s RGB

vrijednostima dodane boje. Štoviše, paleta uopće ne mora biti realizirana tako da ona sadrži niz objekata `Boja` – boje u tekućoj paleti se nalaze zapisane u registrima grafičke kartice. Implementacija palete može prilikom dohvaćanja pojedine boje jednostavno pročitati boju iz registra te vratiti privremeni objekt klase `Boja` koji će sadržavati pročitane vrijednosti. Klasa `Paleta` zapravo služi kao *omotač* (engl. *wrapper*) oko sistemskih poziva grafičke kartice te omogućava korištenje jednostavnog objektnog sučelja umjesto složenih sistemskih poziva.

Zbog svega gore navedenoga, operator `[]` će vraćati privremeni objekt klase `Boja`. Postavljanje boje će se obavljati pomoću funkcijskog člana `PostaviBoju()` koji kao parametre ima cijeli broj za identifikaciju mjesta u paleti na koje se boja postavlja, te referencu na objekt klase `Boja`. Konačna deklaracija klase `Paleta` izgledat će ovako:

```
class Paleta {
public:

    Paleta(int brBoja);

    void BrojBoja(int brBoja);
    int DajBrojBoja();

    Boja operator [](int indeks);
    void PostaviBoju(int indeks, Boja &novaBoja);

};
```

Sljedeæi zadatak koji nas èeka jest definiranje suèelja klase `Slika`. Ta klasa mora uvesti mehanizme kojima se može crtati na izlaznoj jedinici raèunala. Slika se sastoji iz niza toèaka poredanih okomito i vodoravno. Klasa `Tocka` opisuje svaku toèku na slici. Osnovno svojstvo toèke je boja (toènije reèeno, indeks boje u paleti), pa æe ta klasa sadržavati funkcijske èlanove za postavljanje i èitanje boje. Takoðer, kako je indeks u biti cijeli broj, uvest æemo i operator konverzije klase `Tocka` u cijeli broj, te konverziju cijelog broja u objekt klase `Tocka`. Evo moguæe deklaracije suèelja:

```
class Tocka {
public:

    Tocka(int ind = 0); // konstruktor i konverzija
                        // int ⇒ Tocka
    operator int();     // konverzija Tocka ⇒ int

    int DajIndeks();
    void PostaviIndeks(int ind);

};
```

Buduæi da nije za oèekivati da æe klasa `Tocka` ikad biti naslijeðena, niti jedan funkcijski èlan nije virtualan.

Vratimo se na klasu `Slika`. Ona mora posjedovati mehanizme za čitanje razlučivosti u *x* i *y* smjeru, postavljanje razlučivosti te niz rutina za iscrtavanje elemenata slike: točaka, linija, pravokutnika i sl. Iako će komercijalno dobavljiva klasa `Slika` omogućavati rad i s egzotičnijim grafičkim elementima kao što su bit-mape, ikone itd., radi jednostavnosti ćemo se ograničiti na točke, linije i pravokutnike. Također, nećemo komplicirati s mogućnostima kao što su odsijecanje dijela slike izvan nekog zadanog područja, pomak i/ili rastezanje koordinatnog sustava i slično. Naprotiv, operacija koja će pročitati točku na određenom mjestu na ekranu je neophodna. Na kraju, `Slika` će imati konstruktor i destruktor koji će ispravno inicijalizirati objekt početnim načinom prikaza te ga uredno počistiti.

Valja još razmisliti moraju li pojedini članovi klase biti deklarirani virtualnima ili ne, odnosno, očekuje li se nasljeđivanje klase. Na prvi pogled većina će korisnika reći da ne očekuje da će klasa `Slika` biti naslijeđena: na kraju, slika je prikazana na ekranu računala i tu nema mnogo razmišljanja. No ne mora uvijek biti tako.

Na tržištu postoji mnoštvo grafičkih kartica od kojih svaka ima niz različitih mogućnosti. Ima smisla, stoga, klasu `Slika` vezati uz određenu grafičku karticu te na taj način osigurati izvođenje svake operacije na adekvatan način. Takvim pristupom se služe mnogi suvremeni operacijski sustavi: operacije zajedničke svim karticama izluče se na jedno mjesto te se time definira opće korisničko sučelje kojim se služe aplikacije. Za svaku karticu se napiše odgovarajući *pogonitelj* (engl. *driver*) koji za nju definira način na koji se pojedina operacija realizira. Iako se današnji operacijski sustavi ne koriste objektnim tehnologijama za postizanje tog cilja, gore opisano rješenje klase `Slika` omogućava upravo to. Klasa `Slika` može biti definirana kao apstraktna klasa (što znači da su njeni funkcijski članovi definirani kao čiste virtualne funkcije – njihova definicija se prepušta izvedenim klasama). Svrha klase `Slika` jest definiranje sučelja; svaka izvedena klasa predstavlja jedan pogonitelj.

Pomoću gore navedenih relacija moguće je jednostavno ostvariti izvođenje programa na jednom računalu, a prikaz slika na drugom računalu (slično X-Windowsima na UNIX operacijskim sustavima). Možemo definirati klasu `MreznaSlika` izvedenu iz klase `Slika` koja će umjesto iscrtavanja slike na ekranu računala podatke o slici upakirati i odgovarajućim protokolom poslati preko mreže na drugo računalo koje će te podatke otpakirati i prikazati na zaslonu. Takav sustav može biti vrlo koristan, jer pomoću njega možemo pokrenuti programe na više računala, a njihovo izvođenje pratiti na jednom računalu (može biti vrlo praktično u slučaju obavljanja složenog proračuna koji se provodi na više računala u mreži).

Važno je shvatiti da klasa `Slika`, slično klasi `Paleta`, služi kao omotač oko sučelja stvarnih sklopova u računalu. Implementacija nije niti na koji način prejudicirana. Evo moguće deklaracije klase:

```
class Slika {
friend class Ekran;
protected:
    void PostaviPaletu(Paleta *pal);
public:
```

```
    Slika(int pocetniNacin);
    virtual ~Slika();

    virtual int DajXRazlucivost();
    virtual int DajYRazlucivost();
    virtual int DajBrojBoja();
    virtual void PostaviNacinPrikaza(int nacin);

    virtual void CrtajTocku(int x, int y, Tocka &toc);
    virtual Tocka CitajTocku(int x, int y);
    virtual void Linija(int x1, int y1, int x2, int y2,
                        int indeksBoje);
    virtual void Kvadrat(int x1, int y1, int x2, int y2,
                         int indeksBoje);

};
```

Gornja deklaracija je samo primjer koji pokazuje kako bi se deklaracija mogla obaviti – to nipošto nije potpuna deklaracija koja bi obuhvatila svu potrebnu funkcionalnost klase `Slika`. Objasnit æemo ukratko pojedine elemente suèelja.

Konstruktor klase `Slika` ima jedan cjelobrojni parametar kojim se identificira početni način prikaza. Nećemo ulaziti dublje u problematiku što sačinjava pojedini način prikaza te kako se oni numeriraju – to ovisi o konkretnom računalu na kojemu radimo. Recimo samo da se pojedini način prikaza identificira cijelim brojem kojim se određuje razlučivost u *x* i *y* smjeru te broj raspoloživih boja.

Destruktor ima ulogu oslobađanja svih resursa koje je objekt zauzeo, a definiran je virtualnim kako bi se omogućilo nasljeđivanje klase. Tako sve izvedene klase mogu definirati svoje destruktore, a virtualni mehanizam će omogućiti ispravno uništavanje objekata.

U zaštićenom dijelu sučelja naveden je član `PostaviPaletu()`. Njegova je uloga uspostavljanje veze između palete i slike. Član je potrebno pozvati prije nego što se `Slika` počne koristiti i kao parametar mu proslijediti pokazivač na objekt klase `Paleta`. Klasa `Slika` će se time konfigurirati tako da će koristiti proslijeđeni objekt za identifikaciju pojedinih boja. Klasa `Ekran` je učinjena prijateljem klase `Slika` primarno zato da bi se omogućio pristup tom zaštićenom funkcijskom članu.

Slijedi niz članova: `DajXRazlucivost()`, `DajYRazlucivost()` te `DajBrojBoja()` koji omogućavaju čitanje podataka o trenutno postavljenom načinu prikaza. I oni su učinjeni virtualnima kako bi se osiguralo da se za svaku posebnu grafičku karticu ispravno odredi traženi podatak. Funkcijski član `PostaviNacinPrikaza()` omogućava naknadnu promjenu načina prikaza tako da mu se kao parametar proslijedi identifikator načina prikaza.

Slijede članovi za crtanje. Član `CrtajTocku()` će nacrtati točku na zadanom mjestu na ekranu tako da mu se kao parametar proslijede koordinate točke i sam objekt klase `Tocka` koji će identificirati točku. Član `CitajTocku()` radi upravo obrnuto: uz zadane koordinate točke on će vratiti objekt klase `Tocka` koji će identificirati točku na

zadanom mjestu. Član `Linija` iscrtava liniju tako da mu se zadaju koordinate početne i završne točke i indeks boje iz palete. Član `Kvadrat()` će nacrtati kvadrat tako da mu se zadaju koordinate gornjeg lijevog i donjeg desnog kuta te indeks boje iz palete.

Klasa `Ekran` služi objedinjavanju rada slike i palete. Ona će sadržavati konstruktor u kojemu će se stvoriti po jedan objekt klase `Slika` i jedan klase `Paleta`. Javno sučelje klase će sadržavati samo dva funkcijska člana pomoću kojih će se moći dobiti referenca na objekt `Slika` i na objekt `Paleta`, tako da se može pristupati njihovim funkcijskim članovima. Evo deklaracije:

```
class Ekran {
public:

    Ekran();
    virtual ~Ekran();

    Slika &DajSliku();
    Paleta &DajPaletu();

};
```

Pomoæne klase `Mis` i `Tipkovnica` takoðer služe tome da se sklop raèunala uèini apstraktnim te prikaže pomoæu objekta. Klasa `Mis` mora sadržavati funkcijske èlanove za inicijalizaciju i èitanje položaja miša. Klasa `Tipkovnica` æe sadržavati funkcijske èlanove za èitanje kôdova tipki koje su pritisnute. Kao i prilikom razvoja klase `Slika`, možemo ustanoviti da postoje razne vrste miševa i tipkovnica, pa æemo funkcijske èlanove definirati virtualnima, a u izvedenim klasama navesti kôd za rad s konkretnim sklopom. Toèno i precizno formiranje suèelja tih klasa previše bi zadiralo u podruèje sistemskog programiranja, pa æemo njihove deklaracije izostaviti i zadovoljiti se gornjim opisom.

Na sličan način bi se definirala sučelja i preostalih klasa `Prozor`, `UokvireniProzor`, `Izbornik` i `Prometnik`. Definirati sučelje tih klasa ne bi bilo vrlo jednostavno iz razloga što uopće nismo dovoljno precizno definirali svojstva tih klasa. Koje su sve operacije dozvoljene na prozoru? Kako on odgovara na pojedinu operaciju? Kako izgleda prozor na ekranu? Kakav operacijski sustav se koristi na našem računalu? Sve su to stvari o kojima treba voditi računa prilikom izrade biblioteke grafičkog sučelja. Zbog toga ćemo ovdje završiti s definicijom sučelja, a daljnju razradu problema prepustiti nadobudnim čitateljima.

19.9. Korak 6: Implementacija

Naposljetku, navedene klase je potrebno implementirati, što znaèi dodati potrebne podatkovne èlanove, napisati kôd funkcijskih èlanova, ugraditi željene algoritme i slièno. Ako smo postavili dobar temelj u prethodnim koracima, sama implementacija ne bi trebala prouzroèiti velike probleme.

Gornju opasku nipošto ne treba shvatiti na način da će implementacija biti sama po sebi jednostavna; složenost implementacije će direktno ovisiti o složenosti samog modela koji želimo opisati pomoću objekata. Ono što smo mislili reći gornjom tvrdnjom jest da će biti jasno što se želi napraviti. Naime, ako smo prošli dosljedno i temeljito kroz prethodne korake, sada imamo već dobru sliku o tome što koji objekt radi, pa čak i intuitivnu predodžbu o tome kako on to radi. No još uvijek nemamo garanciju da ako smo kvalitetno proveli prethodnih pet koraka, da će i implementacija biti kvalitetna. Naprotiv, sada je sa stanovišta sučelja i željenog cilja potrebno izabrati adekvatan algoritam koji će osigurati željene performanse sustava.

Prilikom pisanja implementacije važno je voditi računa o tome da se poštuje princip skrivanja informacija. Korisnik klase mora znati što manje o ustrojstvu klase, važno je javno sučelje. Zbog toga, kvalitetna implementacija će voditi računa o tome da što više skrije svoje detalje. U pisanju vaših C++ programa vodite se narodnom uzrečicom: *skrivajte implementaciju kao zmija noge*.

Osnovno što se pod time podrazumijeva jest skrivanje podatkovnih članova. Podatkovni članovi su najosjetljiviji dio svakog objekta, jer oni zapravo predstavljaju unutarnje stanje objekta. Svaki objekt treba osiguravati svoj integritet, što znači da će se svaka promjena njegovog stanja obavljati tako da objekt u svakom trenutku bude suvisao. Ako se dodjeljuje vrijednost pojedinom podatkovnom članu, potrebno je obaviti provjeru je li ta vrijednost unutar dozvoljenih granica. Zbog toga se sve dodjele podatkovnim članovima u principu obavljaju unutar funkcijskih članova koji obavljaju sve potrebne provjere.

Ako bi podatkovni član imao javni pristup, tada bi nesmotreno (ili “prljavo”) napisan program mogao promijeniti vrijednost nekog podatkovnog člana tako da objekt više ne bi imao smisla. To bi dalje moglo ugroziti funkcioniranje objekta, a preko njega i funkcioniranje čitavog sustava.

Prilog

A. Standardna biblioteka

Dođite i odaberite nešto iz moje biblioteke,
odagnajte tako svoju tugu.

William Shakespeare, "Titus Andronicus" (1590)

ANSI standard jezika C++ prvenstveno opisuje sintaksu naredbi jezika. Međutim, zbog što bolje prenosivosti kôda, ANSI komitet za standardizaciju je odluèio u standard ukljuèiti i definicije elemenata standardne biblioteke. Osnovna namjena standardne biblioteke jest pružiti efikasne i pouzdane predloške, klase i funkcije te time oslobodi pisca kôda od napornog pisanja trivijalnih struktura podataka i algoritama. Standardna C++ biblioteka sadrži definicije:

- makro funkcija i makro imena,
- simbolièkih konstanti (vrijednosti),
- tipova,
- predložaka,
- klasa,
- funkcija,
- objekata.

Deklaracije i definicije elemenata standardne C++ biblioteke raspodijeljene su kroz trideset dvije datoteke zaglavlja, navedene u tablici A.1.

Tablica A.1. Zaglavlja standardne C++ biblioteke

`<algorithm>`	`<ios>`	`<map>`	`<stack>`
`<bitset>`	`<iosfwd>`	`<memory>`	`<stdexcept>`
`<complex>`	`<iostream>`	`<new>`	`<streambuf>`
`<deque>`	`<istream>`	`<numeric>`	`<string>`
`<exception>`	`<iterator>`	`<ostream>`	`<typeinfo>`
`<fstream>`	`<limits>`	`<queue>`	`<utility>`
`<functional>`	`<list>`	`<set>`	`<valarray>`
`<iomanip>`	`<locale>`	`<sstream>`	`<vector>`

Èak ne posebno pažljiv èitatelj æe sigurno zamijetiti kako nazivi standardnih datoteka nemaju nastavak `.h` koji smo spominjali kroz cijelu knjigu. Vjerujte, i mi smo ostali iznenađeni kada smo otvorili C++ standard i vidjeli gornju tablicu. Nastavak `.h` je jednostavno ukinut! Mnogi C++ prevoditelji nisu još ažurirali nove nazive datoteka, a mi iskreno sumnjamo da æe mnogi to ikad uèiniti (jer æe propasti kada im razjareni programeri, kojima se njihovi postojeæi programi od nekoliko stotina tisuæa linija izvornog kôda neæe moæi prevesti, pošalju 100000 e-mail protestnih poruka). Mi æemo

samo u Prilogu koristiti ovakve nazive datoteka zaglavlja (jer ih prevoditelji još ne podržavaju niti nam se da prolaziti kroz knjigu i mijenjati ih), a vama prepuštamo da saznate da li prevoditelj kojeg koristite doslovce prati standard.

Standardna biblioteka C jezika je uključena i u C++ standard, s time da su deklaracije i definicije koje ta biblioteka traži smještene u zasebne datoteke zaglavlja. Sve C datoteke zaglavlja su navedene u tablici A.2.

Tablica A.2. C++ zaglavlja za C biblioteke

`<cassert>`	`<climits>`	`<cstardg>`	`<ctime>`
`<cctype>`	`<clocale>`	`<cstddef>`	`<cwchar>`
`<cerrno>`	`<cmath>`	`<cstdio>`	`<cwtype>`
`<cfloat>`	`<csetjmp>`	`<cstdlib>`	
`<ciso646>`	`<csignal>`	`<cstring>`	

Postoji znaèajna razlika u odnosu na standard C jezika. Naime, svi nazivi C datoteka zaglavlja imaju prefiks *c*, kako bi se naznaèilo da dotièna datoteka pripada C standardu. Na primjer, sadržaj zaglavlja `cmath` odgovara sadržaju C zaglavlja `math.h`. Takoðer, standard C++ jezika dozvoljava i ukljuèivanje starih datoteka zaglavlja koje imaju nastavak `.h`. Razlika je u tome što su u datotekama koje poèinju sa *c* svi identifikatori smješteni u imenik `std`, èime su uklonjeni iz globalnog podruèja imena. Radi kompatibilnosti preporuèuje se korištenje novih datoteka. Krasno, još samo da su nam dostupne...

Prema namjeni pojedinih komponenti, standardna biblioteka podijeljena je u deset kategorija:

- podrška C++ jeziku (*language support*), koja ukljuèuje komponente za dinamièko rukovanje memorijom (zaglavlje `new`), dinamièku identifikaciju tipa (`typeinfo`) te rukovanje iznimkama (`exception`). U ovu kategoriju ukljuèene su i komponente iz pet C zaglavlja: `cstdarg`, `csetjmp`, `csignal`, `ctime` i `cstdlib`.
- dijagnostika, koja obuhvaæa komponente za otkrivanje i prijavu pogrešaka, definirane u zaglavlju `stdexcept`. Te komponente obuhvaæaju iznimke koje se bacaju u sluèaju pogreške. Ona takoðer ukljuèuje standardne C komponente iz zaglavlja `cassert` i `cerrno`.
- opæe pomoæne komponente (*general utilities*), koje obuhvaæaju komponente i funkcije iz jezgre standardnih klasa zadanih predlošcima (*standard template library*, *STL*), komponente za dinamièko rukovanje memorijom, te funkcije za rukovanje vremenom i datumom. Ove komponente su opisane u zaglavljima `utility` i `memory`, te standardnim C zaglavljima `cstring`, `cstdlib` i `ctime`.
- nizovi. U ovu kategoriju ukljuèene su komponente za rukovanje nizovima znakova tipa `char`, `wchar_t` ili nekog drugog tipa, deklarirane u zaglavlju `string`. Osnovu èini klasa `basic_string` definirana predloškom, iz kojeg su instancirane klase `string` i `wstring`. Dostupne su i komponente iz C zaglavlja `cctype`, `cwctype`, `cstring`, `cwchar`, te višebajtne konverzije iz `cstdlib`.

- mjesne zemljopisne (*localization*) komponente deklarirane u zaglavlju `locale`, te u standardnom C zaglavlju `clocale`.
- kontejneri. U ovu kategoriju su smještene komponente kakve nisu postojale u jeziku C, odnosno njegovim bibliotekama. Te su komponente opisane u osam zaglavlja: `bits`, `deque`, `list`, `queue`, `stack`, `vector`, `map` i `set`.
- iteratori, deklarirani u zaglavlju `iterator`, èine zasebnu kategoriju komponenti koje omoguæavaju iteraciju kroz kontejnere, tokove i meðuspremnike tokova.
- algoritmi. U ovoj kategoriji se nalaze komponente neophodne za algoritamske operacije na kontejnerima i drugim nizovima. Opisane su u zaglavlju `algorithm`, a pridružene su im i funkcije `bsearch()` i `qsort()` iz C zaglavlja `cstdlib`.
- numerièke komponente, namijenjene za obavljanje numerièkih operacija. Sadrži komponente za kompleksni tip podataka, numerièka polja, te opæe numerièke algoritme. Deklarirane su u zaglavljima `complex`, `valarray` i `numeric`, te u C zaglavlju `cmath`.
- ulazno-izlazne komponente, tj. ulazno-izlazni tokovi, deklarirani u zaglavljima `iosfwd`, `iostream`, `ios`, `streambuf`, `istream`, `ostream`, `iomanip`, `sstream`, `fstream`, kao i standardnim C zaglavljima `cstdio`, te `cwchar`.

U nastavku æemo obraditi važnije komponente iz standardnih biblioteka. Nije nam namjera obuhvatiti kompletne standardne biblioteke. To bi zahtijevalo još barem ovoliko stranica. Takoðer, mnoge funkcije i klase nikada neæete koristiti – u standardnu biblioteku je zaista ubaèeno mnogo toga. Mnoge funkcije su posljedica podržavanja specifiènih operacijskih sustava (na primjer, signali su preuzeti s UNIX sustava i nemaju direktnu podršku u tom obliku u drugim sustavima). Zatim, mnoge funkcije, na primjer, podrška za lokalizaciju programa, specifiène su za pojedini sustav. Programer koji želi napisati program usko povezan s operacijskim sustavom koristit æe lokalizacijska svojstva ugraðena u sam sustav. Zbog svega toga, nakon što nauèite C++, odluèite se za operacijski sustav i nabavite knjigu koja æe opisati kako pozivati usluge tog sustava. To æe vam biti daleko korisnije nego prouèavanje standardne biblioteke. No ipak, neke stvari iz standardne biblioteke, kao što su kontejnerske klase i neke standardne funkcije (èesto upravo one naslijeðene iz jezika C), mogu biti od koristi. Mi æemo u nastavku prikazati ono što smatramo bitnim i što se može smjestiti na razuman broj stranica. Za detaljni opis tih klasa èitatelja upuæujemo na ANSI Standard jezika C++, na knjigu: P. J. Plauger, *The Draft Standard C++ Library* (Prentice-Hall, ISBN 0-13-117003-1, 1995) ili na neki od èlanaka u èasopisu *C++ Report*. Osim toga, mnogi prevoditelji još nemaju podržane sve te biblioteke, pa tako prije korištenja, provjerite dokumentaciju prevoditelja.

A.1. Standardne makro funkcije i makro imena

U tablici A.3 navedene su makro funkcije i imena koje standard jezika C++ podržava. Imena iza kojih slijedi naziv zaglavlja unutar zagrada <> definirana su u svim tim zaglavljima, s time da su te definicije meðusobno ekvivalentne. Sve su makro funkcije i

imena naslijeđena iz programskog jezika C, a veæinu standard podržava iskljuèivo zbog kompatibilnosti sa C-programima.

Tablica A.3. Standardne makro funkcije i imena

assert	LC_CTYPE	SEEK_END	stdout
BUFSIZ	LC_MONETARY	SEEK_SET	TMP_MAX
CLOCKS_PER_SEC	LC_NUMERIC	setjmp	va_arg
EDOM	LC_TIME	SIGABRT	va_end
EOF	L_tmpnam	SIGFPE	va_start
ERANGE	MB_CUR_MAX	SIGILL	WCHAR_MAX
errno	NULL <cstddef>	SIGINT	WCHAR_MIN
EXIT_FAILURE	NULL <cstdio>	SIGSEGV	WEOF <cwchar>
EXIT_SUCCESS	NULL <cstring>	SIGTERM	WEOF <cwctype>
FILENAME_MAX	NULL <ctime>	SIG_DFL	_IOFBF
FOPEN_MAX	NULL <cwchar>	SIG_ERR	_IOLBF
HUGE_VAL	offsetoff	SIG_IGN	_IONBF
LC_ALL	RAND_MAX	stderr	
LC_COLLATE	SEEK_CUR	stdin	

Slijede opisi nekih važnijih makro imena i funkcija:

#include <cassert> // stari naziv: <assert.h>
void assert(int test);

Makro funkcija koja testira uvjet `test`. Ako je rezultat testa nula, prekida se izvođenje programa pozivom funkcije `abort()`, a na jedinici za ispis pogreške ispisuje se poruka s nazivom datoteke izvornog kôda i brojem linije u kojoj je pogreška nastupila.

Djelovanje makro funkcije `assert()`se može isključiti tako da se ispred pretprocesorske naredbe za uključivanje zaglavlja `cassert` (ili `assert.h`) definira makro ime `NDEBUG` (naredbom `#define NDEBUG`).

#include <cerrno> // stari naziv: <errno.h>
#include <cmath> // stari naziv: <math.h>
EDOM, ERANGE

`EDOM` kôd za pogrešku domene funkcije.
`ERANGE` kôd pogreške za rezultat izvan opsega.

Gornja dva makro imena koriste se za indikaciju pogreške prilikom poziva matematièkih funkcija. Matematièke funkcije prilikom svog izvođenja postavljaju globalni objekt `errno` èime signaliziraju je li funkcija ispravno obavila svoj posao. Ako prilikom poziva funkcije navedemo vrijednost za koju funkcija nije definirana (na primjer, pokušamo izvaditi logaritam iz negativnog broja), u `errno` æe se upisati `EDOM`, a ako je rezultat funkcije uzrokuje preljev, upisat æe se `ERANGE`.

#include <cstdio>   // stari naziv: <stdio.h>
EOF

Znakovna konstanta `EOF` (vrijednost 26 dekadski). U tekstovnim datotekama taj znak oznaèava kraj datoteke (*End-Of-File*).

#include <cerrno>   // stari naziv: <errno.h>
errno

Globalna cjelobrojna varijabla, koja služi za pohranjivanje kôda pogreške. Èesto se koristi kao provjera kod poziva matematièkih funkcija, koje mu u sluèaju pogreške pridružuju vrijednost `EDOM` ili `ERANGE`.

#include <cstdlib>   // stari naziv: <stdlib.h>
EXIT_FAILURE, EXIT_SUCCESS

`EXIT_FAILURE` nepravilan prekid programa.
`EXIT_SUCCESS` normalan završetak programa.

Konstante koje se navode kao argumenti za funkciju `exit()`.

#include <cmath>   // stari naziv: <math.h>
HUGE_VAL

`HUGE_VAL` preljev vrijednosti matematièkih funkcija.

Ovu vrijednost vraæaju matematièke funkcije kada signaliziraju da je povratna vrijednost prevelika za ugraðeni opseg brojeva.

#include <cstddef>   // stari naziv: <stddef.h>
#include <cstdio>   // stari naziv: <stdio.h>
#include <cstring>   // stari naziv: <string.h>
#include <ctime>   // stari naziv: <time.h>
#include <cwchar>   // stari naziv: <wchar.h>
NULL

Vrijednost nul-pokazivaèa.

#include <cstdlib>   // stari naziv: <stdlib.h>
RAND_MAX

Najveæa vrijednost koju može vratiti funkcija `rand()`. Funkcija `rand()` služi generiranju pseudo-sluèajnih brojeva.

```
#include <cstdarg>                    // stari naziv: <stdarg.h>
void va_start(va_list ap, lastfix);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

`va_start()` postavlja polje `ap` (tipa `va_list` koji je takoðer definiran u istom zaglavlju) tako da pokazuje na prvi neodreðeni argument proslijeðen funkciji. Drugi argument je ime zadnjeg fiksnog argumenta koji se prenosi funkciji s neodreðenim argumentima. `va_start()` se mora pozvati prije `va_arg()` i `va_end()`.

`va_arg()` vraæa sljedeæi argument iz liste `ap` stvarno proslijeðenih argumenata.

`va_end()` osigurava regularan povratak iz funkcije s neodreðenim argumentima. `va_end()` treba obavezno pozvati nakon što su makro funkcijom `va_arg()` oèitani željeni proslijeðeni argumenti.

Gornje makro funkcije služe za dohvaæanje argumenata funkcija s neodreðenim brojem argumenata. Primjer primjene ovih makro funkcija dan je u odsjeèku 5.4.8 posveæenom funkcijama s neodreðenim argumentima.

A.2. Standardne vrijednosti

Jezik C++ nasljeðuje 45 standardnih vrijednosti iz standardne biblioteke jezika C, navedenih u tablici A.4. Vrijednosti definiraju duljine, najveæe i najmanje vrijednost za pojedine ugraðene tipove. Za cjelobrojne tipove navedene su u zaglavlju `limits.h`, a za realne u zaglavlju `float.h`.

Tablica A.4. Standardne vrijednosti

CHAR_BIT	FLT_DIG	INT_MIN	MB_LEN_MAX
CHAR_MAX	FLT_EPSILON	LDBL_DIG	SCHAR_MAX
CHAR_MIN	FLT_MANT_DIG	LDBL_EPSILON	SCHAR_MIN
DBL_DIG	FLT_MAX	LDBL_MANT_DIG	SHRT_MAX
DBL_EPSILON	FLT_MAX_10_EXP	LDBL_MAX	SHRT_MIN
DBL_MANT_DIG	FLT_MAX_EXP	LDBL_MAX_10_EXP	UCHAR_MAX
DBL_MAX	FLT_MIN	LDBL_MAX_EXP	UINT_MAX
DBL_MAX_10_EXP	FLT_MIN_10_EXP	LDBL_MIN	ULONG_MAX
DBL_MAX_EXP	FLT_MIN_EXP	LDBL_MIN_10_EXP	USHRT_MAX
DBL_MIN	FLT_RADIX	LDBL_MIN_EXP	
DBL_MIN_10_EXP	FLT_ROUNDS	LONG_MAX	
DBL_MIN_EXP	INT_MAX	LONG_MIN	

A.3. Standardni tipovi

Standardna C++ biblioteka definira tipove navedene u tablici A.5. Neki od njih su naslijeðeni iz jezika C. Veæina tipova vezanih uz tokove opisana je u poglavlju XX ove knjige. Neke od preostalih tipova upoznat æemo kroz opise funkcija koje ih koriste.

Tablica A.5. Standardni tipovi

clock_t	ptrdiff_t <cstddef>	wctype_t
div_t	sig_atomic_t	wfilebuf
FILE	size_t <cstddef>	wifstream
filebuf	size_t <cstdio>	wint_t <cwchar>
fpos_t	size_t <cstring>	wint_t <cwctype>
ifstream	size_t <ctime>	wios
ios	streambuf	wistream
istream	streamoff	wistringstream
istringstream	streampos	wofstream
jmp_buf	string	wostream
ldiv_t	stringbuf	wostringstream
mbstate_t	terminate_handler	wstreambuf
new_handler	time_t	wstreampos
ofstream	unexpected_handler	wstring
ostream	va_list	wstringbuf
ostringstream	wctrans_t	

A.4. Standardne klase i strukture

Posebno mjesto u C++ biblioteci zauzima standardna biblioteka predložaka (*Standard Template Library*, *STL*), tablica A.6. Radi se o 66 raznovrsnih klasa koje, između ostalog, uključuju klasu znakovnih nizova, kontejnerske klase, klasu kompleksnih brojeva, klasu polja brojčanih vrijednosti. Ovdje ćemo samo ukratko opisati te klase – potpuni opis svih tih klasa iziskivao bi još stotinjak stranica teksta.

Kontejneri su općenito namijenjeni za pohranjivanje drugih objekata. Oni kontroliraju alokaciju i dealokaciju tih objekata pomoću konstruktora, destruktora, te operatora umetanja i brisanja. Kontejneri se mogu podijeliti u dvije grupe: nizove i asocijativne kontejnere.

Nizovi slažu konačan broj elemenata istog tipa u strogo linearnom rasporedu. Kontejnerska biblioteka pruža na raspolaganje tri tipa nizovnih kontejnera: `vector`, `list` i `deque` (deklarirana u svojim pripadajućim zaglavljima, vidi tablicu A.1). Klasa `vector` je kontejner koji se koristi u najopćenitijim slučajevima – ona omogućava proizvoljan pristup pojedinim članovima te jednostavna dodavanja i brisanja elemenata na kraju niza; za dodavanja i brisanja unutar niza potrebno vrijeme raste linearno s udaljenošću elementa od kraja niza. Klasa `list` je pogodna u primjenama kada često dodajemo i brišemo elemente niza. Trajanje operacija umetanja i brisanja elemenata ne ovisi o njihovom položaju u nizu, ali se elementi (za razliku od `vector` ili `deque`) ne mogu dohvaćati izravno. Klasa `deque` je najprikladnija za nizove u kojima se dodavanje ili oduzimanje elemenata obavlja na početku ili kraju; operacije dodavanja i brisanja elemenata unutar niza iziskuju vrijeme proporcionalno udaljenosti od kraja niza.

Asocijativni kontejneri omogućavaju dohvaćanje članova prema nekom ključu. Biblioteka pruža na raspolaganje četiri osnovna tipa asocijativnih kontejnera: `set`, `multiset`, `map` i `multimap` (deklarirana u zaglavljima `set` i `map`). `set` je kontejner

Tablica A.6. Standardna biblioteka predložaka

allocator	map
auto_ptr	mask_array
back_insert_iterator	messages
basic_filebuf	messages_byname
basic_fstream	moneypunct
basic_ios	moneypunct_byname
basic_istream	money_get
basic_istringstream	money_put
basic_ofstream	multimap
basic_ostream	multiset
basic_ostringstream	numeric_limits
basic_streambuf	numpunct
basic_string	num_get
basic_stringbuf	num_put
binary_negate	ostreambuf_iterator
binder1st	ostream_iterator
binder2nd	pointer_to_binary_function
bitset	pointer_to_unary_function
codecvt	priority_queue
codecvt_byname	queue
collate	raw_storage_iterator
collate_byname	reverse_bidirectional_iterator
complex	reverse_iterator
ctype	set
ctype_byname	slice_array
deque	stack
front_insert_iterator	time_get
gslice_array	time_get_byname
indirect_array	time_put
insert_iterator	time_put_byname
istreambuf_iterator	unary_negate
istream_iterator	valarray
list	vector

Tablica A.7. Standardne strukture definirane predlošcima

bidirectional_iterator	ios_traits	negate
binary_function	less	not_equal_to
divides	less_equal	pair
equal_to	logical_and	plus
forward_iterator	logical_not	random_access_iterator
greater	logical_or	string_char_traits
greater_equal	minus	times
input_iterator	modulus	unary_function

koji svakom članu dodjeljuje jedinstveni ključ (jedan član je vezan s ključem koji je jedinstven na nivou kontejnera) i osigurava brzo dohvaćanje ključa. `multiset` dozvoljava jednake ključeve (više članova mogu imati isti ključ). Nasuprot tome, `map` i `multimap` osiguravaju dohvaćanje podataka nekog drugog tipa prema zadanom ključu.

Klasa `basic_string`, deklarirana u zaglavlju `string`, namijenjena je za rukovanje znakovnim nizovima. Za nju su definirani i neki važniji operatori, poput `+`, `!=`, `==`, `<<` i `>>`, što može značajno pojednostavniti operacije sa znakovnim nizovima.

I na kraju, spomenimo još i klasu `complex` (deklariranu u istoimenom zaglavlju `complex`), koja pojednostavnjuje račun sa kompleksnim brojevima. Za nju su također definirani svi aritmetički operatori i operatori za ispis na tok (`!=`, `*`, `*=`, `+`, `+=`, `-`, `-=`, `/`, `/=`, `<<`, `==`, `>>`) i funkcije (npr. `abs()`, `acos()`, `exp()`, `log()`) koji su dozvoljeni s kompleksnim brojevima.

Osim klasa definiranih predlošcima, standardna biblioteka sadrži i strukture definirane predlošcima (tablica A.7), standardne klase (tablica A.8) i standardne strukture (tablica A.9). Standardne klase (npr. `bad_alloc`, `bad_cast`, `domain_error`) uglavnom su namijenjene za hvatanje iznimki.

Tablica A.8. Standardne klase

bad_alloc	ctype_byname<char>	logic_error
bad_cast	domain_error	out_of_range
bad_exception	exception	overflow_error
bad_typeid	gslice	range_error
basic_string<char>	invalid_argument	runtime_error
basic_string<wchar_t>	ios_base	slice
complex<double>	length_error	type_info
complex<float>	locale	vector<bool,allocator>
complex<long double>	locale::facet	
ctype<char>	locale::id	

Tablica A.9. Standardne strukture

bidirectional_iterator_tag	money_base::pattern
codecvt_base	nothrow
ctype_base	output_iterator
forward_iterator_tag	output_iterator_tag
input_iterator_tag	random_access_iterator_tag
ios_traits<char>	string_char_traits<char>
ios_traits<wchar_t>	string_char_traits<wchar_t>
lconv	time_base
money_base	tm <ctime>

B. Standardne funkcije

Franklin: 'Jeste li ikada, gospodine Ravnatelju,
pomislili da su vaši standardi možda malo zastarjeli?'
Ravnatelj: 'Naravno da su zastarjeli. Standardi su
uvijek zastarjeli. To je ono što ih čini standardima.'

Alen Bennet, engleski glumac i pisac,
"Forty Years On" (1969)

Standardna C++ biblioteka pruža 208 standardnih funkcija iz C biblioteke (tablica B.1). Veæina matematièkih funkcija je preoptereæena za objekte klase `complex`. Takoðer,

Tablica B.1. Standardne funkcije

abort	fgetpos	gmtime	log10	rewind	strtok	wcscspn
abs	fgets	isalnum	longjmp	scanf	strtol	wcsftime
acos	fgetwc	isalpha	malloc	setbuf	strxfrm	wcslen
asctime	fgetws	iscntrl	mblen	setlocale	swprintf	wcsncat
asin	floor	isdigit	mbrlen	setvbuf	swscanf	wcsncmp
atan	fmod	isgraph	mbrtowc	signal	system	wcsncpy
atan2	fopen	islower	mbsinit	sin	tan	wcspbrk
atexit	fprintf	isprint	mbsrtowcs	sinh	tanh	wcsrchr
atof	fputc	ispunct	mbstowcs	sprintf	time	wcsrtomba
atoi	fputs	isspace	mbtowc	sqrt	tmpfile	wcsspn
atol	fputwc	isupper	memchr	srand	tmpnam	wcsstr
bsearch	fputws	iswalnum	memcmp	sscanf	tolower	wcstod
btowc	fread	iswalpha	memcpy	strcat	toupper	wcstok
calloc	free	iswcntrl	memmove	strchr	towctrans	wcstol
ceil	freopen	iswctype	memset	strcmp	towlower	wcstomba
clearerr	frexp	iswdigit	mktime	strcoll	towupper	wcstoul
clock	fscanf	iswgraph	modf	strcpy	ungetc	wcsxfrm
cos	fseek	iswlower	perror	strcspn	ungetwc	wctob
cosh	fsetpos	iswprint	pow	strerror	vfwprintf	wctomb
ctime	ftell	iswpunct	printf	strftime	vprintf	wctrans
difftime	fwide	iswspace	putc	strlen	vscanf	wctype
div	fwprintf	iswupper	puts	strncat	vsprintf	wmemchr
exit	fwrite	iswxdigit	putwc	strncmp	vswprintf	wmemcmp
exp	fwscanf	isxdigit	putwchar	strncpy	vwprintf	wmemcpy
fabs	getc	labs	qsort	stroul	wcrtomb	wmemmove
fclose	getchar	ldexp	raise	strpbrk	wcscat	wmemset
feof	getenv	ldiv	rand	strrchr	wcschr	wprintf
ferror	gets	localeconv	realloc	strspn	wcscmp	wscanf
fflush	getwc	localtime	remove	strstr	wcscoll	
fgetc	getwchar	log	rename	strtod	wcscpy	

C++ standard definira posebne tipove kojima se podržava vrlo brza paralelna obrada numerièkih podataka na jakim strojevima. Za tu svrhu uveden je predložak valarray koji definira polje matematièkih vrijednosti. Parametar predloška je tip vrijednosti, pa æemo tako polje realnih brojeva dobiti kao valarray<float>, a polje kompleksnih brojeva sa valarray<complex>. Za pojedine funkcije su u matematièkim bibliotekama definirani i predlošci funkcija koje barataju poljem valarray. Tako postoji poseban predložak, primjerice, funkcije sqrt(), koji kao parametar uzima valarray te raèuna korijen iz pojedine vrijednosti. Definirani su i predlošci operatora koji omoguæavaju raèunanje s poljima vrijednosti istog tipa. Na taj naèin C++ standard omoguæava da se za pojedinu raèunalnu platformu optimizira raèunanje s poljem podataka na najprikladniji naèin (pojedine arhitekture procesora posjeduju zaseban set instrukcija za obradu polja numerièkih podataka), tako da se specijalizira pojedini predložak za odreðeni tip.

Slijedi kratak opis važnijih funkcija, grupiranih prema operacijama koje izvode. Budući da je većina funkcija preuzeta iz standardne C biblioteke, pri navođenju naredbe za uključivanje, osim imena zaglavlja C++ biblioteka, navedena su i imena zaglavlja iz C biblioteke. Njih se može prepoznati po nastavku .h. Ovo je praktično ako su korisniku na raspolaganju starije biblioteke. Iz opisa su izuzete standardne C funkcije za ulazno-izlazne tokove, budući da tokovi iz iostream biblioteke (opisane u poglavlju 16) podržavaju sve operacije koje su na raspolaganju C programerima u stdio.h biblioteci. Štoviše, iostream pruža mnoštvo dodatnih pogodnosti pa vjerujemo da, kada se jednom saživi sa iostream bibliotekom, programeru uopće neće nedostajati C-funkcije printf(), scanf() i slične. Isto tako, iskusni C++ programer će vrlo rijetko posegnuti za C funkcijama za alokaciju i oslobađanje dinamički alocirane memorije (malloc(), calloc(), free()), budući da jezik C++ ima za to ugrađene operatore new i delete.

B.1. Funkcije vezane uz znakove i znakovne nizove

#include <locale>
template<class charT> bool isalnum(charT znak, const locale &lc) const;
template<class charT> bool isalpha(charT znak, const locale &lc) const;
template<class charT> bool iscntrl(charT znak, const locale &lc) const;
template<class charT> bool isdigit(charT znak, const locale &lc) const;
template<class charT> bool isgraph(charT znak, const locale &lc) const;
template<class charT> bool islower(charT znak, const locale &lc) const;
template<class charT> bool isprcharT(charT znak, const locale &lc) const;
template<class charT> bool ispunct(charT znak, const locale &lc) const;
template<class charT> bool isspace(charT znak, const locale &lc) const;
template<class charT> bool isupper(charT znak, const locale &lc) const;
template<class charT> bool isxdigit(charT znak, const locale &lc) const;

Ove funkcije služe za klasifikaciju znaka. Argument znak je kôd znaka kojeg želimo klasificirati. Funkcije vraæaju true ako je zadovoljen uvjet ispitivanja; u protivnom je

povratna vrijednost `false`. Buduæi da su funkcije deklarirane pomoæu predložaka, `charT` može biti bilo koji tip koji se dade svesti na `char` ili `wchar_t`. Drugi argument je referenca na objekt klase `locale` koja specificira mjesne (zemljopisne) parametre, kao što su klasifikacija znakova, formati zapisa brojeva, datuma i vremena i sl. Time je (teoretski) omoguæena ispravna klasifikacija naših dijakritièkih znakova – jedino je potrebno podesiti objekt klase `locale`.

`isalnum()` je li `znak` slovo ili dekadska znamenka.
`isalpha()` je li `znak` slovo.
`iscntrl()` je li `znak` neki kontrolni znak (ASCII kôdovi 0...31 i 127).
`isdigit()` je li `znak` dekadska znamenka (`'0'`...`'9'`).
`isgraph()` je li `znak` znak koji se ispisuje (slovo, broj i interpunkcija), bez bjeline `' '`.
`islower()` je li `znak` malo slovo (`'a'`...`'ž'`).
`isprint()` je li `znak` znak koji se ispisuje (slovo, broj i interpunkcija), ukljuèujuæi bjelinu `' '`.
`ispunct()` je li `znak` neki znak za interpunkciju (`'.'`, `','`, `';'`, `':'`...).
`isspace()` je li `znak` praznina (bjelina `' '`, tabulator `'\t'`, novi redak `'\n'`, povrat `'\r'`, pomak papira `'\f'`).
`isupper()` je li `znak` veliko slovo (`'A'`...`'Ž'`).
`isxdigit()` je li `znak` heksadekadska znamenka (`'0'`...`'9'`, `'a'`...`'f'`, `'A'`... `'F'`).

#include <cctype> // stari naziv: <ctype.h>
int isalnum(int znak);
int isalpha(int znak);
int iscntrl(int znak);
int isdigit(int znak);
int isgraph(int znak);
int islower(int znak);
int isprint(int znak);
int ispunct(int znak);
int isspace(int znak);
int isupper(int znak);
int isxdigit(int znak);

Ove funkcije su naslijeðene iz standardne C biblioteke, gdje su deklarirane u zaglavlju `ctype.h`, a povratna vrijednost je tipa `int`. Ove funkcije ne primaju kao parametar objekt klase `locale`, nego se zemljopisna regija odreðuje standardnom funkcijom `setlocale()`. Pojedine funkcije imaju isto znaèenje kao i nove C++ verzije, pa za objašnjenje pogledajte prethodnu grupu funkcija.

#include <cstring> // stari naziv: <string.h>
char *strcat(char *pocetak, const char *nastavak);
char *strncat(char *pocetak, const char *nastavak, size_t maxDuljina);

Funkcija `strcat()` nadovezuje kopiju niza `nastavak` na niz `pocetak`. Funkcija `strncat()` radi to isto, ali se preslikava do najviše `maxDuljina` znakova iz niza `nastavak`. Obje funkcije dodaju zakljuèni nul-znak na kraj "skrpanog" niza `pocetak`, a kao rezultat vraæaju pokazivaè na `pocetak`. Prilikom korištenja valja paziti da je prije poziva funkcije alociran dovoljan prostor za produljeni niz `pocetak`, jer funkcija ne provodi nikakve provjere.

#include <cstring> // stari naziv: <string.h>
const char *strchr(const char *niz, int znak);
char *strchr(char *niz, int znak);

Funkcija u nizu `niz` traži prvu pojavu znaka `znak`. Ako znak nije pronaðen, kao rezultat se vraæa nul-pokazivaè. U podruèje pretraživanja ukljuèen je i zakljuèni nul-znak.

#include <cstring> // stari naziv: <string.h>
int strcmp(const char *niz1, const char *niz2);
int strncmp(const char *niz1, const char *niz2, size_t maxDuljina);
int strcoll(char char *niz1, char *niz2);

Funkcija `strcmp()` uspoređuje `niz1` i `niz2`, znak po znak, sve dok su odgovarajuæi znakovi meðusobno jednaki ili dok ne naiðe na zakljuèni nul-znak. Rezultat usporedbe je:

`< 0` ako je `niz1` manji od `niz2` (svrstano po abecedi `niz1` dolazi prije `niz2`),
`0` ako su `niz1` i `niz2` meðusobno jednaki,
`> 0` ako je `niz1` veæi od `niz2` (svrstano po abecedi `niz1` dolazi iza `niz2`).

Usporedba se radi po slijedu znakova koji se koristi na dotiènom raèunalu. Valja uoèiti da u najèešæem ASCII slijedu sva velika slova prethode malim slovima.

Funkcija `strncmp()` radi jednaku usporedbu kao i `strcmp()`, ali uspoređuje samo do najviše `maxDuljina` znakova.

Funkcija `strcoll()` radi usporedbu u skladu sa tekućom postavom zemljopisne regije. Ta postava se može podesiti funkcijom `setlocale()`, pri čemu se kao kategorija postavke navodi `LC_COLLATE`. Za detaljnije objašnjenje pogledajte opis funkcije `setlocale()`.

#include <cstring> // stari naziv: <string.h>
char *strcpy(char *odrediste, const char *izvornik);
char *strncpy(char *odrediste, const char *izvornik, size_t maxDuljina);

Funkcija `strcpy()` preslikava sadržaj niza `izvornik` na mjesto gdje pokazuje `odrediste`. Preslikavanje se prekida nakon što se prenese zakljuèni nul-znak. Prije

poziva funkcije treba alocirati dovoljan prostor za preslikani niz, jer funkcija ne provjerava je li za preslikani niz na mjestu `odrediste` odvojeno dovoljno mjesta.

Funkcija `strncpy()` preslikava do najviše `maxDuljina` znakova niza `izvornik` na mjesto `odrediste`. Treba uočiti da, ako je `maxDuljina` manja ili jednaka duljini niza `izvornik`, nul-znak neće biti preslikan pa niz `odrediste` može ostati nezaključen.

#include <cstring> // stari naziv: <string.h>
size_t *strlen(const char *niz);

Funkcija izraèunava i vraæa duljinu niza `niz`, ne raèunajuæi zakljuèni nul-znak.

#include <cstring> // stari naziv: <string.h>
const char *strpbrk(const char *niz, const char *znakovi);
char *strpbrk(char *niz, const char *znakovi);

Funkcija pretražuje niz `niz` do prve pojave bilo kojeg znaka iz niza `znakovi`. Kao rezultat vraæa pokazivaè na prvu pojavu ili nul-pokazivaè ako nijedan znak nije pronaðen.

#include <cstring> // stari naziv: <string.h>
size_t strspn(const char *niz, const char *podNiz);
size_t strcspn(const char *niz, const char *podNiz);

Funkcija `strspn()` traži u nizu `niz` mjesto gdje se spominje prvi znak koji nije naveden u nizu `podNiz`. Kao rezultat ona vraæa indeks pronaðenog znaka. Na primjer, izvoðenjem naredbe

```
cout << strspn("crnac", "nerc") << endl;
```

ispisat æe se broj 3, jer se tek znak `'a'` u prvom nizu ne pojavljuje u drugom nizu, a nalazi se na poziciji 3.

Funkcija `strcspn()` radi upravo suprotno: ona će vratiti indeks prvog znaka niza `niz` koji je sadržan u nizu `podNiz`. Na primjer, donja naredba će ispisati 2, jer je znak `'n'` prvi znak iz drugog niza koji se pojavljuje u prvome te se nalazi na poziciji 2:

```
cout << strspn("crnac", "antimon");
```

#include <cstring> // stari naziv: <string.h>
const char *strstr(const char *niz, const char *podNiz);
char *strstr(char *niz, const char *podNiz);

Funkcija pretražuje `niz` i traži prvu pojavu niza `podNiz` (bez njegovog zaključnog nul-znaka). Rezultat je pokazivaè na poèetak prve pojave niza `podNiz` ili nul-pokazivaè ako `podNiz` nije sadržan u `niz`-u.

#include <cstring> // stari naziv: <string.h>
const char *strrchr(const char *niz, int znak);
char *strrchr(char *niz, int znak);

Funkcija pretražuje `niz` i traži zadnju pojavu znaka `znak`. Funkcija pretražuje `niz` od njegova kraja (ukljuèujuæi i zakljuèni nul-znak). Ako je znak pronaðen, funkcija vraæa pokazivaè na zadnju pojavu znaka u nizu; ako nema `znak`-a u `niz`-u, tada vraæa nul-pokazivaè.

#include <cstring> // stari naziv: <string.h>
char *strtok(char *niz, const char *granicnici);

Funkcija pretražuje `niz` i razbija ga na podnizove koji su razdvojeni nekim od znakova iz niza `granicnici`. Rezultat poziva funkcije je pokazivaè na poèetak sljedeæeg podniza ili nul-pokazivaè ako više nema podnizova.

Želi li se `niz` razbiti na podnizove, prilikom prvog poziva funkcije treba prenijeti pokazivač na taj niz. Funkcija će na kraj podniza staviti zaključni nul-znak. U sljedećim pozivima, za prvi argument se navodi nul-pokazivač, a funkcija ponavlja postupak za ostale podnizove, sve do kraja niza. Podniz `granicnik` se smije mijenjati za pojedine pozive. Valja naglasiti da se pozivom funkcije `strtok()` mijenja sadržaj `niz`-a (graničnici se nadomještaju nul-znakovima). Evo jednostavnog primjera u kojem se `niz` rastavlja na podnizove odvojene znakovima `','` ili `' '`:

```
char niz[] = "I cvrči,cvrči cvrčak";
// u prvom pozivu prosljeđuje se pokazivač na početak niza
char *podniz = strtok(niz, ", ");
if (podniz) {
    do {
        cout << podniz << endl;
        // u daljnjim pozivima prvi argument je nul-pokazivač
        podniz = strtok(NULL, ", ");
    } while(podniz);
}
```

Izvoðenjem gornjeg kôda ispisat æe se:

```
I
cvrči
```

```
cvrči
cvrčak
```

#include <locale>
template <class charT> tolower(charT znak, const locale &loc) const;
template <class charT> toupper(charT znak, const locale &loc) const;
#include <cctype> // stari naziv: <ctype.h>
int tolower(int znak);
int toupper(int znak);

Ako je `znak` veliko slovo, funkcija `tolower()` ga pretvara u njegovo malo slovo (`'A'`...`'Ž'` u `'a'`...`'ž'`). Funkcija `toupper()` èini obrnuto: ako je `znak` malo slovo, pretvara ga u njegovo veliko slovo. Ostale znakove funkcije ostavljaju nepromijenjenima. Povratna vrijednost je kôd (eventualno) pretvorenog znaka. Funkcije iz standardne C++ biblioteke deklarirane su predlošcima, gdje `charT` može biti bilo koji tip koji se može svesti na `char` ili `wchar_t`. Drugi argument jest referenca na objekt klase `locale` koja specificira mjesne (zemljopisne) parametre, kao što su klasifikacija znakova, formati zapisa brojeva, datuma i vremena i sl. Istoimene funkcije iz standardne C biblioteke su tipa `int`.

B.2. Funkcije za međusobne pretvorbe nizova i brojeva

#include <cmath> // stari naziv: <math.h>
double atof(const char *niz);

Pretvara znakovni niz `niz` u realni broj tipa `double`. Broju u znakovnom nizu smiju prethoditi praznine. Broj smije biti zapisan u bilo kom formatu dozvoljenom za realne brojeve, uključujuæi i znanstveni zapis. Prvi nedozvoljeni znak u nizu prekida konverziju. Funkcija vraæa pretvorenu vrijednost ulaznog niza. Ako nastupi brojèani preljev prilikom pretvorbe (primjerice ako je broj u znakovnom nizu prevelik ili premali), funkcija vraæa `HUGE_VAL`, te postavlja globalnu varijablu `errno` na `ERANGE` (*Range Error*).

Funkcija `atof()` je slična funkciji `strtod()`, ali osigurava bolje prepoznavanje pogrešaka. Izvođenjem naredbi:

```
char *niz = " -1.23e-4 take money and run";
double broj = atof(niz);
cout << broj << endl;
```

na zaslonu æe se ispisati broj `-0.000123`.

#include <cstdlib> // stari naziv: <stdlib.h>
int atoi(const char *niz);
long atol(const char *niz);

Funkcija `atoi()` pretvara znakovni niz `niz` u cijeli broj tipa `int`, a funkcija `atol()` u cijeli broj tipa `long`. Broju u znakovnom nizu smiju prethoditi praznine te predznak. Prvi nedozvoljeni znak prekida konverziju. Funkcije ne provjeravaju pojavu preljeva. Kao rezultat vraæaju pretvoreni cijeli broj, a ako se pretvoreni broj ne može svesti na `int`, odnosno `long`, vraæaju 0. Tako æe se izvoðenjem primjera

```
char *niz = "1234.56";
int numera = atoi(niz);
cout << numera << endl;
```

na zaslonu ispisati broj `1234`.

#include <cstdlib> // stari naziv: <stdlib.h>
double strtod(const char *niz, char **pokazKraja);

Funkcija `strtod()` pretvara znakovni niz u broj tipa `double`. U znakovnom nizu `niz` koji sadrži tekstovni prikaz realnog broja, samom broju smiju prethoditi praznine. Broj smije biti prikazan u bilo kojem obliku dozvoljenom za prikaz realnih brojeva, bez praznina unutar broja. Ako je `pokazKraja` prilikom ulaska u funkciju razlièit od nul-pokazivaèa, tada ga funkcija usmjerava na znak koji je prekinuo slijed uèitavanja. Slijed uèitavanja se prekida èim se naiðe na neki znak koji ne može biti sastavni dio broja. Stoga æe se izvoðenjem sljedeæeg primjera:

```
char *niz = " -1.23e-1hahaha", *pokazKraja;
double broj = strtod(niz, &pokazKraja);
// radi ispisa odsijeca se niz iza dozvoljenih znakova
*pokazKraja = '\0';
cout << "Broj " << niz << " je pretvoren u " << broj << endl;
```

na zaslonu ispisati tekst:

```
Broj  -1.23e-1 je pretvoren u 0.123
```

Funkcija `strtod()` slièna je funkciji `atof()`, ali potonja bolje prepoznaje pogreške prilikom pretvorbe.

#include <cstdlib> // stari naziv: <stdlib.h>
long strtol(const char *niz, char **pokazKraja, int baza);

Funkcija pretvara znakovni niz u broj tipa `long`. U znakovnom nizu `niz` koji sadrži tekstovni prikaz cijelog broja koji se želi pretvoriti, samom broju smiju prethoditi praznine. Takoðer, neposredno ispred broja smiju se nalaziti predznak (`'+'` ili `'-'`),

'0' (za sluèaj da niz sadrži oktalni prikaz broja) ili 'x', odnosno 'X' (za sluèaj da niz sadrži heksadekadski prikaz broja). Baza može biti cijeli broj izmeðu ukljuèivo 2 do ukljuèivo 36. Ako je pokazKraja prilikom ulaska u funkciju razlièit od nul-pokazivaèa, tada ga funkcija usmjerava na znak koji je prekinuo slijed uèitavanja. Slijed uèitavanja se prekida èim se naiðe na neki znak koji nije neki od gore spomenutih znakova ili ne može biti znamenka u pripadajuæoj bazi. Na primjer, za heksadekadske brojeve (baza 16) dozvoljeni znakovi su svi dekadski brojevi '0'...'9' te slova 'a'...'f', odnosno 'A'...'F'. Tako æe u sljedeæem primjeru biti uèitane samo prve tri znamenke, jer u oktalnom prikazu znamenka 8 nije dozvoljena:

```
char *niz = " -01188", *pokazKraja;
int baza = 8;
long broj = strtol(niz, &pokazKraja, baza);
// radi ispisa odsijeca niz iza važećih oktalnih znamenki
*pokazKraja = '\0';
cout << "Broj " << niz << " u bazi " << baza << " je "
     << broj << endl;
```

B.3. Funkcije vezane uz vrijeme i datum

Funkcije vezane uz vrijeme i datum definirane su u standardnom zaglavlju ctime (stari naziv time.h). U tom zaglavlju su definirane i dva cjelobrojna tipa: clock_t i time_t, kao i struktura tm. Valja paziti da neke funkcije obraðuju *kalendarsko vrijeme* koje se opæenito razlikuje od *lokalnog vremena* zbog naèina kako se vrijeme pohranjuje na raèunalu ili zbog razlike u vremenskim zonama. Na primjer, na osobnim raèunalima je kalendarsko vrijeme broj sekundi proteklih od 1. sijeènja 1970. u ponoæ po GMT (*Greenwich Mean Time*).

#include <ctime> // stari naziv: <time.h>
char *asctime(const tm *vremenskiBlok);
char *ctime(const time_t *pokVrijeme);

Funkcije pretvaraju vrijeme koje im se prenosi kao argument, u znakovni niz oblika:

```
Wed Jan 15 01:23:45 1997\n\0
```

Za funkciju asctime() argument je struktura tipa tm (definirana u zaglavlju ctime, a opisana na str. 590 uz opis funkcije localtime()), dok je povratna vrijednost znakovni niz. Za funkciju ctime() argument je kalendarsko vrijeme (cjelobrojni tip podatka kakvog vraæa funkcija time()), a povratna vrijednost je znakovni niz u koji je upisano lokalno vrijeme. U sljedeæem primjeru obje naredbe za ispis æe ispisati jednake nizove:

```
time_t vura = time(NULL);
cout << asctime(localtime(&vura));
cout << ctime(&vura);
```

Uoèimo da funkcija na kraj niza, neposredno ispred zaključnog nul-znaka dodaje znak za novi redak `'\n'`, tako da za ispis nije korišten manipulator `endl`.

Budući da je niz koji vraća funkcija `asctime()`, odnosno `ctime()` statička varijabla, on će biti prepisan pri svakom novom pozivu funkcija `asctime()`, odnosno `ctime()` – ako ga želimo sačuvati, valja ga preslikati u neki drugi znakovni niz.

#include <ctime> // stari naziv: <time.h>
clock_t clock(void);

Funkcija kao rezultat vraæa procesorsko vrijeme proteklo od poèetka izvođenja programa, a ako vrijeme nije dostupno, funkcija kao rezultat vraæa `-1`. Rezultat je cjelobrojnog tipa `clock_t` koji je pomoæu `typedef` deklariran u zaglavlju `ctime`. Želimo li to vrijeme pretvoriti u sekunde, valja ga podijeliti sa `CLOCKS_PER_SEC` (standardno makro ime definirano u zaglavlju `ctime`) – ono definira broj otkucaja sistemskog sata u sekundi za raèunalo na kojem radimo. Na primjer:

```
clock_t zacetak, konac;
zacetak = clock();
for (int i = 0; i < 10; i++) cout << i * 10 << endl;
konac = clock();
cout << "Svaki prolaz petlje trajao je u prosjeku "
     << (konac - zacetak) / CLK_TCK / 10. << " s" << endl;
```

#include <ctime> // stari naziv: <time.h>
double difftime(time_t trenutak2, time_t trenutak1);

Funkcija vraæa duljinu vremenskog intervala `trenutak2 - trenutak1`, izraženu u sekundama. `time_t` je sinonim za cjelobrojni tip podatka kojeg kao rezultat vraæa funkcija `time()`.
Argument je kalendarsko vrijeme (dobiveno na primjer pozivom funkcije `time()`), a povratna vrijednost je struktura `tm` (èiji je sadržaj opisan kod funkcija `gmtime()` i `localtime()`). Ta struktura je statièki alocirana, te se prepisuje pri ponovnom pozivu funkcije.

#include <ctime> // stari naziv: <time.h>
tm *gmtime(const time_t *pokVrijeme);
tm *localtime(const time_t *pokVrijeme);

Funkcija `gmtime()` pretvara kalendarsko vrijeme u GMT (*Greenwich Mean Time*), dok funkcija `localtime()`pretvara kalendarsko vrijeme u lokalno vrijeme. Argument obje

funkcije je kalendarsko vrijeme (dobiveno najèešæe pozivom funkcije `time()`), a rezultat funkcija je struktura `tm` koja sadrži podatke o vremenu:

```
struct tm {
    int tm_sec;         // sekundi nakon pune minute (0...59)
    int tm_min;         // minuta nakon punog sata (0...59)
    int tm_hour;    // sati nakon ponoći (0...23)
    int tm_mday;    // dan u mjesecu (1...31)
    int tm_mon;     // mjesec (0...11)
    int tm_year;        // godina počevši od 1900.
    int tm_wday;    // dana od nedjelje (0...6)
    int tm_yday;    // dana od početka godine (0...365)
    int tm_isdst;       // zastavica za ljetni pomak vremena
};
```

Ako je `tm_isdst` pozitivan, tada je aktiviran ljetni pomak vremena, ako je nula tada je ljetni pomak vremena neaktivan, a ako je negativan tada informacija nije dostupna. Valja paziti da je struktura koju stvaraju `gmtime()`, odnosno `localtime()` statièki alocirana, te se prepisuje pri svakom novom pozivu funkcije. Stoga, ako želimo saèuvati oèitano vrijeme, moramo strukturu preslikati u neki drugi objekt.

Sljedeće naredbe će ispisati današnji datum i točno vrijeme:

```
time_t ura;
tm *cajt;
ura = time(NULL);
cajt = localtime(&ura);
cout << "Danas je "
     << cajt->tm_mday << "."
     << (cajt->tm_mon + 1) << "."
     << (1900 + cajt->tm_year) << "." << endl;
cout << "Točno je "
     << cajt->tm_hour << ":"
     << cajt->tm_min << " sati" << endl;
```

#include <ctime> // stari naziv: <time.h>
time_t mktime(tm *pokVrijeme);

Pretvara vrijeme pohranjeno u strukturi `tm` u kalendarsko vrijeme (tj. vrijeme u formatu u kojem ga vraæa funkcija `time()` – u neku ruku je ta funkcija inverzna funkciji `localtime()`). Ako je pretvorba vremena uspješno provedena, funkcija vraæa strukturu `time_t` s upisanim podacima, a ako nije uspjela, rezultat funkcije je `-1`.

```
#include <ctime>                    // stari naziv: <time.h>
time_t time(time_t *vrijeme);
```

Funkcija `time()` vraæa kalendarsko vrijeme, a ako vrijeme nije dostupno, funkcija vraæa `-1`. Kao argument funkciji može se proslijediti nul-pokazivaè; ako se proslijedi pokazivaè na neki objekt tipa `time_t`, tada funkcija pridružuje rezultat tom objektu. Valja spomenuti da se kalendarsko vrijeme može razlikovati od lokalnog vremena, na primjer zbog razlike u vremenskim zonama – da bi se dobilo lokalno vrijeme treba pozvati funkciju `localtime()` koja æe kalendarsko vrijeme pretvoriti u lokalno.

B.4. Matematièke funkcije

Veæina matematièkih funkcija naslijeðena je iz biblioteka jezika C, s time da su dodatno preoptereæene za ostale tipove podataka. Deklaracije funkcija naslijeðenih iz jezika C su oznaèene tamno, dok su deklaracije novododanih preoptereæenih varijanti pisane svjetlijim slovima. Takoðer, matematièke funkcije za koje su moguæe operacije s kompleksnim brojevima, definirane su pomoæu predložaka za klasu `complex`.

```
#include <cstdlib>                  // stari naziv: <stdlib.h>
int abs(int x);
long abs(long x);
long labs(long x);
```

```
#include <cmath>                    // stari naziv: <math.h>
float abs(float x);
double abs(double x);
long double abs(long double x);
float fabs(float x);
```

Funkcije kao rezultat vraæaju apsolutnu vrijednost broja `x`. U zaglavlju `cstdlib` deklarirane su cjelobrojne verzije funkcije, a u `cmath` su definirane realne varijante (ovakva razdvojenost naslijeðena je iz standardnih C biblioteka). Buduæi da u jeziku C nema moguænosti preoptereæivanja imena funkcija, u standardnim C bibliotekama verzija za realne brojeve se zove `fabs()`, a verzija za `long` se zove `labs()`.

```
#include <cmath>                    // stari naziv: <math.h>
float acos(float x);
double acos(double x);
long double acos(long double x);
```

Funkcija raèuna arkus kosinus argumenta `x`. Argument mora biti unutar intervala od –1 do +1, a funkcija vraæa odgovarajuæu vrijednost kuta u radijanima, unutar intervala od 0 do π (3,1415926535...). Ako je argument izvan dozvoljenog intervala, funkcija postavlja globalnu varijablu `errno` na `EDOM` (*Domain Error*).

#include <cmath> // stari naziv: <math.h>
float asin(float x);
double asin(double x);
long double asin(long double x);

Funkcija raèuna arkus sinus argumenta `x`. Argument mora biti unutar intervala od –1 do +1, a funkcija vraæa odgovarajuæu vrijednost kuta u radijanima, unutar intervala od –π/2 do +π/2. Ako je argument izvan dozvoljenog intervala, funkcija postavlja globalnu varijablu `errno` na `EDOM` (*Domain Error*).

#include <cmath> // stari naziv: <math.h>
float atan(float x);
double atan(double x);
long double atan(long double x);

Funkcija raèuna arkus tangens argumenta `x`. Povratna vrijednost je kut izražen u radijanima, unutar intervala od –π/2 do +π/2.

#include <cmath> // stari naziv: <math.h>
float atan2(float y, float x);
double atan2(double y, double x);
long double atan2(long double y, long double x);

Funkcija raèuna arkus tangens omjera argumenata `y/x`. Za razliku od funkcije `atan()`, ova funkcija vraæa vrijednosti kuta u radijanima, za sva èetiri kvadranta, unutar intervala od –π do +π. Ako su oba argumenta jednaka nuli, tada funkcija postavlja globalnu varijablu `errno` na `EDOM` (*Domain Error*).

#include <cmath> // stari naziv: <math.h>
float ceil(float x);
double ceil(double x);
long double ceil(long double x);

float floor(float x);
double floor(double x);
long double floor(long double x);

Funkcija `ceil()` zaokružuje argument `x` na najbliži veæi cijeli broj, a `floor()` zaokružuje argument na najbliži manji cijeli broj. Funkcije vraæaju cjelobrojni dio zaokruženih brojeva. Na primjer, izvoðenje sljedeæih naredbi:

```
double broj = 123.54;
cout << ceil(broj) << endl;
cout << floor(broj) << endl;
```

ispisat æe brojeve `124` i `123`.

#include <cmath> // stari naziv: <math.h>
float cos(float x);
double cos(double x);
long double cos(long double x);

Funkcija raèuna kosinus argumenta `x` (zadanog u radijanima). Funkcija vraæa odgovarajuæu vrijednost unutar intervala od –1 do +1.

#include <cmath> // stari naziv: <math.h>
float cosh(float x);
double cosh(double x);
long double cosh(long double x);

Funkcija raèuna kosinus hiperbolni argumenta `x`. Funkcija vraæa izraèunatu vrijednost. Ako bi toèna vrijednost prouzroèila brojèani preljev, funkcija umjesto vrijednosti vraæa `HUGE_VAL`, a globalna varijabla `errno` se postavlja u `ERANGE` (*Range Error*).

#include <cstdlib> // stari naziv: <stdlib.h>
div_t div(int djeljenik, int djelitelj);
ldiv_t div(long djeljenik, long djelitelj);
ldiv_t ldiv(long djeljenik, long djelitelj);

Funkcije izraèunavaju kvocijent i ostatak dijeljenja dva cijela broja. Kao rezultat vraæaju standardno definirane tipove `div_t`, odnosno `ldiv_t` (definirane pomoæu `typedef` u `cstdlib`). U suštini su to strukture koje se sastoje od dva cijela broja tipa `int`:

```
typedef struct {
    int quot;       // kvocijent
    int rem;    // ostatak
} div_t;
```

odnosno `long`:

```
typedef struct {
    long quot;      // kvocijent
    long rem;       // ostatak
} ldiv_t;
```

Funkcija `div(long, long)` je preoptereæena varijanta funkcije `div(int, int)`. U jeziku C umjesto nje se koristi funkcija `ldiv()`. Ilustrirajmo primjenu funkcije `div()` sljedeæim primjerom:

```
div_t r = div(10, 3);
cout << "kvocijent: " << r.quot << endl
     << "ostatak:   " << r.rem << endl;
```

#include <cmath> // stari naziv: <math.h>
float exp(float x);
double exp(double x);
long double exp(long double x);

Raèuna potenciju e^x za zadani argument `x`, gdje je e = 2,718218... baza prirodnog logaritma. Funkcija vraæa izraèunatu vrijednost. Ako je vrijednost izvan opsega vrijednosti koji se zadanim tipom može prikazati, tada funkcija vraæa vrijednost `HUGE_VAL`, te postavlja globalnu varijablu `errno` na vrijednost `ERANGE` (*Range Error*).

#include <cmath> // stari naziv: <math.h>
float fmod(float djeljenik, float djelitelj);
double fmod(double djeljenik, double djelitelj);

Raèuna ostatak dijeljenja dva realna broja. Tako æe izvoðenje sljedeæih naredbi:

```
double x = 5.0;
double y = 2.2;
cout << fmod(x, y) << endl;
```

na zaslonu ispisati broj `0.6`.

#include <cmath> // stari naziv: <math.h>
float frexp(float x, int *eksponent);
double frexp(double x, int *eksponent);
long double frexp(long double x, int *eksponent);

Rastavlja broj `x` u mantisu i eksponent na bazu 2, tako da je $x = mantisa \cdot 2^{eksponent}$. Funkcija kao rezultat vraæa mantisu veæu ili jednaku `0.5`, a manju od `1`. Eksponent se pohranjuje na mjesto na koje pokazuje drugi argument funkcije. Primjerice, izvoðenje sljedeæih naredbi:

```
double broj = 4.;
int eksponent;
double mantisa = frexp(broj, &eksponent);
cout << broj << " = " << mantisa << " * 2 na " << eksponent
     << endl;
```

ispisat æe:

```
4 = 0.5 * 2 na 3
```

#include <cmath> // stari naziv: <math.h>
float ldexp(float x, int eksponent);
double ldexp(double x, int eksponent);
long double ldexp(long double x, int eksponent);

Funkcija raèuna vrijednost $x \cdot 2^{eksponent}$, pri èemu su x i *eksponent* argumenti koji se prenose funkciji. Inverzna ovoj funkciji je funkcija `frexp()`.

#include <cmath> // stari naziv: <math.h>
float log(float x);
double log(double x);
long double log(long double x);

float log10(float x);
double log10(double x);
long double log10(long double x);

Funkcija `log()` vraæa prirodni logaritam, dok funkcija `log10()` vraæa dekadski logaritam argumenta `x`. Ako je argument realan i manji od 0, tada funkcije postavljaju globalnu varijablu `errno` na vrijednost `EDOM` (*Domain Error*). Ako je argument jednak 0, funkcija vraæa vrijednost minus `HUGE_VAL` te globalnu varijablu `errno` postavlja na vrijednost `ERANGE` (*Range Error*).

#include <cmath> // stari naziv: <math.h>
float modf(float x, float *cijeliDio);
double modf(double x, double *cijeliDio);
long double modf(long double x, long double *cijeliDio);

Funkcija rastavlja broj `x` na njegov cijeli i decimalni dio. Povratna vrijednost funkcije je decimalni dio, dok se cijeli dio prenosi preko pokazivaèa koji se prosljeðuje funkciji kao drugi argument. Tako æe izvoðenje naredbi:

```
double broj = 94.3;
double cijeli;
double decimalni = modf(broj, &cijeli);
cout << broj << " = " << decimalni << " + " << cijeli << endl;
```

ispisati na zaslonu:

```
94.3 = 0.3 + 94
```

```
#include <cmath>                    // stari naziv: <math.h>
float pow(float baza, float eksponent);
float pow(float baza, int eksponent);
double pow(double baza, double eksponent);
double pow(double baza, int eksponent);
long double pow(long double baza, long double eksponent);
long double pow(long double baza, int eksponent);
```

Raèuna potenciju `baza`^{eksponent}. Ako je raèun uspješno proveden, funkcija kao rezultat vraæa izraèunatu potenciju. Ako je rezultat izvan opsega moguæih vrijednosti za rezultirajuæi tip, funkcija vraæa kao rezultat `HUGE_VAL` te postavlja globalnu varijablu `errno` na vrijednost `ERANGE` (*Range Error*). Ako se funkciji prenese realna baza manja od 0 ili ako su oba argumenta funkciji jednaki 0, funkcija postavlja vrijednost `errno` na `EDOM` (*Domain Error*). Za raèunanje potencija e^x praktiènije je koristiti funkciju `exp()`, a za raèunanje kvadratnog korijena na raspolaganju je funkcija `sqrt()`.

```
#include <cstdlib>                  // stari naziv: <stdlib.h>
int rand();
void srand(unsigned int klica);
```

Funkcija `rand()` kao rezultat vraæa pseudosluèajni broj u intervalu od 0 do `RAND_MAX`. Simbolièka konstanta `RAND_MAX` definirana je u zaglavlju `cstdlib`.

Ako se generator slučajnih brojeva ne inicijalizira nekim slučajnim brojem, svako izvođenje programa u kojem se poziva funkcija `rand()` rezultirat će uvijek istim slijedom slučajnih brojeva, koji su jednoliko raspoređeni u intervalu od 0 do `RAND_MAX`. Tako će program:

```
#include <stdlib.h>
#include <iostream.h>

int main() {
    for (int i = 0; i < 10; i++)
       cout << rand() << endl;
    return 0;
}
```

uvijek na nekom stroju ispisati potpuno isti niz brojeva (zato se govori o *pseudosluèajnim* brojevima). Da bi se prilikom svakog pokretanja programa dobili razlièiti nizovi sluèajnih brojeva, generator sluèajnih brojeva treba inicijalizirati nekim brojem koji æe se mijenjati od izvoðenja do izvoðenja programa. Najzgodnije je koristiti sistemsko vrijeme, koje se mijenja dovoljno brzo da korisnik ne može kontrolirati njegovu vrijednost.

Funkcija `srand()` služi za inicijalizaciju generatora slučajnih brojeva. Argument je klica pomoću koje se generira prvi broj u nizu slučajnih brojeva. Ako gornji primjer modificiramo, dodajući mu između ostalog poziv funkcije `srand()` ispred `for`-petlje:

```
#include <stdlib.h>
#include <iostream.h>
#include <time.h>

int main() {
    time_t cajt;
    srand((unsigned) time(&cajt));
    for (int i = 0; i < 10; i++)
       cout << (rand() % 6 + 1) << endl;
    return 0;
}
```

svako pokretanje programa rezultirat æe razlièitim nizom brojeva od 1 do 6.

#include <cmath> // stari naziv: <math.h>
float sin(float x);
double sin(double x);
long double sin(long double x);

Funkcija raèuna sinus argumenta `x` (zadanog u radijanima). Funkcija vraæa odgovarajuæu vrijednost unutar intervala od –1 do +1.

#include <cmath> // stari naziv: <math.h>
float sinh(float x);
double sinh(double x);
long double sinh(long double x);

Funkcija raèuna sinus hiperbolni argumenta `x`. Funkcija vraæa izraèunatu vrijednost. Ako bi toèna vrijednost prouzroèila brojèani preljev, funkcija vraæa kao rezultat `HUGE_VAL`, a globalna varijabla `errno` se postavlja u `ERANGE` (*Range Error*).

#include <cmath> // stari naziv: <math.h>
float sqrt(float x);
double sqrt(double x);

Raèuna kvadratni korijen argumenta `x`. Ako je argument pozitivan realni broj, funkcija vraæa pozitivni korijen. Ako je `x` realan i manji od nule, funkcija postavlja vrijednost globalne varijable `errno` na `EDOM` (*Domain Error*).

#include <cmath> // stari naziv: <math.h>
float tan(float x);
double tan(double x);
long double tan(long double x);

Funkcija raèuna i kao rezultat vraæa tangens kuta `x` (zadanog u radijanima).

#include <cmath> // stari naziv: <math.h>
float tanh(float x);
double tanh(double x);
long double tanh(long double x);

Funkcija raèuna i kao rezultat vraæa tangens hiperbolni argumenta `x`.

B.5. Ostale funkcije

#include <cstdlib> // stari naziv: <stdlib.h>
void abort();

Trenutaèno prekida izvoðenje programa, bez zatvaranja datoteka otvorenih tijekom izvoðenja programa. Procesu koji je pokrenuo program (najèešæe je to operacijski sustav) vraæa vrijednost 3. Ova funkcija poziva se prvenstveno u sluèajevima kada se želi sprijeèiti da program zatvori aktivne datoteke. Za uredan prekid programa se umjesto `abort()` koristi funkcija `exit()`.

#include <cstdlib> // stari naziv: <stdlib.h>
int atexit(void (*funkcija)(void));

Funkcija `atexit()` registrira funkciju na koju se pokazivaè prenosi kao argument. To znaèi da æe se ta funkcija pozvati prilikom urednog završetka programa, naredbom `return` iz `main()` ili pozivom funkcije `exit()`. Ako je više funkcija registrirano, one se pozivaju redoslijedom obrnutim od redoslijeda kojim su registrirane. Zbog toga æe izvoðenje sljedeæeg programa:

```
void izlaz1() {
    cerr << "Izlaz br. 1" << endl;
}

void izlaz2() {
    cerr << "Izlaz br. 2" << endl;
}


int main() {
    atexit(izlaz1);
    atexit(izlaz2);
    return 0;
}
```

na zaslonu ispisati:

```
Izlaz br. 2
Izlaz br. 1
```

#include <cstdlib> // stari naziv: <stdlib.h>
void *bsearch(const void *kljuc, const void *baza, size_t brElem,
size_t sirina, int (*usporedba)(const void *, const void *));

Funkcija obavlja binarno pretraživanje sortiranog polja na koje pokazuje `baza`. Kao rezultat vraæa pokazivaè na prvi èlan polja koji zadovoljava kljuè na koji pokazuje `kljuc`; ako nije uspjela pronaæi èlan koji zadovoljava kljuè, kao rezultat funkcija vraæa nul-pokazivaè. Broj elemenata u polju definiran je argumentom `brElem`, a velièina pojedinog èlana u polju argumentom `sirina`.

Funkcija na koju pokazuje `usporedba` koristi se za usporedbu članova polja sa zadanim ključem. Poredbena funkcija mora biti oblika:

```
int ime_funkcije(const void *arg1, const void *arg2);
```

Ona mora biti definirana tako da vraæa sljedeæe vrijednosti tipa `int`:

`< 0` ako je *`arg1`* manji od (tj. mora u slijedu biti prije) *`arg2`*,
`0` ako su *`arg1`* i *`arg2`* međusobno jednaki,
`> 0` ako je *`arg1`* veæi od (tj. mora u slijedu biti iza) *`arg2`*

Evo i primjera:

```
#include <stdlib.h>
#include <iostream.h>

// dodano da bi se prevario prevoditelj, jer funkcija
// bsearch() očekuje pokazivač na funkciju s argumentima
// tipa void *
typedef int (*fpok)(const void *, const void *);

int usporedba(const int *p1, const int *p2) {
   return(*p1 - *p2);
}

int main() {
    int podaci[] = {123, 234, 345, 456};
    int trazeni = 345;
    int brelem = sizeof(podaci) / sizeof(podaci[0]);
    int *pok = (int *)bsearch (&trazeni, podaci, brelem,
                    sizeof(podaci[0]), (fpok)usporedba);
    if (pok)
        cout << "U polju već postoji taj podatak na mjestu "
             << (pok - podaci) << endl;
    else
        cout << "Ovakvog podatka još nije vidjelo ovo polje!"
             << endl;
    return 0;
}
```

Uoèimo u gornjem primjeru definiciju `typedef` kojom je uveden sinonim za pokazivaè na funkciju usporedbe. Ovo je neophodno, jer funkcija `bsearch()` oèekuje pokazivaè na funkciju s oba argumenta tipa `void *`, a mi joj prosljeðujemo pokazivaè na funkciju koja ima kao argumente `int *`.

#include <cstdlib> // stari naziv: <stdlib.h>
void exit(int status);

Funkcija `exit()` uredno prekida izvoðenje programa. Prije prekida, svi se meðuspremnici prazne, datoteke se zatvaraju, te se pozivaju sve izlazne funkcije, koje su bile proslijeðene funkcijom `atexit()`.

Varijabla `status` koja se prosljeđuje funkciji služi da bi se pozivajući program (najčešće je to sam operacijski sustav), izvijestio o eventualnoj pogreški. Argument može biti `EXIT_FAILURE` ili `EXIT_SUCCESS` – makro imena definirana u zaglavlju `cstdlib`.

Funkcija `exit()` uglavnom se koristi kao "izlaz u slučaju nužde" – ako je nastupila neka fatalna pogreška u programu koja onemogućava daljnje izvođenje programa. U jeziku C++ njena upotreba je minimalizirana zahvaljujući iznimkama i njihovom hvatanju.

#include <clocale> // stari naziv: <locale.h>
lconv *localeconv();
char *setlocale(int kategorija, const char *mjesto);

Funkcija `localeconv()` vraæa trenutaènu postavu lokalnih zemljopisnih kategorija (formata datuma, vremena, realnih brojeva). Podaci su pohranjeni u strukturi tipa `lconv`, koja je deklarirana u zaglavlju `clocale`:

```
struct lconv {
  char *decimal_point;      // znak za razdvajanje cijelih i
                            // decimalnih mjesta u brojevima
  char *thousands_sep;      // znak za razdvajanje grupa
                            // znamenki po tisuću
  char *grouping;           // veličina svake grupe znamenki
  char *int_curr_symbol;    // međunarodna oznaka valute
  char *currency_symbol;    // lokalna oznaka valute
  char *mon_decimal_point;  // znak za razdvajanje cijelih i
                            // dec. mjesta u novčanim iznosima
  char *mon_thousands_sep;  // znak za razdvajanje grupa
                            // znamenki u novčanim iznosima
  char *mon_grouping;       // veličina svake grupe
  char *positive_sign;      // oznaka za pozitivne novč.iznose
  char *negative_sign;      // oznaka za negativne novč.iznose
  char int_frac_digits;     // broj decimalnih znamenki u me-
                            // đunarodnom prikazu novč.iznosa
  char frac_digits;         // broj decimalnih znamenki u
```

```
                                // lokalnom prikazu novč.iznosa
    char p_cs_precedes;         // =1 ako lokalna oznaka valute
                                // prethodi pozitivnom novč.iznosu
    char p_sep_by_space;        // =1 ako bjelina odvaja pozitivni
                                // novč.iznos od oznake valute
    char n_cs_precedes;         // =1 ako lokalna oznaka valute
                                // prethodi negativnom novč.iznosu
    char n_sep_by_space;        // =1 ako bjelina odvaja negativni
                                // novč.iznos od oznake valute
    char p_sign_posn;           // gdje smjestiti pozitivni
                                // predznak u novčanim iznosima
    char n_sign_posn;           // gdje smjestiti negativni
                                // predznak u novčanim iznosima
  };
```

Sadržaj te strukture može se promijeniti samo pomoæu funkcije `setlocale()`. Prvi argument pri pozivu funkcije jest kategorija na koju se promjena odnosi:

`LC_ALL`	obuhvaæa sve kategorije.
`LC_COLLATE`	utjeèe na funkcije `strcoll()` i `strxfrm()`.
`LC_CTYPE`	utjeèe na funkcije za rukovanje jednim znakom.
`LC_MONETARY`	format ispisa novèanih iznosa; vraæa ga funkcija `setlocaleconv()`.
`LC_NUMERIC`	znak za razdvajanje cijelih i decimalnih mjesta (npr. decimalna toèka ili zarez).
`LC_TIME`	utjeèe na `strftime()` funkciju za ispis tekuæeg datuma u obliku znakovnog niza.

Drugi argument funkciji `setlocale()` jest pokazivaè na znakovni niz koji specificira lokalnu postavu. Koji su nizovi podržani, ovisi o implementaciji biblioteke.

#include <cstring> // stari naziv: <string.h>
void *memchr(void *niz, int znak, size_t duljina);
const void *memchr(const void *niz, int znak, size_t duljina);

Pretražuje niz bajtova duljine `duljina`, poèevši od lokacije na koju pokazuje `niz` i traži znak `znak`. Kao rezultat vraæa pokazivaè na prvu pojavu znaka, a ako ga ne pronaðe, povratna vrijednost je nul-pokazivaè.

#include <cstring> // stari naziv: <string.h>
void *memcpy(void *odrediste, const void *izvornik, size_t duljina);
void *memmove(void *odrediste, const void *izvornik, size_t duljina);

Obje funkcije preslikavaju blok duljine `duljina` bajtova s mjesta u memoriji na koje pokazuje pokazivaè `izvornik`, na mjesto na koje pokazuje `odrediste`. Meðutim, ako se izvorni i odredišni blokovi preklapaju, ponašanje funkcije `memcpy()`je nedefinirano, dok æe funkcija `memmove()` preslikati blok korektno. Primjerice, provjerite što æete dobiti ispisom polja `abcd[]` nakon sljedeæih naredbi:

```
char abcd[] = "abcdefghijlkmnopqrstuvwxyz";
memcpy(abcd + 5, abcd, 12);
```

#include <cstring> // stari naziv: <string.h>
int memcmp(const void *niz1, const void *niz2, size_t duljina);

Usporeðuje prvih `duljina` bajtova oba niza, a kao rezultat usporedbe vraæa:
< 0 ako je `niz1` manji od `niz2` (svrstano po abecedi `niz1` dolazi prije `niz2`),
0 ako su `niz1` i `niz2` meðusobno jednaki,
> 0 ako je `niz1` veæi od `niz2` (svrstano po abecedi `niz1` dolazi iza `niz2`).

Usporedba se radi po slijedu znakova koji se koristi na dotiènom raèunalu.

#include <cstring> // stari naziv: <string.h>
void *memset(void *niz, int znak, size_t duljina);

Blok duljine `duljina` bajtova, poèevši od lokacije na koju pokazuje `niz`, funkcija popunjava znakom `znak`. Kao rezultat, funkcija vraæa pokazivaè na poèetak niza.

#include <cstdlib> // stari naziv: <stdlib.h>
void *qsort(void *baza, size_t brElem, size_t sirina,
int (*usporedba)(const void *, const void *));

Sortira polje `baza[0]`...`baza[brElem - 1]` koje èine podaci širine `sirina`. Za usporedbu meðu èlanovima polja koristi se funkcija na koju pokazuje `usporedba`; funkcija mora imati identièna svojstva kao i funkcija koja se poziva u funkciji `bsearch()`:

```
int ime_funkcije(const void *arg1, const void *arg2);
```

Ona mora biti definirana tako da vraæa sljedeæe vrijednosti tipa `int`:
< 0 ako je *arg1* manji od (tj. mora u slijedu biti prije) *arg2*,
0 ako su *arg1* i *arg2* meðusobno jednaki,
> 0 ako je *arg1* veæi od (tj. mora u slijedu biti iza) *arg2*

Funkcija `qsort()` primjenjuje *quick sort* algoritam koji se smatra najbržim za sortiranje opæenitih podataka. Sljedeæi program æe sortirati slova u nizu `podaci` obrnutim abecednim slijedom:

```
int usporedba(const void *p1, const void *p2) {
    return(*(char*)p2 - *(char*)p1);
}

int main() {
    char podaci[] = "adiorwgoerg";
```

```
    int brelem = (sizeof(podaci) - 1) / sizeof(podaci[0]);
    qsort(podaci, brelem, sizeof(podaci[0]), usporedba);
    cout << podaci << endl;

   return 0;
}
```

Uoèimo kako je ovdje poziv funkcije `usporedba()` drugaèije riješen nego kod primjera s funkcijom `bsearch()`: unutar same funkcije `usporedba()` pokazivaèima na `void` se dodjeljuje tip `char *` te se provodi usporedba takvih tipova.

#include <csignal> // stari naziv: <signal.h>
int raise(int dojava);
void (*signal(int dojava, void (*rukovatelj)(int)))(int);

Funkcijom `signal()` definira se koja æe funkcija biti pozvana ako nastupi neka izvanredna situacija, na primjer ako se primi signal prekida (*interrupt*) sa neke vanjske jedinice ili pogreška tijekom izvoðenja programa. Prvi argument je tip dojave na koju se definicija nove funkcije `rukovatelj()` odnosi. Standardno su definirani (u zaglavlju `csignal`) sljedeæi tipovi:

`SIGABRT` nepravilan prekid, primjerice uslijed `abort()`;
`SIGFPE` aritmetièka pogreška, primjerice zbog dijeljenja s nulom ili preljev;
`SIGILL` neispravna instrukcija;
`SIGINT` interaktivno upozorenje, primjerice prekid;
`SIGSEGV` nepravilni pristup memoriji, na primjer pristup nedozvoljenom podruèju memorije;
`SIGTERM` program je zaprimio zahtjev za završetak.

Funkcija `rukovatelj()` je zadužena za obradu dojave. To može biti funkcija koju definira korisnik, ili dva predefinirana rukovatelja:

`SIG_DFL` zakljuèi izvoðenje programa;
`SIG_IGN` ignorira ovaj tip dojave.

Rukovateljska funkcija prihvaæa jedan cjelobrojni argument – tip dojave. Ako je poziv funkcije `signal()` bio uspješan, funkcija kao rezultat vraæa pokazivaè na prethodni rukovatelj navedenog tipa dojave; ako poziv nije bio uspješan, funkcija vraæa `SIG_ERR`. Valja voditi raèuna da se pozivom rukovatelja, briše dojava te ponovno treba instalirati funkciju za rukovanje.

Funkcija `raise()` šalje dojavu signala `dojava` programu. Ako je odašiljanje bilo uspješno, funkcija vraća vrijednost različitu od nule. U sljedećem primjeru ilustrirana je primjena obje funkcije:

```
void hvataljka(int dojava) {
    signal(SIGFPE, hvataljka);  // ponovno instalira hvataljku
    cout << "No, no: znaš da nije dozvoljeno dijeljenje s "
            "nulom" << endl;
```

```
    return;
}

int main() {
    int a = 10;
    int b = 0;
    if (signal(SIGFPE, hvataljka) == SIG_ERR)
        cout << "Hvatanje pogreške nije postavljeno - "
             << "nastavak programa na vlastitu odgovornost."
             << endl;
    if (b == 0)
        raise(SIGFPE);
    else
        a = a / b;
    return 0;
}
```

U jeziku C++ funkcije ove funkcije se rijetko koriste, jer se rukovanje izvanrednim situacijama elegantnije rješava hvatenjem iznimki. Takoðer, signali su ostaci UNIX biblioteke. U modernim, suvremenim i naprednim sustavima signali imaju bolju i kvalitetniju implementaciju.

#include <new> // stari naziv: <new.h>
new_handler set_new_handler(new_handler funkcija);

Funkcija `set_new_handler()` definira koja æe funkcija biti pozvana ako globalni operatori `new()` ili `new[]()` ne uspiju alocirati traženi memorijski prostor. Ako nije posebno zadano, operatori æe baciti iznimku tipa `bad_alloc` – pozivom funkcije `set_new_handler()` ovakvo ponašanje se može promijeniti. Funkcija koja obraðuje nedostatak memorijskog prostora kao parametar prima podatak tipa `size_t` koji pokazuje koliko se memorije tražilo, a kao rezultat vraæa `int`. Ako funkcija vrati nulu, time se signalizira operatoru `new` da dodatne memorije nema barem u zahtjevanoj kolièini. Vrijednost razlièita od nule signalizira da se uspjelo pronaæi dodatne memorije te da operator `new` može ponoviti alokaciju.

#include <exception>
terminate_handler set_terminate(terminate_handler funkcija);
void terminate();

Funkcija `terminate()` važna je za obradu iznimaka. Ona se automatski poziva u sljedeæim sluèajevima:

- ako neka baèena iznimka nije uhvaæena te je “izletila” iz funkcije `main()`,
- ako mehanizam za rukovanje iznimkama utvrdi da je stog poremeæen,
- kada destruktor pozvan tijekom odmatanja stoga izazvanog iznimkom pokuša izaæi pomoæu iznimke.

Funkcija `terminate()` æe prekinuti izvoðenje programa pozivom funkcije `abort()`. Ako ne želimo prekid programa na ovaj naèin, možemo sami zadati funkciji pomoæu funkcije `set_terminate()`. Prilikom poziva funkcije `set_terminate()`, ona kao rezultat vraæa pokazivaè na prethodno instaliranu funkciju.

Pokušaj da se funkcijom `set_terminate()` instalira funkcija koja neće prekinuti izvođenje programa, nego će se pokušati vratiti se pozivajućem kôdu, rezultirat će pogreškom prilikom izvođenja.

#include <cstdlib> // stari naziv: <stdlib.h>
int system(const char *naredba);

Funkcija šalje naredbu okružju iz kojeg je program pokrenut (najèešæe je to operacijski sustav). Primjerice, naredba:

```
system("blabla");
```

æe pokrenuti program `blabla`; nakon okonèanja tog programa, izvoðenje æe se nastaviti naredbom koja slijedi iza poziva funkcije `system()`.

#include <exception>
unexpected_handler set_unexpected(unexpected _handler funkcija);
void unexpected();

Funkcija `unexpected()` važna je za obradu iznimaka. Ona se automatski poziva ako iznimka "izleti" iz neke funkcije, a da pri tome nije navedena u listi moguæih iznimaka. Ako nije drugaèije specificirano, funkcija `unexpected()` æe pozvati funkciju `terminate()`, a ova æe prekinuti izvoðenje programa pozivom funkcije `abort()`. Ako ne želimo prekid programa na ovaj naèin, možemo sami zadati funkciji pomoæu funkcije `set_unexpected()`. Prilikom poziva funkcije `set_unexpected()`, ona kao rezultat vraæa pokazivaè na prethodno instaliranu funkciju.

C. Rjeènik èešæe korištenih pojmova

Hrvatsko-engleski rjeènik

anonimna unija	*anonymous union*
apstrakcija podataka	*data abstraction*
apstraktna klasa	*abstract class*
argument	*argument*
asembler	*assembler*
automatska smještajna klasa	*automatic storage class*
automatski objekt	*automatic object*
bacanje iznimke	*throwing an exception*
bajt	*byte*
bezimena unija	*nameless union*
bezimeni imenik	*nameless namespace*
biblioteka	*library*
blok pokušaja	*try block*
cijeli broj	*integer*
cjelobrojna promocija	*integral promotion*
èisti virtualni funkcijski èlan	*pure virtual function member*
datoteèni imenik	*directory*
datoteèno podruèje	*file scope*
datoteka	*file*
datoteka zaglavlja	*header file*
definicija funkcije	*function definition*
deklaracija unaprijed	*forward declaration*
destruktor	*destructor*
dinamièki objekt	*dynamic object*
dinamièki poziv	*dynamic call*
dinamièko povezivanje	*dynamic binding*
diskriminanta unije	*union discriminant*
djelomièna specijalizacija	*partial specialization*
dodjela tipa	*type cast*
dominacija	*dominance*
duboka kopija	*deep copy*
enkapsulacija	*encapsulation*
formalni argument	*formal argument*
funkcijski èlan	*member functions*
globalno podruèje	*global scope*
hrpa	*heap*
hvatati	*catch*
identifikacija tipova tijekom izvoðenja	*run-time type identification*

imenik	*namespace*
implementacija objekta	*implementation*
instanciranje predloška	*template instantiation*
integrirana razvojne okoline	*integrated development environment, IDE*
iskljuèivi ili	*exclusive or*
isprazniti	*flush*
izbornik	*menu*
izlazni tok	*output stream*
iznimka	*exception*
izvedbeni program	*executable*
izvedena klasa	*derived class*
izvorni kôd	*source code*
javni	*public*
javna osnovna klasa	*public base class*
javno suèelje	*public interface*
kasno povezivanje	*late binding*
klasa	*class*
konstantnost	*constness*
konstruktor	*constructor*
konstruktor kopije	*copy constructor*
kontejnerska klasa	*container class*
kurzor	*cursor*
lvrijednost	*lvalue*
makro funkcija	*macro function*
makro ime	*macro name*
manipulator	*manipulator*
međupohranjivanje	*buffering*
međuspremnik	*buffer*
memorijska napuklina	*memory leak*
metoda	*methods*
mjesto instantacije	*point of instantiation*
najdalje izvedena klasa	*most derived class*
nasljeđivanje	*inheritance*
neimenovani privremeni objekt	*unnamed temporary*
nevezano prijateljstvo	*unbound template friendship*
nul-pokazivaè	*null-pointer*
nul-znak	*null-character*
objektni kôd	*object code*
objektno orijentirano programiranje	*object oriented programming*
obnavljajuæe pridruživanje	*update assignment*
odbaciti konstantnost	*cast away constness*
odmatanje stoga	*stack unwinding*
omotaè	*wrapper*
operator izluèivanja	*extraction operator*

operator umetanja	*insertion operator*
operator za indeksiranje	*indexing operator*
operator za odreðivanje podruèja	*scope resolution operator*
operator za pristup èlanu	*member selection operator*
oporavak od iznimke	*exception recovery*
osnovna klasa	*base class*
pametni pokazivaè	*smart pointer*
parametar	*parameter*
plitka kopija	*shallow copy*
pobrojenje	*enumeration*
pobrojani tip	*enumerated type*
podatkovni segment	*data segment*
podizanje iznimke	*raising an exception*
podrazumijevana vrijednost argumenta	*default argument value*
podrazumijevani konstruktor	*default constructor*
podruèje	*scope*
pogonitelj	*driver*
pogonjeno dogaðajima	*event-driven*
pogreška	*bug*
pogreška pri izvoðenju	*run-time error*
pogreška pri povezivanju	*link-time error*
pogreška pri prevoðenju	*compile-time error*
pogrešno	*false*
pokazivaè	*pointer*
pokazivaè datoteke	*file pointer*
pokazivaè instrukcija	*instruction pointer*
pokazivaè na èlan klase	*class member pointer*
pokazivaè stoga	*stack pointer*
polimorfizam	*polimorphysm*
polje bitova	*bit-fields*
polje podataka	*array*
pomak udesno	*shift right*
pomak ulijevo	*shift left*
ponovnu iskoristivost kôda	*code reusability*
popratna pojava	*side-effect*
posebna sekvenca	*escape sequence*
potpis funkcije	*function signature*
povezivaè	*linker*
povratna vrijednost	*return value*
pravila provjere tipa	*type-checking rules*
pravilo “od palca”	*rule of the thumb*
prazan	*void*
predložak	*template*
predložak funkcije	*function template*

predložak klase	*class template*
prekid	*interrupt*
prekomjerno bujanje kôda	*code bloat*
preoptereæenje funkcije	*function overloading*
preoptereæenje operatora	*operator overloading*
pretvorba naniže	*downcast*
pretvorba naviše	*upcast*
prevoditelj	*compiler*
prijatelj klase	*friend of a class*
prijenos po referenci	*pass by reference*
prijenos po vrijednosti	*pass by value*
privatna osnovna klasa	*private base class*
privatni	*private*
program za lociranje pogrešaka	*debugger*
program za praæenje efikasnosti kôda	*profiler*
program za ureðivanje teksta	*text editor*
programersko suèelje	*application programming interface*
promjenjiv	*volatile*
proširenje imenika	*namespace extension*
prototip funkcije	*function prototype*
razluèivanje podruèja	*scope resolution*
realni broj	*floating-point number*
registar	*register*
registarska smještajna klasa	*register storage class*
rekurzija	*recursion*
relacijski operator	*relational operator*
rukovanje iznimkama	*exception handling*
skrivanje podataka	*data hiding*
skupljanje smeæa	*garbage collection*
smještajna klasa	*storage class*
specijalizacija predloška	*template specialization*
središnja procesorska jedinica	*central processing unit, CPU*
standardna biblioteka predložaka	*standard template library*
stanje	*state*
statièki objekt	*static object*
statièki poziv	*static call*
statièko povezivanje	*static binding*
stog	*stack*
struktura	*structure*
stvarni argument	*actual argument*
tijelo funkcije	*function body*
tip	*type*
tip povezivanja	*linkage*
toèno	*true*
tok	*stream*

ugniježðena klasa	*nested class*
ulazni tok	*input stream*
umetnuta definicija	*inline definition*
umetnuta funkcija	*inline function*
unija	*union*
unutarnje povezivanje	*internal linkage*
vanjsko povezivanje	*external linkage*
vezana lista	*linked list*
vezano prijateljstvo	*bound template friendship*
virtualna osnovna klasa	*virtual base class*
virtualni funkcijski èlan	*virtual function member*
virtualni poziv	*virtual call*
viseæa referenca	*dangling reference*
viseæi pokazivaè	*dangling pointer*
višestruko nasljeðivanje	*multiple inheritance*
zagaðenje globalnog podruèja	*global scope pollution*
zaobilaženje prilikom nasljeðivanja	*overriding*
zastavica	*flag*
zaštiæen	*protected*
zaštiæena osnovna klasa	*protected base class*
znakovni niz	*character string*

Englesko-hrvatski rjeènik

abstract class	*apstraktna klasa*
actual argument	*stvarni argument*
anonymous union	*anonimna unija*
application programming interface	*programersko suèelje*
argument	*argument*
array	*polje podataka*
assembler	*asembler*
automatic object	*automatski objekt*
automatic storage class	*automatska smještajna klasa*
base class	*osnovna klasa*
bit-fields	*polje bitova*
bound template friendship	*vezano prijateljstvo*
buffer	*međuspremnik*
buffering	*međupohranjivanje*
bug	*pogreška*
byte	*bajt*
cast away constness	*odbaciti konstantnost*
catch	*hvatati*
central processing unit, CPU	*središnja procesorska jedinica*
character string	*znakovni niz*
class	*klasa*
class member pointer	*pokazivaè na èlan klase*
class template	*predložak klase*
code bloat	*prekomjerno bujanje kôda*
code reusability	*ponovnu iskoristivost kôda*
compile-time error	*pogreška pri prevođenju*
compiler	*prevoditelj*
constness	*konstantnost*
constructor	*konstruktor*
container class	*kontejnerska klasa*
copy constructor	*konstruktor kopije*
cursor	*kurzor*
dangling pointer	*viseæi pokazivaè*
dangling reference	*viseæa referenca*
data abstraction	*apstrakcija podataka*
data hiding	*skrivanje podataka*
data segment	*podatkovni segment*
debugger	*program za lociranje pogrešaka*
deep copy	*duboka kopija*
default argument value	*podrazumijevana vrijednost argumenta*
default constructor	*podrazumijevani konstruktor*

derived class	*izvedena klasa*
destructor	*destruktor*
directory	*datoteèni imenik*
dominance	*dominacija*
downcast	*pretvorba naniže*
driver	*pogonitelj*
dynamic binding	*dinamièko povezivanje*
dynamic call	*dinamièki poziv*
dynamic object	*dinamièki objekt*
encapsulation	*enkapsulacija*
enumerated type	*pobrojani tip*
enumeration	*pobrojenje*
escape sequence	*posebna sekvenca*
event-driven	*pogonjeno dogaðajima*
exception	*iznimka*
exception handling	*rukovanje iznimkama*
exception recovery	*oporavak od iznimke*
exclusive or	*iskljuèivi ili*
executable	*izvedbeni program*
external linkage	*vanjsko povezivanje*
extraction operator	*operator izluèivanja*
false	*pogrešno*
file	*datoteka*
file pointer	*pokazivaè datoteke*
file scope	*datoteèno podruèje*
flag	*zastavica*
floating-point number	*realni broj*
flush	*isprazniti*
formal argument	*formalni argument*
forward declaration	*deklaracija unaprijed*
friend of a class	*prijatelj klase*
function body	*tijelo funkcije*
function definition	*definicija funkcije*
function overloading	*preoptereæenje funkcije*
function prototype	*prototip funkcije*
function signature	*potpis funkcije*
function template	*predložak funkcije*
garbage collection	*skupljanje smeæa*
global scope	*globalno podruèje*
global scope pollution	*zagaðenje globalnog podruèja*
header file	*datoteka zaglavlja*
heap	*hrpa*
implementation	*implementacija objekta*
indexing operator	*operator za indeksiranje*
inheritance	*nasljeðivanje*

inline definition	*umetnuta definicija*
inline function	*umetnuta funkcija*
input stream	*ulazni tok*
insertion operator	*operator umetanja*
instruction pointer	*pokazivaè instrukcija*
integer	*cijeli broj*
integral promotion	*cjelobrojna promocija*
integrated development environment, IDE	*integrirana razvojne okoline*
internal linkage	*unutarnje povezivanje*
interrupt	*prekid*
late binding	*kasno povezivanje*
library	*biblioteka*
linkage	*tip povezivanja*
linked list	*vezana lista*
linker	*povezivaè*
link-time error	*pogreška pri povezivanju*
lvalue	*lvrijednost*
macro function	*makro funkcija*
macro name	*makro ime*
manipulator	*manipulator*
member functions	*funkcijski èlan*
member selection operator	*operator za pristup èlanu*
memory leak	*memorijska napuklina*
menu	*izbornik*
methods	*metoda*
most derived class	*najdalje izvedena klasa*
multiple inheritance	*višestruko nasljeðivanje*
nameless namespace	*bezimeni imenik*
nameless union	*bezimena unija*
namespace	*imenik*
namespace extension	*proširenje imenika*
nested class	*ugniježðena klasa*
null-character	*nul-znak*
null-pointer	*nul-pokazivaè*
object code	*objektni kôd*
object oriented programming	*objektno orijentirano programiranje*
operator overloading	*preoptereæenje operatora*
output stream	*izlazni tok*
overriding	*zaobilaženje prilikom nasljeðivanja*
parameter	*parametar*
partial specialization	*djelomièna specijalizacija*
pass by reference	*prijenos po referenci*
pass by value	*prijenos po vrijednosti*
point of instantiation	*mjesto instantacije*

pointer	*pokazivaè*
polymorphism	*polimorfizam*
private	*privatni*
private base class	*privatna osnovna klasa*
profiler	*program za praæenje efikasnosti kôda*
protected	*zaštiæen*
protected base class	*zaštiæena osnovna klasa*
public	*javni*
public base class	*javna osnovna klasa*
public interface	*javno suèelje*
pure virtual function member	*èisti virtualni funkcijski èlan*
raising an exception	*podizanje iznimke*
recursion	*rekurzija*
register	*registar*
register storage class	*registarska smještajna klasa*
relational operator	*relacijski operator*
return value	*povratna vrijednost*
rule of the thumb	*pravilo "od palca"*
run-time error	*pogreška pri izvoðenju*
run-time type identification	*identifikacija tipova tijekom izvoðenja*
scope	*podruèje*
scope resolution	*razluèivanje podruèja*
scope resolution operator	*operator za odreðivanje podruèja*
shallow copy	*plitka kopija*
shift left	*pomak ulijevo*
shift right	*pomak udesno*
side-effect	*popratna pojava*
smart pointer	*pametni pokazivaè*
source code	*izvorni kôd*
stack	*stog*
stack pointer	*pokazivaè stoga*
stack unwinding	*odmatanje stoga*
standard template library, STL	*standardna biblioteka predložaka*
state	*stanje*
static binding	*statièko povezivanje*
static call	*statièki poziv*
static object	*statièki objekt*
storage class	*smještajna klasa*
stream	*tok*
structure	*struktura*
template	*predložak*
template instantiation	*instanciranje predloška*
template specialization	*specijalizacija predloška*
text editor	*program za ureðivanje teksta*
throwing an exception	*bacanje iznimke*

true	*toèno*
try block	*blok pokušaja*
type	*tip*
type cast	*dodjela tipa*
type-checking rules	*pravila provjere tipa*
unbound template friendship	*nevezano prijateljstvo*
union	*unija*
union discriminant	*diskriminanta unije*
unnamed temporary	*neimenovani privremeni objekt*
upcast	*pretvorba naviše*
update assignment	*obnavljajuæe pridruživanje*
virtual base class	*virtualna osnovna klasa*
virtual call	*virtualni poziv*
virtual function member	*virtualni funkcijski èlan*
void	*prazan*
volatile	*promjenjiv*
wrapper	*omotaè*

Abecedno kazalo

#

!, 57
!=, 59
" (#include), 475
" (znakovni niz), 26, 60, 139
"C", 509
"C++", 511
#, 480
##, 480
%, 46
%:, 70
%:%:, 70
%=, 69
%>, 70
&, 63, 114
&&, 57
&=, 69
', 60
(), 76, 152, 154, 310
  preopterećenje, 317
* (množenje), 46
* (pokazivač), 114
*=, 69
+, 46
++, 45, 125
  preopterećenje, 321
+=, 69
,, 74, 88
-, 46
--, 45, 125
  preopterećenje, 321
-=, 69
->, 221, 310
  preopterećenje, 318
->*, 274, 278
., 221
.*, 274, 278
... (funkcije)
  argumenti funkcije, 176
... (iznimke)
  hvatanje iznimke, 454
/, 46
/* */ (komentar), 29
// (komentar), 29
/=, 69
:, 95, 297
::, 186, 223, 259, 262, 270, 342, 435, 436, 437
  i virtualni član, 379
:>, 70
;, 25, 219
<, 59
<%, 70
<:, 70
<< (izlazni operator), 26, 517
  preopterećenje, 519
  razlikovanje << i >>, 28
<< (pomak ulijevo), 65
<<=, 69
<=, 59
<> (#include), 475
<> (predložak), 211, 392, 398, 409, 412
=, 39, 69, 310
  i konstruktor konverzije, 370
  nasljeđivanje, 369
  preopterećenje, 313
=0, 380
==, 59
>, 59
>=, 59
>> (pomak udesno), 65
>> (ulazni operator), 27, 521
  preopterećenje, 523
  razlikovanje << i >>, 28
>>=, 69
?:, 83
??!, 71
??', 71
??(, 71
??), 71
??-, 71
??/, 71

`??<`, 71
`??=`, 71
`??>`, 71
`[]`, 104, 310
  preopterećenje, 315
\ (nastavak reda), 31, 474
\ (posebna, *escape* sekvenca), 60
`\"`, 60
`\'`, 60
`\0`, 60, 140
`\?`, 60
`\\`, 60
`^`, 64
`^=`, 69
`{}`, 77, 153
`|`, 64
`|=`, 69
`||`, 57
~, 62, 246

A

`\a`, 60
`abort()`, 574, 598
`abs()`, 591
abstract class. *Vidi* apstraktna klasa
`acos()`, 591
`adjustfield`, 533
`algorithm`, 573
`and`, 70
`and_eq`, 70
anonymous unions. *Vidi* unija, anonimna
ANSI C++, 18
API, 558
`app`, 545
application programming interface. *Vidi* programersko sučelje (API)
apstrakcija
  definicija, 560
  definicija javnog sučelja, 571
  implementacija, 578
  implementacijski zavisna, 569
  ocjenjivanje, 559
  odnosi i veze, 563
  pronalaženje, 559
apstrakcija podataka, 555
apstraktna klasa, 330, 380
`argc`, 202
argument
  formalni, 154
  klasa kao, 363
  konstantni, 172
  konstruktora, 238, 240
  konverzija, 163
  neodređeni, 176
  podrazumijevani, 173, 192, 201
  pokazivač kao, 164
  polje kao, 168
  predloška funkcije, 211, 392
  predloška funkcije, konstantni izraz, 396
  predloška klase, 409
  predloška klase, konstantni izraz, 420
  preopterećene funkcije, 191
  prijenos po referenci, 165
  prijenos po vrijednosti, 162, 181
  privremeni objekt za prijenos po vrijednosti, 284
  redoslijed izračunavanja, 163
  referenca kao, 165
  stvarni, 154
  točno podudaranje, 364
  znakovni niz kao, 169
`argv`, 202
array. *Vidi* polje
ASCII, 61
`asctime()`, 588
asembler, 10, 511
  uključivanje u C++ kôd, 511
`asin()`, 592
`asm`, 511
`assert()`, 236, 248, 483, 574
`assert.h`, 236, 483
`atan()`, 205, 592
`atan2()`, 204, 592
`ate`, 545
`atexit()`, 598
`atof()`, 203, 586
`atoi()`, 587
`atol()`, 587
`auto`, 182
automatic storage class. *Vidi* smještajna klasa, automatska
automatski objekti, 128

B

`\b`, 60

`bad()`, 516
`bad_cast`, 469
`badbit`, 516
base class. *Vidi* osnovna klasa
`basefield`, 533
`basic_string`, 572, 578
`before()`, 466
`beg`, 549
bezimeni imenik, 437
biblioteka, 22
  funkcija, 204, 580. *Vidi* standardna funkcija
  standardna, 571
`binary`, 545
`bitand`, 70
bit-fields. *Vidi* polje bitova
`bitor`, 70
bitovni operator
  i, 62, 63
  ili, 62, 64
  isključivi *ili*, 62, 64
  komplement, 62
  pomak udesno, 65
  pomak ulijevo, 65
`bits`, 573
blok
  hvatanja, 448, 452, 453
  hvatanja, određivanje odgovarajućeg bloka, 453
  pokušaja, 448, 452
blok naredbi, 38, 77
`bool`, 57
`boolalpha`, 533
bound template friendship. *Vidi* vezano prijateljstvo
`break`, 84, 93
broj
  cijeli, 40
  dekadski, 41
  heksadekadski, 41
  oktalni, 41
  preljev, 46
  realni, 40
  s pomičnom decimalnom točkom. *Vidi* broj, realni
brojčana konstanta, 35, 51
buffer. *Vidi* međuspremnik
buffering. *Vidi* međupohranjivanje

C

`.c`, 22, 495
`case`, 84
`cassert`, 572
cast. *Vidi* dodjela tipa
`catch`, 448, 450
catching an exception. *Vidi* iznimka, hvatanje
`cctype`, 572
`ceil()`, 592
`cerr`, 516
`cerrno`, 572
`char`, 44, 60
cijeli broj, 40
  dijeljenje, 48
`cin`, 27, 515, 521
`class`, 211, 218, 392, 409. *Vidi* klasa
class member pointer. *Vidi* pokazivač, na član klase
class template. *Vidi* predložak klase
`clear()`, 516, 548
`clocale`, 573
`clock()`, 589
`clock_t`, 588, 589
`CLOCKS_PER_SEC`, 589
`clog`, 516
`close()`, 545
`cmath`, 573
code reusability. *Vidi* ponovna iskoristivost
compiler. *Vidi* prevoditelj
`compl`, 70
`complex`, 573, 578
`complex.h`, 205
`const`, 53, 136, 172, 413. *Vidi* simbolička konstanta
  argument funkcije, 172
  `const_cast`, 471
  funkcijski član, 253
  odbacivanje konstantnosti, 139
  podatkovni član, 244
  pokazivač, 137
  pokazivač na, 137
  pokazivač na `const`, 138
  razlika `const` i `#define`, 477
  reference na, 145
`const_cast`, 471
constructor. *Vidi* konstruktor
container class. *Vidi* kontejnerska klasa

`continue`, 94
copy constructor. *Vidi* konstruktor, konstruktor kopije
`cos()`, 593
`cosh()`, 593
`cout`, 26, 516, 517
`.cp`, 22, 495
`__cplusplus`, 478, 510
`.cpp`, 22, 495
CRC kartica, 563
`csetjmp`, 572
`csignal`, 572
`cstdarg`, 572
`cstdio`, 573
`cstdlib`, 572, 573
`cstring`, 572
`ctime`, 572, 588
`ctime()`, 588
`ctype.h`, 528
`cur`, 549
`cwchar`, 572, 573
`cwctype`, 572

Č

član
- funkcijski. *Vidi* funkcijski član
- imenika, korištenje, 437
- isključivanje iz nasljeđivanja, 353
- javni, 226
- podatkovni. *Vidi* podatkovni član
- podrazumijevano pravo pristupa, 227
- pokazivač na, 271
- polja bitova, 297
- pravo pristupa, 226
- pristup, 220
- privatni, 226
- puno ime, 223, 261
- razlika statičkog povezivanja i statičkog člana, 491
- virtualne osnovne klase, 385
- zaštićeni, 226

D

dangling pointer. *Vidi* pokazivač, viseći
dangling reference. *Vidi* referenca, viseća
data abstraction. *Vidi* apstrakcija podataka
data hiding. *Vidi* skrivanje podataka
data segment. *Vidi* podatkovni segment
`__DATE__`, 478
datotečno područje, 262
datoteka
- binarna, 550
- čitanje iz, 541
- mod otvaranja, 545
- otvaranje, 544
- pisanje u, 541
- pokazivač, 548
- zatvaranje, 545

datoteka izvornog kôda, 486
datoteka zaglavlja, 155, 494. *Vidi* zaglavlje
- i statički objekt, 258
- što u nju ne staviti, 497
- što u nju staviti, 495

debugger, 23, 53
- simbolički, 23

`dec`, 533, 538
deep copy. *Vidi* duboka kopija
`default`, 84
default argument. *Vidi* argument, podrazumijevani
default constructor. *Vidi* konstruktor, podrazumijevani
`#define`, 53, 476, 479
- razlika `#define` i `const`, 477
- trajanje definicije, 477

definicija
- funkcije, 153
- funkcijskog člana klase, 223
- lokalne klase, 270
- podatkovnog člana, 220
- predloška funkcije, 392, 393
- predloška klase, 409
- preopterećenog operatora, 309
- specijalizacija predloška klase, 417
- specijalizacije predloška funkcije, 405

definiranje `goto` oznake, 95
deklaracija, 38
- apstraktne klase, 380
- `const` pokazivača, 137
- `const` pokazivača na `const`, 138
- čistog virtualnog člana, 380
- čitanje deklaracije pokazivača, 138
- `extern`, 158
- funkcija s podrazumijevanim argumentom, 175
- funkcije, 152, 494

funkcije s neodređenim argumentima, 176
globalnog objekta, 184, 248
imenika, 435
izvedene klase, 335
klase, 218, 232
klase unaprijed, 220
mjesto navođenja, 39
objekta, 232
pokazivača na `const`, 137
pokazivača na funkcijski član, 278
pokazivača na podatkovni član, 272
polja, 102
polja bitova, 297
polja znakovnih nizova, 143
predloška funkcije, 393
razlika deklariranja struktura u C i C++ jezicima, 293
razlika deklariranja unija u C i C++ jezicima, 294
reference, 144
reference na `const`, 145
statičkog funkcijskog člana, 260
statičkog objekta, 248
statičkog podatkovnog člana, 258
unutar bloka, 77
`using`, 353, 439
`using`, unutar klase, 441
`virtual`, 375
virtualne osnovne klase, 384
znakovnog niza, 139
`delete`, 129, 247, 381
i polje objekata, 250
nasljeđivanje, 371
preopterećenje, 323
`delete []`, 129, 250
i polje objekata, 250
preopterećenje, 324
`deque`, 573, 577
derived class. *Vidi* izvedena klasa
destructor. *Vidi* destruktor
destruktor, 216, 246
apstraktne klase, 382
dealokacija memorije objekta, 246
deinicijalizacija, 356
deinicijalizacija objekta, 246
eksplicitan poziv, 252
i virtualni poziv, 379
izvedene klase, 356
poziv, 246
poziv za globalne i statičke objekte, 249
redoslijed pozivanja, 356, 389
virtualni, 380
`difftime()`, 589
dijeljenje cijelih brojeva, 48
dinamička alokacija, 128
i iznimke, 455
operator `delete`, 129
operator `delete []`, 129
operator `new`, 128
operator `new []`, 129
polja, 129
polja objekata, 250
dinamički objekt, 128
alokacija, 238
inicijalizacija, 238
dinamički poziv, 375
dinamičko povezivanje, 373
direktiva
pretprocesorska. *Vidi* pretprocesor, naredba
`using`, 443
`using`, unutar imenika, 444
`div()`, 593
`div_t`, 593
dodjela tipa, 50, 400. *Vidi* konverzija
dominacija, 387
dominance. *Vidi* dominacija
`double`, 43, 44
`do-while`, 92
downcast. *Vidi* pretvorba naniže
driver. *Vidi* pogonitelj
duboka kopija, 242
dynamic binding. *Vidi* povezivanje, dinamičko
dynamic call. *Vidi* poziv, dinamički
`dynamic_cast`, 468
`bad_cast`, 469

E

`EDOM`, 574
`#elif`, 482
`#else`, 482
`else`, 80
encapsulation. *Vidi* enkapsulacija
`end`, 549
`#endif`, 248, 482
`endl`, 521, 538

`ends`, 521, 538
enkapsulacija, 12, 13, 215, 513
`enum`, 55
`EOF`, 526, 575
`eof()`, 516
`eofbit`, 516, 548
`ERANGE`, 574
`errno`, 574, 575
`#error`, 484
escape sequence. *Vidi* posebna sekvenca
exception, 572. *Vidi* iznimka
exception handling. *Vidi* rukovanje iznimkama
exception recovery. *Vidi* oporavak od iznimke
executable. *Vidi* izvedbeni kôd
`exit()`, 210, 249, 544, 600
`EXIT_FAILURE`, 575, 600
`EXIT_SUCCESS`, 575, 600
`exp()`, 193, 594
`explicit`, 302
`extern`, 158, 185, 394, 413, 492
`extern "C"`, 509
`extern "C++"`, 511
external linkage. *Vidi* povezivanje, vanjsko

F

`\f`, 60
`fabs()`, 199, 591
`fail()`, 516
`failbit`, 516, 522
`false`, 57
`__FILE__`, 266, 326, 478, 485
`FILE`, 126
file pointer. *Vidi* pokazivač datoteke
file scope. *Vidi* datotečno područje
`fill()`, 531
`fixed`, 533, 535
flag. *Vidi* zastavica
`flags()`, 532
`float`, 44
`float.h`, 205
`floatfield`, 533
floating-point. *Vidi* realni broj
`floor()`, 592
flush. *Vidi* međuspremnik, pražnjenje
`flush`, 520, 538
`flush()`, 520
`fmod()`, 594
`fopen()`, 126
`for`, 86
forward declaration. *Vidi* klasa, deklaracija unaprijed
`free()`, 325
`frexp()`, 594
`friend`, 230
- razlika prijateljstva i nasljeđivanja, 350
- unutar deklaracije klase, 231

friend of a class. *Vidi* prijatelj klase
`fstream`, 515, 541, 546, 573
`fstream.h`, 91, 205, 541
function overloading. *Vidi* funkcija, preopterećenje
function template. *Vidi* predložak funkcije
function template instantiation. *Vidi* predložak funkcije, instantacija
function template specialization. *Vidi* predložak funkcije, specijalizacija
funkcija, 150
- `argc`, 202
- argument, 150, 161
- `argv`, 202
- bez argumenata, 161
- datoteka zaglavlja, 155
- definicija, 153
- deklaracija, 152
- formalni argument, 154
- `inline`, 489
- konstantni argument, 172
- konverzija argumenata, 163
- konverzija rezultata, 158
- lista mogućih iznimaka, 458
- `main()`, 24, 202, 249, 453, 530
- mijenjanje pokazivačkih argumenata, 166
- operatorska, 309
- parametar. *Vidi* argument. *Vidi* funkcija, argument
- podrazumijevani argument, 173
- pokazivač kao argument, 164
- pokazivač na, 196
- polje kao argument, 168
- potpis, 153
- povratna vrijednost, 150, 159, 179
- poziv, 154
- poziv operatorske funkcije, 310
- pozivanje C funkcije, 509

predložak. *Vidi* predložak funkcije
preopterećenje, 189
preopterećenje predloška, 403
prijenos argumenta po referenci, 165
prijenos argumenta po vrijednosti, 162
prototip, 152, 494
provjera argumenta, 154
razlučivanje preopterećenih funkcija, 191
redoslijed deklaracije i poziva funkcije, 155
referenca kao argument, 165
rekurzivna, 195
rezultat, 288
s neodređenim argumentom, 176
s podrazumijevanim argumentom, 192, 201
standardna. *Vidi* standardna funkcija
stvarni argument, 154
tijelo, 153
tip, 158
umetnuta, 188, 489
`void`, 159
vraćanje vrijednosti, 153, 158
znakovni niz kao argument, 169
funkcijski član, 221
`const`, 253
čisti virtualni, 380
`inline`, 226
isključivanje iz nasljeđivanja, 353
javni, 226
konstantni, 253
podrazumijevano pravo pristupa, 227
pokazivač na, 277
poziv, 223, 372
pravo pristupa, 226
pristup, 223
privatni, 226
puno ime, 223, 261
razlika statičkog člana i statičkog poziva, 376
razlika statičkog povezivanja i statičkog člana, 491
statički, 260
statički, poziv, 261
umetnuti, 225
virtualne osnovne klase, 385
virtualni, 374
`volatile`, 257
vraćanje reference na objekt, 224
zaštićeni, 226

G

garbage collection. *Vidi* skupljanje smeća
`gcount()`, 526, 552
`get()`, 524
`getline()`, 526
global scope. *Vidi* globalno područje
global scope pollution. *Vidi* zagađenje globalnog područja
globano područje, 262
`gmtime()`, 589
`good()`, 516
`goodbit`, 516
`goto`, 94
definiranje oznake, 95
grananje toka, 77, 79, 84

H

`.h`, 495
header file. *Vidi* zaglavlje
heap. *Vidi* hrpa
`hex`, 63, 533, 538
hijerarhija operatora, 74
`.hpp`, 495
hrpa, 128, 213
`HUGE_VAL`, 575

I

IDE, 22
identifikacija tipa, 463, 465
`before()`, 466
`dynamic_cast`, 468
`name()`, 466
`type_info`, 465
`typeid`, 465
`typeinfo.h`, 465
usporedba, 465
identifikator, 35
dozvoljeno ime, 35
`#if`, 481
`if`, 79
`#ifdef`, 481
`#ifndef`, 248, 481
`ifstream`, 91, 515, 541
`ignore()`, 526
imenik, 434

alternativno ime, 436
bezimeni, 437
definicija člana, 435
deklaracija, 435
deklaracija člana, 435
deklaracija using, 439
direktiva using, 443
pristup članu, 437
pristup članu bezimenog imenika, 438
proširivanje, 436
proširivanje i deklaracija using, 440
proširivanje i direktiva using, 444
puni naziv člana, 435
puni naziv ugnježđenog člana, 436
ugnježđivanje, 436
umjesto static, 437
using direktiva unutar imenika, 444
using namespace, 443
implementacija klase, 229
implementacija objekta, 215, 578
implementation. *Vidi* objekt, implementacija
in, 545
#include, 156, 475, 494
", 475
<>, 475
inheritance. *Vidi* nasljeđivanje
inicijalizacija, 39
const objekta, 54
dinamički alociranog objekta, 128
izvedene klase, 355
konstantnog člana, 244
pokazivača, 116
polja, 103
polja objekata, 250
polja znakovnih nizova, 143
redoslijed inicijalizacije, 237, 356
reference, 145
reference kao člana, 243
specijalizirana, statičkog člana predloška, 420
statičkog člana u predlošku klase, 419
statičkog objekta, 186
statičkog podatkovnog člana, 258
u konstruktoru, 237
virtualne osnovne klase, 387
višedimenzionalnog polja, 109
znakovnog niza, 139
inline, 188, 226, 394, 489
inline function. *Vidi* funkcija, umetnuta
input stream. *Vidi* tok, ulazni
insertion operator. *Vidi* tok, operator umetanja
instruction pointer. *Vidi* pokazivač instrukcija
int, 41, 44
integer. *Vidi* cijeli broj
integral promotion. *Vidi* tip, cjelobrojna promocija
integrirana razvojna okolina, 22
internal, 533
internal linkage. *Vidi* povezivanje, unutarnje
iomanip, 573
iomanip.h, 205, 538
ios, 515, 573
iosfwd, 573
iostate, 516
iostream, 573
iostream.h, 26, 205, 513, 515, 517, 521
isalnum(), 529, 582
isalpha(), 529, 582
iscntrl(), 582
isdigit(), 528, 582
isgraph(), 582
islower(), 582
isprint(), 582
ispunct(), 582
isspace(), 528, 582
istream, 515, 546, 573
istringstream, 553
istrstream, 553
isupper(), 582
isxdigit(), 582
iterator, 573
izlazni tok, 26. *Vidi* tok, izlazni
iznimka, 446
bacanje, 449
bad_cast, 469
blok hvatanja, 448, 452, 453
blok pokušaja, 448, 452
dojava pogreške iz konstruktora, 460
hvatanje, 450
hvatanje objekta ili reference, 457
i dinamička alokacija, 455
konverzija prilikom hvatanja, 450
lista mogućih iznimaka, 458
neočekivana iznimka, 459
neuhvaćena, 453

odmatanje stoga, 452
određivanje bloka hvatanja, 453
podizanje, 448
prosljeđivanje, 454
`terminate()`, 453
tijek obrade, 451
tip, 449
tipa `...`, 454
`unexpected()`, 459
izvedbeni kôd, 20
izvedena klasa, 332, 335
deklaracija, 335
destruktor, 356
inicijalizacija, 355
konstruktor, 355
konverzija pokazivača na, 350
najdalje izvedena, 388
podrazumijevani konstruktor, 356
područje, 341, 360
pravo pristupa, 344
izvorni kôd, 21
u više datoteka, 486

J

javno sučelje, 215, 229
definicija, 571
javni pristup članu, 229
jezik
Actor, 555
Ada, 12
Algol68, 12
asembler, 11, 511
BASIC, 11
BCPL, 11
C, 11, 14
C i C++, usporedba, 14
C s klasama, 11
C++, 555
Clu, 12
COBOL, 11
FORTRAN, 11, 556
PASCAL, 315
Simula, 11
SmallTalk, 470, 555

K

kasno povezivanje, 373
klasa, 37, 215
apstraktna, 330, 380
definicija unutar deklaracije `friend`, 231
deinicijalizacija osnovne klase, 356
deklaracija, 218, 232
deklaracija izvedene klase, 335
deklaracija unaprijed, 220
deklaracija `using`, 441
destruktor, 216
destruktor izvedene klase, 356
eksplicitni konstruktor, 302
funkcijski član, 221, 372
implementacija, 229
inicijalizacija člana, 237
inicijalizacija konstantnog člana, 244
inicijalizacija osnovne klase, 355
inicijalizacija reference kao člana, 243
izvedena, 332, 335
javna osnovna, 335, 345, 365
javno sučelje, 229
kao argument, 363
konstantni funkcijski član, 253
konstruktor, 216, 233, 281
konstruktor kopije, 241, 285
kontejnerska, 408. *Vidi* kontejnerska klasa
konverzija konstruktorom, 300
konverzija prilikom nasljeđivanja, 357
lokalna, 270
najdalje izvedena, 388
nasljeđena, pravo pristupa, 344
nasljeđivanje instance predloška, 427
nasljeđivanje predloška iz, 428
operator `.`, 221
operator `::`, 223
osnovna, 332, 335
podatkovni član, 219
podrazumijevani konstruktor, 239
podrazumijevano pravo pristupa, 227
područje imena, 219, 262
područje izvedene klase, 341, 360
područje ugnježđene klase, 268
pokazivač `this`, 224
prava pristupa lokalne klase, 270
pravo pristupa, 226
predložak. *Vidi* predložak klase
predložak nasljeđuje predložak, 428
prijatelj klase, 216, 230
pristup članu, 220
pristup nasljeđenom članu, 341

pristup ugnježđenoj klasi, 269
privatna osnovna, 335, 347
puni naziv predloška klase, 413
razlika između klasa i struktura, 293
razlika klase i objekta, 215
redoslijed inicijalizacije članova, 237
statički funkcijski član, 260
statički podatkovni član, 257
tijelo, 218
`type_info`, 465
ugnježđena, 265
umetnuti funkcijski član, 225
virtualna osnovna, 384
`volatile` funkcijski član, 257
zaglavlje, 218
zaštićena osnovna, 335, 349
ključna riječ, 35
kôd
izvedbeni, 20
izvorni, 21, 486
objektni, 22, 486
komentar, 29
upute za komentiranje, 30
konstanta. *Vidi* simbolička konstanta
konstruktor, 216, 233
dojava pogreške, 460
eksplicitan poziv, 281
eksplicitni, 302
`explicit`, 302
i alokacija memorije, 236
i polja objekata, 250
i virtualni poziv, 379
inicijalizacija člana, 237
inicijalizacija dinamičkog objekta, 238
inicijalizacija konstantnog člana, 244
inicijalizacija osnovne klase, 355
inicijalizacija reference kao člana, 243
inicijalizacijska lista, 355
izvedene klase, 355
konstruktor konverzije i operator =, 370
konstruktor kopije, 233, 241, 285
konstruktor kopije, i iznimke, 458
konstruktor kopije, poziv, 241
konverzija konstruktorom, 300
konverzija konstruktorom i nasljeđivanje, 370
podrazumijevani, 233, 239
podrazumijevani, izvedene klase, 356
poziv, 234, 238, 240
poziv za globalne i statičke objekte, 249
prosljeđivanje parametara, 240
redoslijed pozivanja, 356, 389
kontejnerska klasa, 408
asocijativni kontejner, 577
`deque`, 577
`list`, 577
`map`, 578
`multiset`, 578
nizovi, 577
pamćenje elemenata ili pamćenje pokazivača, 411
`set`, 577
`vector`, 577
vezana lista, 336, 408
konverzija. *Vidi* dodjela tipa
argumenta kod instantacije predloška, 400
`bad_cast`, 469
`const_cast`, 471
`dynamic_cast`, 468
eksplicitna, 302
i preopterećenje operatora, 311
iznimke prilikom hvatanja, 450
konstruktorom, 300
konstruktorom i nasljeđivanje, 370
korisnički definirana, 301, 366
naniže, 467
naniže, sigurna, 468
naviše, 467
operator konverzije, 302
pokazivača, 347, 348, 350, 364, 371
razlučivanje operatora konverzije, 304
reference, 347, 348, 350
`reinterpret_cast`, 473
standardna, 347, 348, 350, 357, 364
`static_cast`, 471
statička, 471
trivijalna, 191, 364
ugrađenih tipova, 40
kvalifikator
`const`, 53, 136, 172, 413
`volatile`, 54, 139, 413

L

`labs()`, 591
late binding. *Vidi* povezivanje, kasno
`LC_ALL`, 601
`LC_COLLATE`, 601

LC_CTYPE, 601
LC_MONETARY, 601
LC_NUMERIC, 601
LC_TIME, 601
lconv, 600
ldexp(), 595
ldiv(), 593
ldiv_t, 593
left, 533
limits.h, 43, 205
#line, 485
__LINE__, 266, 326, 478, 485
linked list. *Vidi* vezana lista
linker. *Vidi* povezivač
list, 573, 577
lista. *Vidi* vezana lista
locale, 573
locale.h, 205
localeconv(), 600
localtime(), 589
log(), 193, 595
log10(), 595
logički operator
- *i*, 57
- *ili*, 57
- negacija, 57

logički tip, 57
lokalna varijabla, 78
long, 41, 44
long double, 43, 44
long int, 41, 44
lvrijednost, 39, 127
- promjenjiva, 39

M

macro name. *Vidi* makro ime
main(), 24, 202, 249, 453, 530
makro funkcija, 391, 479
- problemi s parametrima, 480
- standardna, 573

makro ime, 476
- standardno, 573

malloc(), 325
manipulator, 537
- dec, 538
- dodavanje, 539
- endl, 521, 538
- ends, 521, 538
- flush, 520, 538
- hex, 538
- implementacija, 538
- oct, 538
- resetiosflag(), 538
- setbase(), 538
- setfill(), 538
- setiosflags(), 538
- setprecision(), 538
- setw(), 531, 538
- ws, 538

map, 573, 577, 578
math.h, 43, 46, 193, 199, 203, 204, 205
međupohranjivanje, 514
- streambuf, 515

međuspremnik, 514
- pražnjenje, 514

member function. *Vidi* funkcijski član
member selection operator. *Vidi* operator, za pristup članu
memchr(), 601
memcmp(), 602
memcpy(), 601
memmove(), 601
memorijska napuklina, 251
memory, 572
memory leak. *Vidi* memorijska napuklina
memset(), 602
method. *Vidi* metoda
metoda, 221
mktime(), 590
mod toka, 545
- app, 545
- ate, 545
- binary, 545
- in, 545
- nocreate, 545
- noreplace, 545
- out, 545
- trunc, 545

modf(), 595
modul, 486
most derived class. *Vidi* najdalje izvedena klasa
multiple inheritance. *Vidi* nasljeđivanje, višestruko
multiset, 578
mutable, 255

N

\n, 60
najdalje izvedena klasa, 388
name(), 466
nameless namespace. *Vidi* imenik, bezimeni
nameless union. *Vidi* unija, bezimena
namespace, 435. *Vidi* imenik
namespace extension. *Vidi* imenik, proširivanje
naredba
- break, 84, 93
- case, 84
- continue, 94
- default, 84
- do-while, 92
- else, 80
- for, 86
- goto, 94
- goto, definiranje oznake, 95
- if, 79
- nastavak u sljedeći redak, 31
- pisanje, 30
- pretprocesorska, 474
- return, 153, 158
- strukturiranje, 95
- switch, 84
- throw, 449
- throw, bez parametara, 454
- while, 90

nasljeđivanje, 12, 14, 217, 332
- deinicijalizacija izvedene klase, 356
- deklaracija izvedene klase, 335
- deklaracija using, 441
- deklaracija virtualne osnovne klase, 384
- i područje, 360
- i predložak klase, 427
- i preopterećenje, 363
- i pripadnost, 354
- i ugnježđeni tipovi, 361
- inicijalizacija osnovne klase, 355
- instance predloška, 427
- isključivanje člana, 353
- javno, 345
- konstruktor izvedene klase, 355
- konstruktora konverzije, 370
- lista osnovnih klasa, 335
- operatora =, 369
- operatora delete, 371
- operatora new, 371
- područje izvedene klase, 341
- polimorfizam, 371
- predloška iz konkretne klase, 428
- predloška iz predloška, 428
- preopterećenog operatora, 368
- pristup naslijeđenom članu, 341
- private, 335
- privatno, 347
- protected, 335
- public, 335
- razlika nasljeđivanja i preopterećenja, 361, 368
- razlika nasljeđivanja i prijateljstva, 350
- standardna konverzija, 357
- tip nasljeđivanja, 335
- virtualna osnovna klasa, 384
- višestruko, 335, 360
- zaštićeno, 344, 349

NDEBUG, 248, 483, 574
nested class. *Vidi* klasa, ugnježđena
nevezano prijateljstvo, 427
nevirtualni član, 377
::new, 324
new, 128, 572
- alokacija dinamičkog objekta, 238
- alokacija na željenom mjestu, 251
- globalna verzija, 324
- nasljeđivanje, 371
- preopterećenje, 323

::new [], 324
new [], 129, 134
- globalna verzija, 324
- i polja objekata, 250
- preopterećenje, 323

new.h, 251
nocreate, 545
noreplace, 545
not, 70
not_eq, 70
NULL, 119, 575
nul-pokazivač, 119, 575
nul-znak, 140
numeric, 573

O

.o, 22

.obj, 22
object. *Vidi* objekt
object oriented programming. *Vidi* objektno orijentirano programiranje
objekt, 37, 215
- automatski, 128
- deklaracija, 232
- deklariran na osnovu strukture, 292
- dinamička alokacija, 238
- dinamički, 128
- globalni, 182, 185, 248
- implementacija, 215
- inicijalizacija, 356
- kao argument, 363
- lokalni, 181, 185
- polje objekata, 249
- predloška klase, 413
- privremeni, 241, 280, 315
- privremeni, eksplicitno stvoren, 282
- privremeni, prilikom vraćanja iz funkcije, 288
- privremeni, za prijenos po vrijednosti, 284
- razlika objekta i klase, 215
- smještajne klase globalnog objekta, 185
- statički, 248, 413
- statički, u funkciji, 186
- stvaranje, 234
- svojstvene operacije, 372
- točno podudaranje tipova, 364
- trivijalna konverzija, 364
- vanjski, 413

objektni kôd, 22, 486
objektno orijentirana paradigma, 12, 556
objektno orijentirano programiranje, 215, 555
oct, 533, 538
odmatanje stoga, 452
odnos
- *biti*, 563
- *jedan na jedan*, 565
- *jedan na više*, 565
- *koristiti*, 564
- *posjedovati*, 564
- *više na jedan*, 565
- *više na više*, 565

ofstream, 515, 541
omotač, 575
OOP, 555
open(), 544
operator, 308
- !, 57
- !=, 59
- #, 480
- ##, 480
- %, 46
- %:, 70
- %:%:, 70
- %=, 69
- %>, 70
- &, 63
- & (dohvaćanje adrese), 114
- &&, 57
- &=, 69
- (), 154
- (), preopterećenje, 317
- *, 46
- * (deklaracija pokazivača), 114, 120
- * (dereferenciranje), 115
- *=, 69
- +, 46
- ++, 45, 125
- ++, preopterećenje, 321
- +=, 69
- ,, 74, 88
- -, 46
- --, 45, 125
- --, preopterećenje, 321
- -=, 69
- ->, 221
- ->, preopterećenje, 318
- ->*, 274, 278
- ., 221
- .*, 274, 278
- /, 46
- /=, 69
- ::, 186, 223, 259, 262, 270, 435, 436, 437
- ::, i izvedena klasa, 342
- ::, i virtualni član, 379
- :>, 70
- <, 59
- <%, 70
- <:, 70
- <<, 26, 28
- << (izlazni operator), 517
- << (pomak ulijevo), 65

<<=, 69
<=, 59
=, 39, 69
= i konstruktor konverzije, 370
=, nasljeđivanje, 369
=, preopterećenje, 313
==, 59
>, 59
>=, 59
>>, 27
>>, 28
>> (pomak udesno), 65
>> (ulazni operator), 521
>>=, 69
?:, 83
[], 104
[], preopterećenje, 315
^, 64
^=, 69
|, 64
|=, 69
||, 57
~, 62
alternativne oznake, 35, 70
and, 70
and_eq, 70
aritmetički, 45
binarni, 45, 308
bitand, 70
bitor, 70
bitovni, 62
compl, 70
const_cast, 471
definicija preopterećenog operatora, 309
dekrement, 45
delete, 129, 247, 381
delete, nasljeđivanje, 371
delete, preopterećenje, 323
delete [], 129, 250
delete [], preopterećenje, 324
dodjele tipa, 50, 139, 400
dozvoljeni operatori za preopterećenje, 307
dynamic_cast, 468
hijerarhija, 74
inkrement, 45
izlučivanja, 521
konverzije, 302
logički, 57
new, 128
new, nasljeđivanje, 371
new, preopterećenje, 323
new [], 129, 134, 250
new [], preopterećenje, 323
not, 70
not_eq, 70
obnavljajućeg pridruživanja, 69, 125
operator, 308
or, 70
or_eq, 70
poredbeni, 58, 125
postfiks, 45
poziv operatorske funkcije, 310
prefiks, 45
preopterećenje, 299, 307
preopterećenje i nasljeđivanje, 368
pridruživanja, 39
razlika = i ==, 59, 83
razlikovanje << i >>, 28
razlučivanje operatora konverzije, 304
redoslijed izvođenja operatora, 74
reinterpret_cast, 473
sizeof, 43, 73
static_cast, 471
typeid, 465
umetanja, 517
unarni, 45, 308
uvjetni, 83
xor, 70
xor_eq, 70
za određivanje područja, 186, 270, 342, 379
za pristup članu, 220
za razlučivanje imena, 223
za razlučivanje područja, 435, 436, 437
operator overloading. *Vidi* operator, preopterećenje
oporavak od iznimke, 448
or, 70
or_eq, 70
osnovna klasa, 332, 335
deinicijalizacija, 356
dominacija, 387
inicijalizacija, 355
javna, 335, 345, 365
konverzija pokazivača, 358

konverzija pokazivača na, 347, 348, 364, 371
privatna, 335, 347
virtualna, 384
virtualna, deklaracija, 384
virtualna, i pravo pristupa, 386
virtualna, inicijalizacija, 387
virtualna, pristup članu, 385
zaštićena, 335, 349
`ostream`, 515, 546, 573
`ostrstream`, 553
`out`, 545
output stream. *Vidi* tok, izlazni
overriding. *Vidi* zaobilaženje prilikom nasljeđivanja

P

pametni pokazivač, 320
parametar. *Vidi* argument
partial specialization. *Vidi* predložak funkcije, djelomična specijalizacija
`peek()`, 527
petlja, 77
beskonačna, 88
`do-while`, 92, 93
`for`, 86, 93
razlika `for` i `while`, 91
s uvjetom na kraju, 92
s uvjetom na početku, 86, 90
ugnježđivanje, 89
`while`, 90, 93
plitka kopija, 242
pobrojenje, 55
ugnježđeno u predložak, 423
podatkovni član, 219
inicijalizacija konstantnog člana, 244
inicijalizacija konstruktorom, 237
inicijalizacija reference kao člana, 243
isključivanje iz nasljeđivanja, 353
javni, 226
`mutable`, 255
podrazumijevano pravo pristupa, 227
pokazivač na, 272
pravo pristupa, 226
pristup, 220
privatni, 226
puno ime, 223, 261
razlika statičkog povezivanja i statičkog člana, 491
redoslijed inicijalizacije, 237
statički, 257
statički i datoteka zaglavlja, 258
statički, inicijalizacija, 258
statički, pristup, 259
statički, u predlošku klase, 419
statički, u predlošku klase, inicijalizacija, 419
statički, u predlošku, specijalizirana inicijalizacija, 420
virtualne osnovne klase, 385
zaštićeni, 226
podatkovni segment, 213
područje, 78, 262
i nasljeđivanje, 360, 368
imenik, 435
područje klase, 219
pravila za razlučivanje, 264
problem spajanja područja, 434
ugnježđene klase, 268
zagađenje globalnog područja, 435
pogonitelj, 576
pogreška
prilikom izvođenja, 22
prilikom povezivanja, 22
prilikom prevođenja, 22
point of instantiation
Vidi predložak funkcije, mjesto instantacije.
Vidi predložak klase, mjesto instantacije.
pointer. *Vidi* pokazivač
pointer aliasing, 242
pokazivač, 114
aritmetika, 124
`const` na `const` objekt, 138
deklaracija, 114, 120
dekrement, 125
dereferencirani, poziv člana, 379
dodavanje broja, 124
dodjela `const` pokazivača, 138
dozvoljene operacije s pokazivačima na član, 274
inicijalizacija, 116, 118
inkrement, 125
kao povratna vrijednost funkcije, 179
konstantni, 137
konverzija u pokazivač na osnovnu klasu, 358, 364, 371
međusobne operacije, 117
mijenjanje unutar funkcije, 166

na `const` objekt, 137
na član klase, 271
na član, implementacija, 274
na član, korištenje, 274, 278
na funkcijski član, 277
na funkcijski član, deklaracija, 278
na funkciju, 196
na funkciju s podrazumijevanim argumentom, 201
na podatkovni član, 272
na podatkovni član, deklaracija, 272
na pokazivač, 134, 167
neinicijalizirani, 118
`NULL`, 119
nul-pokazivač, 119
odbacivanje konstantnosti, 139
oduzimanje pokazivača, 125
operator `&`, 114
operator `*` (deklaracija), 114, 120
operator `*` (dereferenciranje), 115
preusmjeravanje, 116
sinonim za tip, 148
sličnost s referencom, 145
standardna konverzija, 347, 348, 350
`this`, 224, 261, 277
usporedba, 125
usporedba s nulom, 125
veza s poljem, 121
virtualni poziv, 378
viseći, 148, 286
`void *`, 118
`vptr`, 375
zauzeće memorije, 119

pokazivač datoteke, 548
beg, 549
cur, 549
end, 549
seekdir, 549
streamoff, 549
streampos, 548

pokazivač instrukcija, 213
pokazivač na virtualnu tablicu, 375
pokazivač stoga, 213
polimorfizam, 12, 14, 218, 371, 463
polimorphysm. *Vidi* polimorfizam
polje, 102
adresa početnog člana, 121
članovi, 102
deklaracija, 102
dinamička alokacija, 129
dinamička alokacija višedimenzionalnog polja, 133
dvodimenzionalno, 108
indeks, 102, 104
inicijalizacija, 103
jednodimenzionalno, 102
kao argument funkcije, 168
kao povratna vrijednost funkcije, 181
nedozvoljeni indeks, 107
objekata, 249
objekata, dinamički alocirano, 250
objekata, inicijalizacija, 250
pohranjivanje u memoriju, 123
raspon indeksa, 122
veza s pokazivačem, 121
višedimenzionalno, 109, 113
višedimenzionalno, inicijalizacija, 109
višedimenzionalno, različitih duljina redaka, 135
zauzeće memorije, 113
znakovnih nizova, 143

polje bitova, 297
ponovna iskoristivost, 14, 555, 558
popratne pojave, 163
posebna sekvenca, 60
`"`, 60
`\'`, 60
`\0`, 60
`\?`, 60
`\\`, 60
`\a`, 60
`\b`, 60
`\`*`ddd`*, 60
`\f`, 60
`\n`, 60
`\r`, 60
`\t`, 60
`\v`, 60
`\x`*`ddd`*, 60

postfiks operator, 45
potpis funkcije, 153
povezivač, 21
povezivanje, 486
dinamičko, 373
kasno, 373
s C funkcijama, 510
s drugim programskim jezicima, 508
statičko, 373, 491
unutarnje, 185, 489
vanjsko, 185, 489, 492

povezivanje kôda, 21
pow(), 46, 163, 596
poziv
  C funkcije, 509
  destruktora, 246
  dinamički, 375
  iz destruktora, 379
  iz konstruktora, 379
  konstruktora, 234, 238, 240
  konstruktora kopije, 241
  preko objekta, 379
  preko pokazivača, 378
  preko reference, 378
  razlika statičkog poziva i statičkog člana, 376
  rekurzivan, 195
  statički, 376
  statički, unatoč virtualnoj deklaraciji, 379
  virtualni, 375
  virtualni iz člana klase, 378
  virtualnog destruktora, 381
poziv funkcije, 154
#pragma, 484
pravo pristupa, 226
  definicija unutar deklaracije friend, 231
  friend, 230
  i deklaracija using, 442
  i konstruktor, 245
  i virtualna osnovna klasa, 386
  javno, 226
  lokalne klase, 270
  podrazumijevano, 227
  prilikom nasljeđivanja, 344
  privatno, 226
  zaštićeno, 226, 344
praznina, 30, 31
precision(), 535
predložak funkcije, 211, 391
  <>, 211, 392, 398
  definicija, 392
  definicija funkcije, 393
  deklaracija funkcije, 393
  djelomična specijalizacija, 405
  extern, 394
  formalni argument, 392, 403
  inline, 394
  instantacija, 394, 395
  instantacija, eksplicitna, 402
  instantacija, implicitna, 397
  konstantan argument, 396
  konverzija argumenta, 400
  mjesto instantacije, 429
  navođenje argumenata, 395, 398
  nevezano prijateljstvo, 427
  određivanje funkcije, 399
  određivanje specijalizirane funkcije, 406
  organizacija kôda, 497
  podrazumijevani argument, 396, 412
  povratna vrijednost, 394
  predložak funkcijskog člana, 424
  preopterećenje, 403
  provjera sintakse, 400
  rezultat funkcije, 404
  specijalizacija, 405
  static, 394
  stvarni argument, 394
  typename, 395
  upotreba, 393
  uspoređivanje argumenata, 399
  vezano prijateljstvo, 426
predložak klase, 408
  <>, 409
  argument, 409
  const, 413
  definicija, 409
  djelomična specijalizacija, 418
  extern, 413
  instantacija konstantnim izrazom, 421
  instantacija, eksplicitna, 416
  instantacija, implicitna, 412
  izraz kao argument, 409
  konstantan izraz kao argument, 420
  mjesto instantacije, 429
  nasljeđivanje, 427
  nasljeđivanje instance, 427
  nasljeđivanje konkretne klase, 428
  nevezano prijateljstvo, 427
  organizacija kôda, 497
  predložak funkcijskog člana, 424
  predložak nasljeđuje predložak, 428
  prijateljstvo, 426
  pristup ugnježđenom tipu, 423
  provjera sintakse, 413
  puni naziv klase, 413
  specijalizacija, 416
  specijalizacija cijele klase, 417
  specijalizacija funkcijskog člana, 417
  static, 413
  statički član, 419

statički član, inicijalizacija, 419
statički član, specijalizirana inicijalizacija, 420
tip ugnježđen u predložak, 422
ugnježđena pobrojenja, 423
ugnježđeni, 423
vezano prijateljstvo, 426
`volatile`, 413
prefiks `L` za dugi znak, 62
prefiks operator, 45
prekid, 54
preljev, 46
preopterećenje
funkcije, 189
i nasljeđivanje, 363
operatora, 299, 307
operatora `--`, 321
operatora `()`, 317
operatora `[]`, 315
operatora `++`, 321
operatora `<<`, 519
operatora `=`, 313
operatora `->`, 318
operatora `>>`, 523
operatora `delete`, 323
operatora `delete []`, 324
operatora i nasljeđivanje, 368
operatora `new`, 323
operatora `new []`, 323
operatora, unutar ili izvan klase, 310
poziv operatorske funkcije, 310
predloška funkcije, 403
razlika preopterećenja i nasljeđivanja, 361, 368
točno podudaranje tipova, 364
pretprocesor, 474
`"`, 475
`#`, 480
`##`, 480
`#define`, 53, 476, 479
`#define`, trajanje definicije, 477
`#elif`, 482
`#else`, 482
`#endif`, 482
`#error`, 484
`#if`, 481
`#ifdef`, 481
`#ifndef`, 481
`#include`, 156, 475, 494
`#line`, 485
`#pragma`, 484
`#undef`, 478
\ (nastavak reda), 474
`__cplusplus`, 478, 510
`__DATE__`, 478
`__FILE__`, 266, 326, 478, 485
`__LINE__`, 266, 326, 478, 485
`__STDC__`, 478
`__TIME__`, 478
`<>`, 475
makro funkcija, 391, 479
makro ime, 476
naredba, 474
`NDEBUG`, 483
pretvorba naniže, 467
pretvorba naviše, 467
prevoditelj, 21
prevođenje kôda, 21
prijatelj klase, 216, 230
i predlošci, 426
`printf()`, 176
`private`, 226, 335, 347, 384
private base class. *Vidi* osnovna klasa, privatna
programersko sučelje (API), 558
programiranje
objektno orijentirano, 13
pogonjeno događajima, 13
proceduralno, 12
projekt, 487
`protected`, 226, 335, 344, 349, 384. *Vidi* pravo pristupa, zaštićeno
protected base class. *Vidi* osnovna klasa, zaštićena
prototip funkcije, 152, 494
`public`, 221, 226, 335, 345, 384
public base class. *Vidi* osnovna klasa, javna
public interface. *Vidi* javno sučelje
pure virtual function member. *Vidi* funkcijski član, čisti virtualni
`put()`, 520
`putback()`, 527

Q

`qsort()`, 602
`queue`, 573

R

`\r`, 60
`raise()`, 603
raising an exception. *Vidi* iznimka, podizanje
`rand()`, 93, 187, 575, 596
`RAND_MAX`, 93, 575, 596
`randomize()`, 93
razlika klase i objekta, 215
`rdstate()`, 516
`read()`, 526, 552
realni broj, 40. *Vidi* tip, `double`
redoslijed izvođenja operatora, 74
referenca, 144
  deklaracija, 144
  inicijalizacija, 145
  kao povratna vrijednost funkcije, 179
  na konstantan objekt, 145
  na pokazivač, 168
  sličnost s pokazivačem, 145
  standardna konverzija, 347, 348, 350
  trivijalna konverzija, 364
  virtualni poziv, 378
  viseća, 411
  za hakere, 145
registarska smještajna klasa. *Vidi* smještajna klasa, registarska
`register`, 182
`reinterpret_cast`, 473
rekurzija, 195
relacija
  biti, 354, 563
  jedan na jedan, 565
  jedan na više, 565
  korisiti, 564
  posjedovati, 564
  sadrži, 354
  više na jedan, 565
  više na više, 565
`resetiosflag()`, 538
`return`, 153, 158
reusability. *Vidi* ponovna iskoristivost
`right`, 533
rukovanje iznimkama, 448
run-time type identification. *Vidi* identifikacija tipa

S

`scientific`, 533, 535
scope. *Vidi* područje
scope resolution operator. *Vidi* operator, za određivanje područja
`seekdir`, 549
`seekg()`, 529, 548
`seekp()`, 521, 548
`set`, 573, 577
`set_new_handler()`, 604
`set_terminate()`, 605
`set_unexpected()`, 605
`setbase()`, 538
`setf()`, 532
`setfill()`, 538
`setiosflags()`, 538
`setlocale()`, 601
`setprecision()`, 538
`setw()`, 90, 538
shallow copy. *Vidi* plitka kopija
`short`, 44
`short int`, 44
`showbase`, 533
`showpoint`, 533, 535, 536
`showpos`, 533
side-effects. *Vidi* popratne pojave
`SIG_DFL`, 603
`SIG_ERR`, 603
`SIG_IGN`, 603
`SIGABRT`, 603
`SIGFPE`, 603
`SIGILL`, 603
`SIGINT`, 603
`signal()`, 603
`SIGSEGV`, 603
`SIGTERM`, 603
simbolička konstanta, 52, 489. *Vidi* `const`
  inicijalizacija, 54
`sin()`, 199, 597
`sinh()`, 597
`size_t`, 74, 323, 371
`sizeof`, 43, 73
`skipws`, 533
skrivanje podataka, 12, 557, 571
skupljanje smeća, 11, 320
smart pointer. *Vidi* pametni pokazivač
smještajna klasa, 181

automatska, 182
imenik umjesto `static`, 437
`mutable`, 255
registarska, 182
statička, 185
vanjska, 185
source code. *Vidi* izvorni kôd
specijalizacija predloška funkcije, 405
`sqrt()`, 204, 597
`srand()`, 596
`sstream`, 553, 573
stack, 573. *Vidi* stog
stack pointer. *Vidi* pokazivač stoga
stack unwinding. *Vidi* iznimka, odmatanje stoga
standard C++, 18
makro funkcije, 573
makro imena, 573
standardna biblioteka predložaka, 577
standardne klase, 578
standardne strukture, 578
standardne vrijednosti, 576
standardni tipovi, 576
zaglavlja, 571
Standard Template Library. *Vidi* standardna biblioteka predložaka
standardna biblioteka, 571
standardna biblioteka predložaka, 577
standardna funkcija, 580
`abort()`, 574, 598
`abs()`, 591
`acos()`, 591
`asctime()`, 588
`asin()`, 592
`atan()`, 205, 592
`atan2()`, 204, 592
`atexit()`, 598
`atof()`, 203, 586
`atoi()`, 587
`atol()`, 587
`ceil()`, 592
`clock()`, 589
`cos()`, 593
`cosh()`, 593
`ctime()`, 588
`difftime()`, 589
`div()`, 593
`exit()`, 210, 249, 544, 600
`exp()`, 193, 594
`fabs()`, 199, 591
`floor()`, 592
`fmod()`, 594
`free()`, 325
`frexp()`, 594
`gmtime()`, 589
`isalnum()`, 582
`isalpha()`, 529, 582
`iscntrl()`, 582
`isdigit()`, 528, 582
`isgraph()`, 582
`islower()`, 582
`isprint()`, 582
`ispunct()`, 582
`isspace()`, 528, 582
`isupper()`, 582
`isxdigit()`, 582
`labs()`, 591
`ldexp()`, 595
`ldiv()`, 593
`localeconv()`, 600
`localtime()`, 589
`log()`, 193, 595
`log10()`, 595
`malloc()`, 325
`memchr()`, 601
`memcmp()`, 602
`memcpy()`, 601
`memmove()`, 601
`memset()`, 602
`mktime()`, 590
`modf()`, 595
`pow()`, 46, 163, 596
`printf()`, 176
`qsort()`, 602
`raise()`, 603
`rand()`, 187, 575, 596
`set_new_handler()`, 604
`set_terminate()`, 605
`set_unexpected()`, 605
`setlocale()`, 601
`signal()`, 603
`sin()`, 199, 597
`sinh()`, 597
`sqrt()`, 204, 597
`srand()`, 596
`strcat()`, 208, 583

strchr(), 583
strcmp(), 209, 583
strcoll(), 583
strcpy(), 207, 583
strcspn(), 584
strlen(), 169, 207, 584
strlwr(), 209
strncat(), 583
strncmp(), 583
strncpy(), 584
strpbrk(), 584
strrchr(), 585
strspn(), 584
strstr(), 585
strtod(), 587
strtok(), 585
strtol(), 587
system(), 605
tan(), 597
tanh(), 598
terminate(), 453, 604
time(), 591
tolower(), 586
toupper(), 586
unexpected(), 459, 605
standardna klasa, 578
basic_string, 578
complex, 578
valarray, 573
standardna makro funkcija, 573
assert(), 236, 248, 483, 574
va_arg(), 179, 576
va_end(), 179, 576
va_start(), 179, 576
standardna struktura, 578
standardna vrijednost, 576
standardni tip, 576
standardno makro ime, 573
standardno zaglavlje, 571
state. *Vidi* tok, stanje
static, 185, 186, 258, 394, 413, 491
razlika statičkog povezivanja i statičkog člana, 491
static binding. *Vidi* povezivanje, statičko
static call. *Vidi* poziv, statički
static_cast, 471
statički poziv, 376
razlika statičkog poziva i statičkog člana, 376
unatoč virtualnoj deklaraciji, 379
statičko povezivanje, 373, 491
stdarg.h, 178, 205
__STDC__, 478
stddef.h, 74
stdexcept, 572
stdlib.h, 93, 187, 205, 210, 325
STL. *Vidi* standardna biblioteka predložaka
stog, 128, 213
storage class. *Vidi* smještajna klasa
strcat(), 208, 583
strchr(), 583
strcmp(), 209, 583
strcoll(), 583
strcpy(), 207, 583
strcspn(), 584
streambuf, 515, 529, 573
streamoff, 549
streampos, 548
string. *Vidi* znakovni niz
string (klasa), 572
string (zaglavlje), 572, 578
string.h, 142, 205, 206
strlen(), 169, 207, 584
strlwr(), 209
strncat(), 583
strncmp(), 583
strncpy(), 584
Stroustrup, Bjarne, 11
strpbrk(), 584
strrchr(), 585
strspn(), 584
strstr(), 585
strstream.h, 553
strtod(), 587
strtok(), 585
strtol(), 587
struct, 292
structure. *Vidi* struktura
struktura, 292
razlika između struktura i klasa, 293
razlika u C i C++ jezicima, 293
strukturiranje kôda, 95
sučelje. *Vidi* javno sučelje
sufiks brojčane konstante, 51
f, 51

- F, 51
- l, 51
- L, 51
- u, 51
- U, 51

switch, 84
system(), 605

T

\t, 60
tablica virtualnih članova, 375
tan(), 597
tanh(), 598
tellg(), 529, 548
tellp(), 521, 548
template
- *Vidi* predložak funkcije.
- *Vidi* predložak klase.

template, 211, 392, 405, 409
terminate(), 453, 604
this, 224, 261, 277
throw, 449
- bez parametara, 454
- lista mogućih iznimaka, 458

throwing an exception. *Vidi* iznimka, bacanje
tie(), 530
__TIME__, 478
time(), 591
time.h, 205, 588
time_t, 588
tip, 37
- bool, 57
- broj, 40
- char, 44, 60
- char *, 139
- cjelobrojna promocija, 47
- double, 43, 44
- float, 42, 44
- identifikacija. *Vidi* identifikacija tipa
- int, 41, 44
- iznimke, 449
- klasa, 215
- logički, 57
- long, 41, 44
- long double, 43, 44
- long int, 41, 44
- pobrojani, 55
- pokazivač. *Vidi* pokazivač
- pokazivač na član klase, 271
- pokazivač na funkciju, 196
- polje, 102
- pravila konverzije, 40, 47
- pravila provjere tipa, 40
- razlika između char i char *, 142
- referenca. *Vidi* referenca
- short, 44
- short int, 44
- sinonim za pokazivački tip, 148
- type_info, 465
- typedef, 71, 490
- ugnježđen u predlošku, 422
- ugnježđeni, i nasljeđivanje, 361
- ugrađeni, 40
- unsigned, 44
- usporedba, 465
- void, 159
- wchar_t, 62, 515
- wchar_t *, 143
- znakovni, 60
- znakovni niz, 60, 139

tm, 590
tok
- bad(), 516
- badbit, 516
- beg, 549
- binarno pisanje i čitanje, 550
- cerr, 516
- cin, 27, 515, 521
- clear(), 516, 548
- clog, 516
- close(), 545
- cout, 26, 516, 517
- cur, 549
- čitanje i pisanje u datoteku, 541
- end, 549
- eof(), 516
- eofbit, 516, 548
- fail(), 516
- failbit, 516, 522
- fill(), 531
- flags(), 532
- flush(), 520
- fstream, 515, 541, 546
- gcount(), 526, 552

get(), 524
getline(), 526
good(), 516
goodbit, 516
ifstream, 91, 515, 541
ifstream, konstruktor, 544
ignore(), 526
ios, 515
iostate, 516
isalnum(), 529
ispis korisničkih tipova, 519
istream, 515, 546
istringstream, 553
istrstream, 553
izlazni, 26, 513
konvezija u void *, 516
main(), 530
manipulator. *Vidi* manipulator
međupohranjivanje, 514
mod otvaranja, 545. *Vidi* mod
ofstream, 515, 541
ofstream, konstruktor, 544
open(), 544
operator !, 517
operator izlučivanja, 521
operator umetanja, 517
ostream, 515, 546
ostrstream, 553
otvaranje datoteke, 544
peek(), 527
precision(), 535
put(), 520, 554
putback(), 527
rdstate(), 516
read(), 526, 552
seekdir, 549
seekg(), 529, 548
seekp(), 521, 548
setf(), 532
setw(), 90
stablo nasljeđivanja, 515
stanje, 516
str(), 554
streambuf, 515, 529
streamoff, 549
streampos, 548
širina ispisa, 530
tellg(), 529, 548
tellp(), 521, 548
testiranje stanja, 516
tie(), 530
ulazni, 27, 513
unsetf(), 532
upis korisničkih tipova, 523
vezanje cin i cout, 530
vezivanje, 529
werr, 516
width(), 530
wifstream, 515
win, 516
wios, 515
wistream, 515
wlog, 516
wofstream, 515
wostream, 515
wout, 516
write(), 520, 552
wstreambuf, 515
zastavica, 532. *Vidi* zastavica
zatvaranje datoteke, 545
znak za popunjavanje, 531

tolower(), 586
toupper(), 586
trigraf nizovi, 71
trigraph. *Vidi* trigraf nizovi
true, 57
trunc, 545
try, 448
try block. *Vidi* blok, pokušaja
type cast. *Vidi* dodjela tipa
type_info, 465
before(), 466
name(), 466
typedef, 71, 148, 490
typeid, 465
typeinfo, 572
typeinfo.h, 465
typename, 395

U

ugnježđena klasa. *Vidi* klasa, ugnježđena
ugnježđivanje imenika. *Vidi* imenik, ugnježđivanje
uključivanje asemblerskog kôda, 511
ulazni tok, 27. *Vidi* tok, ulazni

unbound template friendship. *Vidi* nevezano prijateljstvo
`#undef`, 478
`unexpected()`, 459, 605
unija, 294
- anonimna, 296
- bezimena, 296
- diskriminanta unije, 295
- razlika u C i C++ jezicima, 294

union, 294. *Vidi* unija
union discriminant. *Vidi* unija, diskriminanta unije
`unitbuf`, 533
unnamed temporary. *Vidi* objekt, privremeni
`unsetf()`, 532
`unsigned`, 44
upcast. *Vidi* pretvorba naviše
upozorenje pri prevođenju i povezivanju, 23
`uppercase`, 533, 536
`using`
- deklaracija, 439
- deklaracija i proširivanje, 440
- deklaracija unutar klase, 441
- direktiva, 443
- direktiva i proširivanje, 444
- direktiva u sklopu imenika, 444

`using` declaration. *Vidi* `using`, deklaracija
usporedbe, 58
`utility`, 572
uvjetni operator, 83
uvjetno prevođenje, 481
- pronalaženje pogrešaka, 483

V

`\v`, 60
`va_arg()`, 179, 576
`va_end()`, 179, 576
`va_list`, 178, 576
`va_start()`, 179, 576
`valarray`, 573
`values.h`, 43
vanjsko povezivanje, 492
varijabla, 35, 37
- deklaracija, 38
- lokalna, 78
- područje, 78
- pridruživanje vrijednosti, 38
- vidljivost unutar bloka, 77

`vector`, 573, 577
vezana lista, 336
- izbacivanje člana, 339
- pokazivač `glava`, 337
- pokazivač `rep`, 337
- pomoću predložaka, 408
- umetanje člana, 337

vezano prijateljstvo, 426
`virtual`, 375, 381, 384, 556
- deklaracija virtualne osnovne klase, 384

virtual base class. *Vidi* virtualna osnovna klasa
virtual call. *Vidi* virtualni poziv
virtual function member. *Vidi* virtualni, funkcijski član
virtualna osnovna klasa, 384
- deklaracija, 384
- dominacija, 387
- i pravo pristupa, 386
- inicijalizacija, 387
- pristup članu, 385

virtualni
- čisti virtualni funkcijski član, 380
- destruktor, 380
- funkcijski član, 343, 374

virtualni poziv, 375
- iz destruktora, 379
- iz konstruktora, 379
- određivanje člana, 378
- preko pokazivača, 378
- preko reference, 378
- zaobilaženje, eksplicitno, 379

višestruko nasljeđivanje, 335
`void`, 159
`volatile`, 54, 139, 413
- `const_cast`, 471
- funkcijski član, 257

`vptr`, 375
`vtable`, 375

W

`wchar_t`, 62, 515
`wchar_t *`, 143
`werr`, 516
`while`, 90

width(), 530
wifstream, 515
win, 516
wios, 515
wistream, 515
wlog, 516
wofstream, 515
wostream, 515
wout, 516
wrapper. *Vidi* omotač
write(), 520, 552
ws, 538
wstreambuf, 515
wstring, 572

X

xor, 70
xor_eq, 70

Z

zagađenje globalnog područja, 435
zaglavlje, 26, 155, 494
- algorithm, 573
- assert.h, 236, 483
- bits, 573
- cassert, 572
- cctype, 572
- cerrno, 572
- clocale, 573
- cmath, 573
- complex, 573, 578
- complex.h, 205, 206
- csetjmp, 572
- csignal, 572
- cstdarg, 572
- cstdio, 573
- cstdlib, 572, 573
- cstring, 572
- ctime, 572, 588
- ctype.h, 528
- cwchar, 572, 573
- cwctype, 572
- deque, 573, 577
- exception, 572
- float.h, 205
- fstream, 573
- fstream.h, 91, 206, 541
- i statički objekt, 258
- iomanip, 573
- iomanip.h, 205, 206, 538
- ios, 573
- iosfwd, 573
- iostream, 573
- iostream.h, 26, 205, 206, 513, 515, 517, 521
- istream, 573
- iterator, 573
- limits.h, 43, 205, 206
- list, 573, 577
- locale, 573
- locale.h, 205, 206
- map, 573, 577
- math.h, 43, 46, 193, 199, 203, 204, 205, 206
- memory, 572
- new, 572
- new.h, 251
- numeric, 573
- ostream, 573
- queue, 573
- set, 573, 577
- sstream, 553, 573
- stack, 573
- standardna, 571
- standardne datoteke, 205
- stdarg.h, 178, 205, 206
- stddef.h, 323
- stdexcept, 572
- stdio.h, 126
- stdlib.h, 93, 187, 205, 206, 210, 325
- streambuf, 573
- string, 572, 578
- string.h, 142, 205, 206
- strstream.h, 553
- što u nj ne staviti, 497
- što u nj staviti, 495
- time.h, 205, 206, 588
- typeinfo, 572
- typeinfo.h, 465
- utility, 572
- valarray, 573
- values.h, 43
- vector, 573, 577

zaobilaženje prilikom nasljeđivanja, 333
zastavica, 532
  boolalpha, 533
  dec, 533
  fixed, 533, 535
  grupa adjustfield, 533
  grupa basefield, 533
  grupa floatfield, 533
  hex, 533
  internal, 533
  left, 533
  oct, 533
  right, 533
  scientific, 533, 535
  showbase, 533
  showpoint, 533, 535, 536
  showpos, 533
  skipws, 533
  unitbuf, 533
  uppercase, 533, 536
znakovni niz, 60, 139
  funkcije za manipulaciju, 206
  kao argument funkcije, 169
  kao povratna vrijednost funkcije, 181
  polje znakovnih nizova, 143
  rastavljanje, 31
  zaključni nul-znak, 140
Error! Cannot open file referenced on page 641