



Irina Branović

OBJEKTNO ORIJENTISANO PROGRAMIRANJE: C++

UNIVERZITET SINGIDUNUM

Irina Branović

OBJEKTNO ORJENTISANO PROGRAMIRANJE C++

Prvo izdanje

Beograd, 2011.

OBJEKTNO ORJENTISANO PROGRAMIRANJE C++

Autor:

Doc. dr Irina Branović

Recenzenti:

Prof. dr Dejan Živković

Doc. dr Snežana Mladenović

Izdavač:

UNIVERZITET SINGIDUNUM

Beograd, Danijelova 32

Za izdavača:

Prof. dr Milovan Stanišić

Lektor:

Stanka Vatović

Tehnička obrada:

Irina Branović

Dizajn korica:

Aleksandar Mihajlović

Godina izdanja:

2011.

Tiraž:

300 primeraka

Štampa:

Mladost Grup

Loznica

ISBN: 978-86-7912-373-2

Copyright:

© 2011. Univerzitet Singidunum

Izdavač zadržava sva prava.

Reprodukcija pojedinih delova ili celine ove publikacije nije dozvoljena.

Sadržaj

Predgovor	7
Glava 1. Uvod u jezik C++	9
1.1. Istorijat jezika C++	9
1.2. Standardizacija jezika	10
1.3. Objektno orijentisane osobine jezika C++	11
1.4. Unošenje, prevođenje i izvršavanje C++ programa	13
1.4.1. Integrisana razvojna okruženja za programiranje (IDE)	15
1.4.2. Prevođenje i interpretiranje	15
1.4.3. Konzolni i grafički programi	16
1.5. Prvi C++ program	16
Pitanja	18
Glava 2. Tipovi podataka i operatori	23
2.1. Identifikatori	23
2.2. Specijalni znakovi	25
2.3. Ugrađeni tipovi podataka	26
2.3.1. Promenljive i konstante	28
2.3.2. Celobrojni podaci (int)	28
2.3.3. Znakovni podaci (char)	29
2.3.4. Brojevi u pokretnom zarezu (float, double)	30
2.3.5. Logički tip (bool)	30
2.4. Operatori	31
2.4.1. Operator dodele	31
2.4.2. Aritmetički operatori	32
2.4.3. Relacioni i logički operatori	33
2.4.4. Definisanje sinonima za tipove podataka: typedef	35
2.4.5. Operator sizeof	35
2.4.6. Operator nabiranja (,)	36
2.4.7. Osnovne operacije ulaza i izlaza	37
2.4.8. Prioritet i asocijativnost operatora	37
2.5. Konverzija tipova (cast)	39
2.6. Simboličke konstante	40
Pitanja	41
Glava 3. Naredbe za kontrolu toka programa	49
3.1. Blokovi naredbi	49

3.2.	Naredba if	50
3.2.1.	Ugnježdene naredbe if	52
3.2.2.	If else if lestvica	52
3.2.3.	Uslovni izraz (operator ? :)	54
3.3.	Naredba switch	54
3.4.	Petlja for	56
3.5.	Petlja while	58
3.6.	Petlja do-while	59
3.7.	Naredbe break i continue	60
3.8.	Ugnježdene petlje	61
3.9.	Naredba bezuslovnog skoka (goto)	62
	Pitanja	62
Glava 4.	Nizovi, stringovi i pokazivači	67
4.1.	Jednodimenzionalni nizovi	67
4.2.	Višedimenzionalni nizovi	69
4.3.	Stringovi	70
4.3.1.	Učitavanje stringa sa tastature	71
4.3.2.	Funkcije C++ biblioteke za stringove	72
4.4.	Inicijalizacija nizova	73
4.4.1.	Inicijalizacija nizova bez dimenzija	74
4.5.	Nizovi stringova	75
4.6.	Pokazivači	76
4.6.1.	Pokazivač NULL	77
4.6.2.	Operacije sa pokazivačima	78
4.6.3.	Osnovni tip pokazivača	79
4.6.4.	Dodela vrednosti promenljivoj preko pokazivača	80
4.6.5.	Pokazivačka aritmetika	80
4.6.6.	Pokazivači i nizovi	80
4.7.	Dinamička alokacija memorije	83
	Pitanja	85
Glava 5.	Funkcije	91
5.1.	Argumenti funkcije	92
5.2.	Povratak iz funkcije	93
5.2.1.	Funkcija exit()	95
5.3.	Doseg promenljivih (scope)	96
5.3.1.	Lokalni doseg	96
5.3.2.	Globalni doseg	98
5.3.3.	Statičke promenljive	100
5.4.	Imenici	101
5.5.	Nizovi i pokazivači kao argumenti funkcija	102
5.6.	Reference	105
5.6.1.	Reference u funkcijama	106
5.6.2.	Samostalne reference	107
5.6.3.	Pokazivači i reference	107
5.6.4.	Reference kao povratni tip funkcija	107

5.6.5. Korišćenje modifikatora const u funkcijama	108
5.7. Prototipovi funkcija	109
5.8. Preklapanje funkcija	110
5.8.1. Podrazumevani argumenti funkcija	111
5.9. Rekurzivne funkcije	112
Pitanja	113
Glava 6. Klase i objekti	119
6.1. Kako i zašto je nastalo objektno orijentisano programiranje?	119
6.2. Pojam klase	120
6.3. Kontrola pristupa članovima klase	124
6.4. Inline funkcije članice	125
6.5. Konstruktori i destruktori	125
6.5.1. Kada se izvršavaju konstruktori i destruktori?	127
6.5.2. Konstruktori sa parametrima	128
6.5.3. Preklapanje konstruktora	129
6.5.4. Podrazumevani (default) konstruktori	131
6.6. Zajednički članovi klasa	131
6.7. Prosledivanje objekata funkciji	135
6.8. Dodela objekata jedan drugome	136
6.9. Pokazivači na objekte	137
6.10. Pokazivač this	137
6.11. Konstruktor kopije	139
6.12. Klasa string	141
Pitanja	144
Glava 7. Nasleđivanje, virtuelne funkcije i polimorfizam	151
7.1. Nasleđivanje	151
7.2. Kontrola pristupa članovima klase	153
7.2.1. Specifikator pristupa protected	153
7.2.2. Javno i privatno izvođenje	154
7.2.3. Inspektori i mutatori	155
7.3. Konstruktori i destruktori izvedenih klasa	156
7.4. Višestruko nasleđivanje	160
7.5. Pokazivači na objekte izvedenih klasa	161
7.6. Nadjačavanje funkcija u izvedenim klasama	161
7.7. Virtuelne funkcije i polimorfizam	163
7.7.1. Statičko i dinamičko povezivanje	165
7.7.2. Čiste virtuelne funkcije i apstraktne klase	166
7.7.3. Virtuelni destruktori	167
Pitanja	168
Glava 8. Ulazno izlazni (U/I) sistem	171
8.1. Pojam toka	171
8.1.1. Binarni i tekstualni tokovi	172
8.1.2. Predefinisani tokovi	172
8.1.3. Klase tokova	173

8.2.	Formatiranje ulaza i izlaza	174
8.2.1.	Formatiranje pomoću funkcija članica klase ios	174
8.3.	Korišćenje U/I manipulatora	177
8.4.	Rad sa datotekama	180
8.4.1.	Čitanje i upis u tekstualne datoteke	182
	Pitanja	184
	Dodatak A: ASCII kôdovi znakova	187
	Dodatak B: Preprocesorske direktive	189
	Direktiva #define	189
	Makroi funkcija	190
	Direktiva #error	190
	Direktiva #include	190
	Direktive za uslovno prevođenje	191
	Dodatak C: Prevođenje programa sa više datoteka	193
	Dodatak D: Poređenje jezika C++ i Java	195
	Dodatak E: Okruženje Microsoft Visual C++ 2010 Express	201
	Odgovori na pitanja	205
	Bibliografija	207
	Indeks	209

Predgovor

Sadržaj i obim ove knjige prilagođeni su nastavnom programu istoimenog predmeta na Fakultetu za informatiku i računarstvo Univerziteta Singidunum u Beogradu. Podrazumeva se da čitaocima knjige ovo nije prvi susret sa programiranjem, tj. očekuje se da su ranije koristili neki programski jezik i da poznaju proceduralno programiranje. Naglasak je na objektno orijentisanom pristupu projektovanju. Ipak, radi kompletnosti, u knjizi su prikazani svi elementi jezika C++ kroz koji se ilustruju objektno orijentisani principi.

Uložen je veliki napor da se pronađu zanimljivi i poučni primeri za obrađene teme. Primeri su birani tako da budu što jednostavniji, ali da istovremeno ilustruju stvarne probleme sa kojima se svakodnevno susrećemo u programiranju. Stil pisanja bio je takav da se čitaocima predstavi što više praktičnog znanja o objektno orijentisanim principima primenjenim u jeziku C++. Kada savlada gradivo iz knjige, čitalac sasvim sigurno neće znati sve detalje jednog tako složenog jezika kao što je C++, ali će mu razumevanje objektno orijentisanog pristupa biti dovoljno da lako nastavi samostalno učenje. Iako je knjiga pisana tako da posluži kao udžbenik, ambicija autora jeste da ona posluži svima koji se prvi put susreću sa jezikom C++, bilo da žele da nauče principe OO projektovanja, ili žele da savladaju sam jezik C++. Na kraju svakog poglavlja data su pitanja kroz koje se može utvrditi naučeno gradivo.

Jedno od najčešćih pitanja koje studenti postavljaju jeste da li se na predmetu OOP može naučiti programiranje u jeziku C++ za Windows. Ova knjiga objašnjava osnove jezika C++ i sigurno je da ćete, kada je savladate, moći da se pohvalite da znate jezik C++. Međutim, za pisanje Windows programa potrebno je dodatno specifično znanje, koje se nadograđuje na osnove kojima se bavi ova knjiga. Čitaoci koji su zainteresovani za tu oblast upućuju se na nekoliko odličnih knjiga u odeljku literature.

I konačno, svi komentari i preporuke mogu se uputiti na adresu `ibranovic@singidunum.ac.rs`.



Glava 1

Uvod u jezik C++

U ovoj glavi upoznaćemo se sa jezikom C++, pregledati njegov istorijat, ideje primenjene tokom projektovanja jezika i najvažnije objektno orijentisane osobine. U procesu savladavanja novog programskog jezika najteže je to što su njegovi elementi međusobno povezani i ne mogu da se razmatraju nezavisno. Da bismo prevazišli taj problem, u ovom odeljku pomenućemo nekoliko najvažnijih svojstava jezika C++, bez upuštanja u detalje, ali dovoljno da sagledamo opšte principe zajedničke za sve C++ programe.

1.1. Istorijat jezika C++

C++ je najvažniji programski jezik kada je reč o razvoju softvera visokih performansi. Sintaksa ovog jezika postala je standard i u drugim programskim jezicima, a ideje koje su korišćene tokom njegovog dizajniranja često se sreću u računarstvu.

Istorijat jezika C++ počinje od jezika C, iz prostog razloga što se na njemu zasniva. C++ je proširen i poboljšan jezik C sa podrškom za objektno orijentisano programiranje (OOP, o kome će biti detaljno reči u knjizi). Često se kaže i da je jezik C++ "objektno orijentisani C". Osim podrške za OOP, u jeziku C++ su poboljšane i ugrađene biblioteke u poređenju sa jezikom C. I pored svega toga, duh i suština jezika C++ direktno su nasleđeni iz jezika C. Zbog toga, da bi se u potpunosti razumeo C++, mora se prvo razumeti C.

Pojavljivanje jezika C sedamdesetih godina prošlog veka označilo je početak modernog doba u programiranju. Dizajn ovog jezika uticao je na sve kasnije programske jezike, jer je iz osnova promenio pristup programiranju. Jezik C je osmislio i prvi implementirao Denis Riči pod operativnim sistemom UNIX, da bi ubrzo stekao brojne sledbenike širom sveta iz više razloga: bio je lak za savladavanje, omogućavao je modularno pisanje programa, sadržavao je samo naredbe koje se lako prevode u mašinski jezik, davao je brzi kôd. Od tada pa sve do pre desetak godina većina programa bila je pisana u jeziku C, ponegde dopunjena delovima u assembleru ako je brzina bila kritični faktor.

Međutim, kako su se programi razvijali, i na polju programskih jezika stanje se menjalo. Krajem sedamdesetih godina programi sa nekoliko stotina hiljada redova

kôda više nisu bili retkost, tj. počeli su da postaju toliko složeni da nisu mogli da se sagledaju u celini. Pojavila se potreba za rešenjima koja će upravljati tom složenošću. C++ i OOP nastali su kao odgovor na te probleme.

C++ je projektovao Danac Bjarne Stroustrup 1979. godine. Pre toga, on je radio na svom doktoratu na Kembridžu gde je proučavao distribuiranu obradu (paralelno programiranje za više procesora istovremeno). Pri tom je koristio jezik Simula, koji je posedovao neke važne osobine koje su ga činile pogodnim za taj zadatak; na primer, podržavao je pojam klasa, tj. struktura koje objedinjuju podatke i funkcije za rad sa njima. Dobijeni kôd je bio vrlo razumljiv, a Stroustrup je bio posebno oduševljen načinom na koji je sâm jezik usmeravao programera prilikom razmišljanja o problemu. Međutim, iako na prvi pogled idealan u teoriji, jezik Simula je posrnuo u praksi: prevođenje je bilo dugotrajno, a dobijeni kôd se izvršavao izuzetno sporo. Dobijeni program je bio neupotrebljiv, pa je Stroustrup morao ponovo da ga napiše u jeziku BCPL niskog nivoa koji je omogućio dobre performanse. Međutim, iskustvo pisanja programa u takvom jeziku bilo je užasno i Stroustrup je odlučio da više nikad neće pokušavati da rešava složene probleme neodgovarajućim alatima kao što su Simula i BCPL. Zato je 1979. započeo rad na novom jeziku koji je prvobitno nazvao "C sa klasama", ali ga je 1983. preimenovao u C++. Stroustrup nije izmislio potpuno nov jezik, već je unapredio jezik C, tj. podržao je u potpunosti sve funkcije jezika C, ali i objektno orijentisan pristup. U suštini, C++ je objektno orijentisan C, tj. C i "još malo više" (odatle ++ u imenu). Zbog toga su programeri koji su već poznavali jezik C mogli lako da ga usvoje, savladavajući samo objektno orijentisane novine.

Kada je projektovao C++, Stroustrup je zadržao pristup jezika C, uključujući njegovu efikasnost, prilagodljivost i filozofiju, a istovremeno je dodao podršku za OOP. Iako je C++ prvobitno projektovan kao pomoć u projektovanju veoma velikih programa, on nipošto nije ograničen isključivo na njih, jer se OOP atributi mogu efikasno primeniti na bilo koji programerski zadatak. Često se sreću C++ aplikacije poput editora, baza podataka, sistema za upravljanje datotekama, umrežavanje i komunikaciju. Pošto je C++ zadržao efikasnost jezika C, mnogi softverski sistemi kod kojih su kritične performanse programirani su u jeziku C++. Takođe, C++ se često sreće i u programima za operativne sisteme Windows i Linux.

1.2. Standardizacija jezika

Od svog prvog pojavljivanja, jezik C++ je pretrpeo tri velike prepravke, tokom kojih je proširivan i menjan. Prva revizija desila se 1985, a druga 1990. godine (kada je već bio u toku proces standardizacije jezika). Formiran je zajednički komitet za standardizaciju koji su činile organizacije ANSI (American National Standards Institute) i ISO (International Standards Organization). U prvom nacrtu standarda zadržane su osobine jezika koje je definisao Stroustrup, kojima su dodate i neke nove, ali je u

suštini taj nacrt standarda odražavao status jezika C++ u tom trenutku. Ubrzo pošto je kompletiran prvi nacrt standarda, Rus Aleksandar Stepanov je objavio biblioteku STL (Standard Template Library), tj. skup generičkih metoda za rad sa podacima. Komisija za standardizaciju je zatim odlučila da uključi tu biblioteku u specifikaciju jezika, što je bitno promenilo početnu definiciju jezika i usporilo proces standardizacije. Iako je proces standardizacije trajao mnogo duže nego što se očekivalo, a verzija koju je standardizovao ANSI/ISO C++ komitet bila daleko obimnija od početne Stroustrupove definicije jezika, ona je ipak konačno završena 1998. godine. Ta specifikacija se obično navodi kao standardni C++, potpuno je prenosiva i podržavaju je najpoznatiji C++ kompajleri (uključujući i Microsoftov Visual C++), te će o njoj biti reči u ovoj knjizi.

1.3. Objektno orijentisane osobine jezika C++

Suština jezika C++ jeste podrška za objektno orijentisano programiranje (OOP), koje je i bilo razlog za njegov nastanak. Zbog toga je pre nego što se napiše i najjednostavniji C++ program važno razumeti OOP principe. Objektno orijentisano programiranje preuzelo je najbolje ideje strukturnog programiranja koje su proširene sa nekoliko novina. Kao rezultat toga, poboljšan je način organizovanja programa. U najopštijem smislu, program može da bude organizovan na dva načina: sa naglaskom na kôdu (tj. na tome šta se dešava) ili sa naglaskom na podatke (sa kojima se radi). Korišćenjem isključivo tehnika strukturnog programiranja, programi su obično organizovani oko kôda koji operiše sa podacima. Objektno orijentisani programi rade na drugi način. Oni su organizovani oko podataka, pri čemu je ključni princip da podaci kontrolišu pristup kôdu. U objektno orijentisanom jeziku, tip podataka precizno definiše šta sa tim podacima može da se radi.

Tri su važne osobine programskog jezika koje ga čine objektno orijentisanim:

1. skrivanje podataka (encapsulation)
2. nasleđivanje (inheritance)
3. polimorfizam (polymorphism)

Skrivanje podataka

Skrivanje (kapsuliranje) podataka je programerski mehanizam koji povezuje kôd i podatke sa kojima radi, na taj način da i kôd i podaci budu sigurni od spoljašnjih uticaja. U objektno orijentisanom jeziku, kôd i podaci mogu da se povežu tako da ne budu vidljivi spolja, tako što se smeštaju u objekat. Unutar objekta, kôd i podaci mogu da budu privatni ili javni. Privatnom kôdu ili podacima ne mogu da pristupaju delovi programa koji se nalaze izvan objekta, a javni delovi objekta koriste se za obezbeđivanje kontrolisanog interfejsa ka privatnim elementima objekta. Osnovna jedinica kojom se postiže skrivanje podataka u jeziku C++ je klasa. Klasa definiše iz-

gled objekta i određuje podatke i kôd koji će raditi sa podacima. Objekti su primerci (instance) klase. Kôd i podaci koji čine klasu zovu se članovi klase. Podaci članovi (*member data*) klase su podaci koji su definisani u klasi, a funkcije članice (*member functions*) čini kôd koji radi sa tim podacima.

Polimorfizam

Polimorfizam (na grčkom “više oblika”) je osobina koja omogućuje projektovanje zajedničkog interfejsa koje će se zatim preslikavati u različite akcije. Pojam polimorfizma se obično izražava frazom “jedan interfejs, više metoda”, a najlakše se razume na primeru. Korisnik koji kupuje auto sigurno neće biti zadovoljan ako se njegov novi model razlikuje od starog po načinu korišćenja, na primer, da se umesto pritiskom na papučicu gasa auto ubrzava povlačenjem ručne kočnice. Vozač auta s pravom očekuje da se auto ubrzava kada pritisne papučicu za gas; ono što eventualno može da bude promenjeno u novom modelu auta može da bude vreme ubrzavanja do 100 km/h. Slično je i sa programskim komponentama: programer ne treba da se opterećuje time koju verziju komponente koristi, već će jednostavno tražiti od komponente određenu uslugu, a na njoj je da mu je pruži na odgovarajući način. To je polimorfizam.

Polimorfizam uprošćava programe jer omogućuje definisanje generalnih operacija; kompajleru je ostavljen zadatak da izabere određenu akciju (tj. metod) koji se primenjuje na konkretnu situaciju. Programer treba samo da pozna i koristi uopšteni interfejs.

Nasleđivanje

Nasleđivanje je proces kojim se jednom objektu omogućuje da preuzme svojstva drugog objekta. To je važno zato što se time podržava pojam hijerarhijske klasifikacije. Na primer, sorta jabuke crveni delišes je deo klasifikacije jabuka, koje su zauzvrat deo klase voća, a voće opet potpada pod širu klasu hrane. Klasa hrane konkretno poseduje određene kvalitete (ukus, hranljiva vrednost i sl.) koji se zatim primenjuju i na njegovu potklasu, tj. voće. Osim pomenutih karakteristika hrane, klasa voće ima određene karakteristike (sočnost, slast i sl.) koje ga razlikuju od druge hrane. Klasa jabuka definiše svojstva karakteristična za jabuke (raste na drveću, nije tropsko voće i sl.). Crveni delišes nasleđuje svojstva svih prethodno pomenutih klasa, a definiše samo ona koja ga čine jedinstvenim (npr. mora biti crvene boje). Bez korišćenja hijerarhija, svaki objekat morao bi eksplicitno da definiše sve svoje karakteristike. Kada se koristi nasleđivanje, objekat treba da definiše samo svojstva koja ga čine jedinstvenim unutar klase, a svoje generalne osobine nasleđuje od roditeljskih klasa. Na taj način mehanizam nasleđivanja omogućuje jednom objektu da bude specifični primerak nekog opštijeg slučaja.

1.4. Unošenje, prevođenje i izvršavanje C++ programa

Da bi se izvršio C++ program, potrebna su četiri koraka:

1. pisanje izvornog kôda
2. kompajliranje programa
3. povezivanje programa
4. izvršavanje programa

Procesor računara je u stanju da razume i direktno izvršava samo binarni kôd (tj. kôd koji se sastoji samo od nula i jedinica). Nažalost, takav kôd čovek veoma teško razume. Zbog toga se nakon pisanja izvornog kôda programa (koji je razumljiv ljudima) on prevodi u mašinski jezik, tj. u binarni ili objektni kôd (koji razume računar) pomoću alatke koja se zove prevodilac ili kompajler (compiler).

Pre nego što nastavimo, naglasimo još jednom razliku između izvornog kôda i objektnog kôda. Izvorni kôd (source code) je čitljiv tekst programa koji se čuva u tekstualnoj datoteci. Objektni kôd (object code) je izvršni oblik programa koji kreira prevodilac (kompajler).

Unošenje kôda programa u tekstualnu datoteku obavlja se pomoću nekog tekst editora (npr. Notepad, WordPad, vi i sl.), a nikako u programu za obradu teksta kao što je Microsoftov Word (zato što napredni programi za obradu teksta osim "čistog" teksta u svojoj datoteci čuvaju i neke druge podatke koji zbunjuju C++ kompajlere). Ime datoteke koja sadrži izvorni C++ kôd može da bude proizvoljno, ali se obično koristi nastavak .cpp (ili ređe .cp). Nastavak je potreban samo da bi se datoteka sa izvornim kôdom kasnije lakše pronašla.

Način kompajliranja (prevođenja) programa zavisi od okruženja za programiranje koje se koristi. Šta više, mnoga okruženja za programiranje nude više načina za kompajliranje datoteke. Zbog toga nije moguće dati opšte uputstvo za kompajliranje, već treba proučiti okruženje koje se koristi. Osim prevođenja izvornog kôda u objektni kôd, kompajler ima i zadatak da proveri da li je izvorni kôd sintaksno ispravan, odnosno da li su poštovana pravila programskog jezika. Greške koje pronalazi kompajler zovu se greške tokom prevođenja (compile-time errors). U svakom slučaju, kompajler kao izlaz daje objektni kôd koji ima nastavak .o ili .obj. Za svaki programski jezik postoji poseban kompajler; takođe, izvorni kôd koji je kompajliran za jedan tip procesora po pravilu neće raditi na računaru sa drugim tipom procesora, ili sa drugačijim operativnim sistemom. Kaže se da kôd nije prenosiv (kompatibilan).

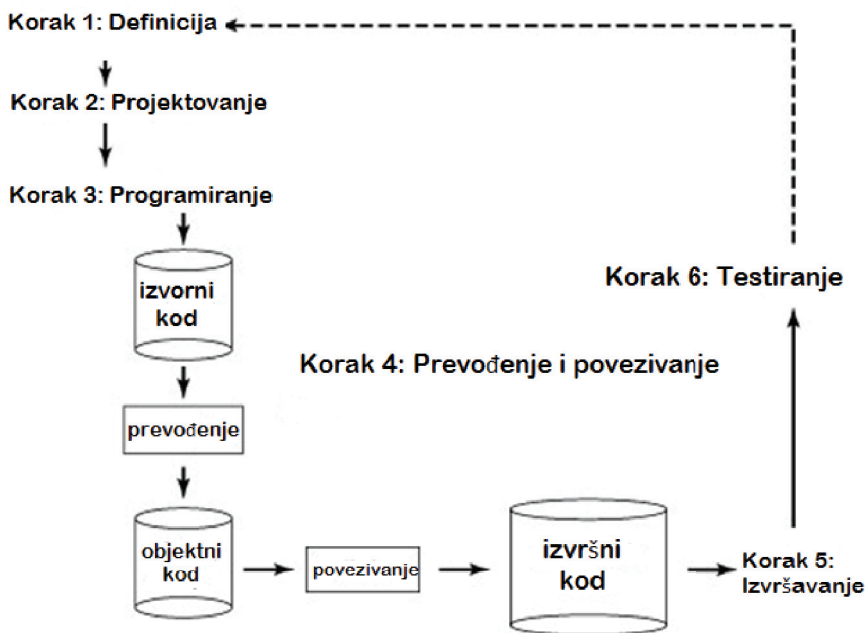
Često se u pisanju programa koriste tzv. biblioteke kôda (libraries). To su zbirke standardnog, profesionalnog kôda koji podržava i(li) proširuje programski jezik. Neke biblioteke su i ugrađene u sâm jezik (npr. biblioteka matematičkih funkcija, metode za rad sa stringovima, algoritmi za sortiranje itd), a može ih kupiti ili razvijati i sâm programer. Korišćenje biblioteka gotovog, proverenog kôda uvek se preporučuje umesto pisanja sopstvenog kôda za istu namenu iz više razloga: taj kôd je proveren,

profesionalan, efikasniji od kôda koji biste sami napisali, a uostalom, nema ni svrhe gubiti vreme na “izmišljanje tople vode”.

Zbog korišćenja biblioteka, sledeći korak u procesu nastanka programa jeste tzv. povezivanje (linking). Linker je poseban alat koji povezuje različite module koje je kompajler generisao od izvornog kôda, dodaje već preveden kôd iz biblioteka (ako su one korišćene u programu) i sve to povezuje u celinu koja se naziva izvršni kôd (executable code). Pri tom, slično kao i kompajler, i linker prijavljuje eventualne greške (npr. navođenje kôda iz nepostojeće biblioteke); takve greške zovu se greške tokom povezivanja (link-time errors).

Nakon kompajliranja i povezivanja, dobija se izvršna datoteka (executable file) koja po pravilu ima isto ime kao datoteka sa izvornim kôdom, a pod operativnim sistemom Windows nastavak joj je .exe (Slika 1.1).

Kada se program kompajlira, on može da se izvrši, najlakše pozivom iz komandne linije. Ipak, to što je program uspešno preveden i povezan ne znači da u njemu nema grešaka. Na primer, dešava se da pravilno radi sa jednim podacima, a nepravilno sa drugim. Greške otkrivene tokom testiranja programa zovu se greške tokom izvršavanja (run-time errors). Postupak testiranja programa, pronalaženja i uklanjanja grešaka nastavlja se sve dok se ne proceni da je program dovoljno stabilan. Pronalaženje grešaka zove se debugovanje (debugging), a poseban alat koji služi za tu namenu jeste debager (debugger).



Slika 1.1: Ciklus razvoja aplikacije.

1.4.1. Integrisana razvojna okruženja za programiranje (IDE)

Postoje brojna integrisana razvojna okruženja za jezik C++ (Microsoft Visual Studio, Borland Enterprise Studio, KDevelop, CodeForge itd.); koje ćete od njih koristiti najviše zavisi od toga za koji ćete se operativni sistem odlučiti (Windows, Linux, MacOS) i od ličnih sklonosti. U svako razvojno okruženje po pravilu su integrisane sve potrebne alate za razvoj programa (od tekst editora, preko kompajlera, linkera, biblioteke, sve do debagera). Prevođenje i povezivanje programa obično su objedinjeni u istu komandu, tj. povezivanje se obavlja automatski. Izvršavanje programa obično se postiže komandom Run iz menija, ili komandom Debug ukoliko se program izvršava u režimu traženja grešaka. U Dodatku E detaljno je opisano razvojno okruženje Microsoft Visual C++ Express Edition koje može besplatno da se preuzme sa Interneta, a korišćeno je za proveru programa u ovoj knjizi.

1.4.2. Prevođenje i interpretiranje

Postoje programski jezici, kao što su Java ili C#, čiji se izvorni kôd ne prevodi direktno u izvršni kôd, već se interpretira. To zapravo znači da prevođenje i izvršavanje programa nisu vremenski razdvojeni, već objedinjeni procesi. Umesto da se prevede u mašinski kôd koji će se izvršavati u procesoru, Javin izvorni kôd se prevodi u tzv. bajt kod (bytecode) koji interpretira i izvodi Javina virtuelna mašina. C# kôd se izvršava u okruženju Common Language Runtime (CLR). Prednost ovakvog pristupa je to što se jednom prevedeni bajt kôd može izvršavati na bilo kom računaru i operativnom sistemu na kome je instalirana virtuelna mašina. S druge strane, takav kôd neće biti optimizovan za procesor na kome se izvršava, pa su zato interpretirani programi mnogo sporiji od aplikacija koje se direktno kompajliraju u izvorni kôd. C++ programi su poznati po svojoj efikasnosti, koja je direktna posledica toga što se kompajliraju u mašinski kôd. Ostale razlike između jezika C++ i Java detaljno su opisane u Dodatku D.

Često se raspravlja o tome koji je jezik "bolji": C++, Java ili C#, pa ćemo se ovde ukratko osvrnuti i na tu problematiku. Sintaksa sva tri jezika je gotovo istovetna, a osnovna razlika je tip okruženja za koje su namenjeni. Jezik C++ je izmišljen da bi se pisali veoma efikasni programi za određeni tip procesora i operativnog sistema. Ako želite da izvršite isti C++ program na drugom sistemu, moraćete ponovo da ga kompajlirate i specijalno optimizujete za tu platformu. Java je razvijena posebno za potrebe Internet programiranja, a C# za programiranje modularnih softverskih komponentata.

Dakle, odgovor na pitanje koji je jezik najbolji zavisi od toga kakav se problem rešava: ako je neophodna efikasnost, korišćićemo C++; za Web programiranje odlučićemo se za Javu, a za specifične Windows programe kod kojih efikasnost nije najvažnija upotrebićemo C#.

1.4.3. Konzolni i grafički programi

Gotovo svi primeri programa u ovoj knjizi biće konzolni programi. Konzolni programi izvršavaju se u komandnoj liniji, tj. ne koriste grafički korisnički interfejs (Graphical User Interface, GUI). Pod grafičkim interfejsom podrazumevamo izgled programa na koji smo navikli, sa dijalozima, menijima sa opcijama, dugmadima za potvrdu i sl. Razlog za izbegavanje grafičkih programa je lako razumeti: čak i najjednostavniji grafički program za Windows ima 50 do 70 redova kôda i nije pogodan za objašnjavanje osnovnih pojmova. Drugi razlog za izbegavanje grafičkih programa jeste njihova nekompatibilnost sa standardnim jezikom C++. Grafička okruženja u jeziku C++ za različite operativne sisteme po pravilu koriste verzije jezika C++ koje odstupaju od one propisane standardom (kojom se mi bavimo u ovoj knjizi), ili zahtevaju savladavanje posebnih grafičkih biblioteka, što prevazilazi okvire ovog kursa. Takođe, programiranje grafičkih programa u jeziku C++ veoma zavisi od toga za koji se operativni sistem aplikacija pravi; grafički C++ program napisan za Windows po pravilu neće moći da radi pod Linuxom i obrnuto.

1.5. Prvi C++ program

Došli smo do trenutka kada ćemo napisati prvi, najjednostavniji program u jeziku C++ i objasniti najvažnije elemente njegove sintakse.

```
/* ovo je najprostiji C++ program
nazovite ovu datoteku primer.cpp */

//tipican C++ komentar koji se proteže
//samo do kraja reda

#include<iostream>
using namespace std;

int main() {
    cout << "OOP se uči na" <<
        " II godini Fakulteta za Informatiku i Racunarstvo " <<
        " Univerziteta Singidunum." << endl;
    return 0;
};
```

Rezultat izvršavanja:

OOP se uči na II godini Fakulteta za Informatiku i Racunarstvo Univerziteta Singidunum.

U jeziku C++ postoje dve vrste komentara. Tip komentara koji počinje sa `/*` nasleđen je iz jezika C, može da se proteže u više redova, a završava se tek kada naiđe znak `*/`. Sve što se nalazi između simbola `/*` i `*/` prevodilac potpuno zanemaruje. Komentar koji počinje znakom `//` je tipičan za C++, a završava se na kraju istog reda. Obično se C komentari koriste za pisanje detaljnijih objašnjenja, a komentar `//` za kratke opaske.

Iako pisanje komentara zahteva dodatno vreme i napor, ono se uvek isplati. Ako se desi da neko drugi mora da prepravlja vaš kôd, ili (još gore) da vi nakon izvesnog vremena morate da ispravljate sopstveni kôd, komentari će vam olakšati da proniknete u to kako je autor kôda razmišljao. Ipak, sa komentarima ne treba preterivati; višak komentara podjednako je loš kao i njihovo potpuno odsustvo, a prava mera dolazi sa iskustvom i proučavanjem kôda iskusnih programera.

Sve praznine u kôdu prevodilac takođe zanemaruje; one se stavljaju isključivo radi povećanja čitljivosti programa. Pod prazninama se osim razmaka (whitespace) podrazumevaju i praznine dobijene pomoću tastera Tab (tabulatori) i prelasci u novi red. Naredbe se mogu protezati u više redova, a svaka naredba u jeziku C++ se mora završavati znakom tačka-zarez (`;`).

U sedmom redu nalazi se naredba `#include<iostream>` kojom se od kompajlera zahteva da u program uključi biblioteku `iostream`. U toj biblioteci (koja je ugrađena u jezik C++) nalazi se *izlazni tok* (engl. *output stream*), odnosno funkcije za ispis teksta na ekranu. Ovde naglašavamo da `#include` nije naredba jezika C++, već tzv. preprocesorska direktiva nasleđena iz jezika C. Kada naiđe na preprocesorsku direktivu, kompajler će prekinuti postupak prevođenja kôda u tekućoj datoteci, skočiti u datoteku `iostream`, prevesti je, a zatim se vratiti u polaznu datoteku iza reda `#include`. Sve preprocesorske direktive počinju znakom `#`. Pošto preprocesorske direktive nisu C++ naredbe, one ne zahtevaju tačku-zarez na kraju; mnogi kompajleri to čak prijavljuju kao grešku.

`iostream` je primer datoteke zaglavlja (engl. *header file*). U zaglavljlama se nalaze deklaracije funkcija sadržanih u odgovarajućim datotekama. Jedna od osnovnih karakteristika jezika C++ je mali broj funkcija ugrađenih u sâm jezik. Ta oskudnost olakšava učenje samog jezika, a istovremeno pojednostavljuje i ubrzava prevođenje. Za specifične zahteve na raspolaganju je veliki broj gotovih biblioteka klasa.

U petom redu nalazi se naredba `using namespace std`. `using` i `namespace` su ključne (zaštićene) reči jezika C++ kojim se "aktivira" određena oblast imena (imenik, engl. *namespace*), a `std` je naziv imenika koji sadrži sve standardne funkcije, uključujući i funkcije iz već pomenute biblioteke `iostream`. Imenici su se prilično kasno pojavili u jeziku C++, a uvedeni su da bi se izbegao sukob sa nazivima funkcija i promenljivih iz različitih biblioteka. Na primer, ako dve različite funkcije iz različitih biblioteka imaju isto ime, a koriste se u istoj datoteci, kompajler će javiti grešku. Kada ne bismo imali na raspolaganju imenike, jedino rešenje u takvom slučaju bilo bi da

se jednoj funkciji u nekoj od biblioteka promeni ime, ali to ponekad nije moguće jer autori obično te biblioteke objavljuju već prevedene. O imenicima će detaljnije biti reči u Poglavlju 5, a do tada ćemo ih uzeti “zdravo za gotovo”.

Izvršavanje programa počinje od reda `int main()`. Svi C++ programi sastavljeni su od jedne funkcije ili više njih. Svaka C++ funkcija mora da ima ime, a svaki C++ program mora da ima jednu (i samo jednu) funkciju `main()`. U funkciji `main()` počinje (i najčešće se završava) izvršavanje programa. Otvorena vitičasta zagrada iza `main()` označava početak kôda funkcije. Reč `int` ispred `main()` označava tip podatka koji vraća funkcija `main()`. Kao što ćemo saznati u nastavku, C++ ugrađeno je nekoliko tipova podataka, a ceo broj (`int`, od `integer`) je jedan od njih.

Treća naredba unutar programa `main()` je `cout`, što je skraćenica za *console output*. To je ime izlaznog toka definisanog u zaglavlju `iostream`, pridruženom ekranu računara. Ono što sledi iza operatora `<<` upućuje se na standardni izlaz, tj. na ekran računara (u našem slučaju, to je poruka *OOP se uči na II godini Fakulteta za Informatiku i računarstvo Univerziteta Singidunum*). Poruka “OOP se uči na II godini Fakulteta za Informatiku i računarstvo Univerziteta Singidunum” je string, odnosno niz znakova između navodnika. Identifikator `endl` je objekat toka koji označava kraj reda (*end line*), odnosno prelazak u ispis u novom redu.

Korisno je znati da je `iostream` novi (standardom propisani) naziv zaglavlja `iostream.h`. Do te promene naziva došlo je s razlogom. Kada su uvedeni imenici, trebalo je sve standardne datoteke uvesti u imenik `std`. Pošto je tada već postojalo mnogo kôda koji je koristio biblioteku `iostream.h`, ona nije smela da se menja zbog održavanja kompatibilnosti sa starim kôdom. Rešenje je bilo da se definiše nova biblioteka sličnog imena u kojoj su definisane sve funkcije kao u zaglavlju `iostream.h`, ali unutar imenika `std`. Zato je u bibliotekama većine modernih C++ kompajlera unutar zaglavlja `iostream` zapravo samo uključena (preprocesorskom direktivom `#include`) datoteka `iostream.h`, a slično je i sa svim ostalim starim zaglavljima.

Naredba `return 0;` završava program i vraća vrednost 0 procesu koji je pozvao program (a to je obično operativni sistem). Vrednost 0 po pravilu znači da se program pravilno izvršio, dok druge vrednosti znače da se program nepravilno završio zbog neke greške. Zatvorena vitičasta zagrada na kraju programa u redu 15 i formalno ga završava.

Pitanja

1. Jezik C++ je bliski rođak jezika C# i Java.
 - a) tačno
 - b) pogrešno

2. Autor jezika C++ je:

- a) Bjarne Stroustrup
- b) Denis Riči
- c) Aleksandar Stepanov

3. Jezik C++ je nastao od jezika C.

- a) tačno
- b) pogrešno

4. Kompajliranjem (prevođenjem) programa dobija se:

- a) izvorni kôd
- b) objektni kôd
- c) izvršni kôd

5. Alatka koja služi za povezivanje objektnog kôda sa bibliotekama zove se:

- a) kompajler
- b) linker
- c) debager

6. C++ program počinje izvršavanje:

- a) u prvom redu kôda
- b) u datoteci main.cpp
- c) u funkciji main

7. Svaki C++ program mora da ima:

- a) naredbu cout
- b) funkciju main
- c) naredbu #include
- d) sve pomenuto

8. Preprocesorske direktive počinju sa:

- a) #
- b) !
- c) <
- d) ništa od pomenutog

9. Svaka kompletna naredba u jeziku C++ završava se:

- a) tačkom-zarezom
- b) zarezom
- c) dvotačkom
- d) zatvorenom zagradom

10. Koja od sledećih naredbi je ispravna?

- a) `#include (iostream.h)`
- b) `#include <iostream>`
- c) `#include {iostream.h}`
- d) sve navedene

11. Komentar `/* Ovaj program napisao je A. Programer */` je u:

- a) C stilu
- b) C++ stilu

12. Kada se izvršavaju preprocesorske direktive?

- a) pre nego što prevodilac kompajlira program
- b) nakon što prevodilac kompajlira program
- c) istovremeno dok prevodilac kompajlira program

13. Programski jezik C++ je:

- a) objektno orijentisano proširenje programskog jezika C
- b) proceduralno proširenje programskog jezika C

14. Programski jezik C++ je standardizovan.

- a) tačno
- b) pogrešno

15. Sve C++ naredbe moraju da se završe u istom redu u kome su započete.

- a) tačno
- b) pogrešno

16. Program napisan u programskom jeziku C++:

- a) mora se prevesti na mašinski jezik
- b) ne mora se prevesti na mašinski jezik jer je objektno orijentisan
- c) ne prevodi se, već se interpretira

17. Tačan redosled u pisanju programa je:

- a) `program.obj -> program.cpp -> program.exe`
- b) `program.exe -> program.obj -> program.cpp`
- c) `program.exe -> program.obj -> program.exe`
- d) `program.cpp -> program.obj -> program.exe`

18. Objektni kôd je rezultat rada:

- a) editora
- b) kompajlera
- b) linkera

19. Naziv imenika (namespace) koji sadrži sve nazive standardnih C++ funkcija je `cout`.

- a) tačno
- b) pogrešno





Glava 2

Tipovi podataka i operatori

Jezgro svakog programskog jezika čine tipovi podataka i operatori koji definišu njegova ograničenja. C++ podržava brojne tipove podataka i operatore, što ga čini pogodnim za različite programerske probleme. Iako su operatori dosta široka tema, počecemo pregledom osnovnih tipova podataka jezika C++ i najčešće korišćenih operatora. Bliže ćemo se upoznati i sa promenljivama i izrazima.

2.1. Identifikatori

Mnogim elementima C++ programa (promenljivama, klasama) potrebno je dati određeno ime, tzv. *identifikator*. Imena koja dodeljujemo identifikatorima su proizvoljna, uz uslov da poštujemo sledeća tri pravila:

1. Identifikator može da bude sastavljen od slova engleske abecede (A - Z, a - z), brojeva (0 - 9) i donje crte (_)
2. Prvi znak mora da bude slovo ili donja crta
3. Identifikator ne sme da bude neka ključna reč (Tabela 2.1) niti neka alternativna oznaka operatora (Tabela 2.2). To ne znači da ključna reč ne može da bude deo identifikatora (na primer, `moj_int` je dozvoljeni identifikator iako je `int` ključna reč). Iako `main` nije ključna reč, treba se ponašati kao da jeste. Treba izbegavati dvostruku donju crtu u identifikatorima, ili započinjanje identifikatora donjom crtom iza koje sledi veliko slovo, jer su takvi identifikatori rezervisani za C++ implementacije i standardne biblioteke (npr. `__LINE__`, `__FILE__`)

Tabela 2.1. Ključne reči jezika C++.

<code>asm</code>	<code>false</code>	<code>sizeof</code>
<code>auto</code>	<code>float</code>	<code>static</code>
<code>bool</code>	<code>for</code>	<code>static_cast</code>
<code>break</code>	<code>friend</code>	<code>struct</code>
<code>case</code>	<code>goto</code>	<code>switch</code>
<code>catch</code>	<code>if</code>	<code>template</code>
<code>char</code>	<code>inline</code>	<code>this</code>
<code>class</code>	<code>int</code>	<code>throw</code>

Tabela 2.1. Ključne reči jezika C++.

const	long	true
const_cast	mutable	try
continue	namespace	typedef
default	new	typeid
delete	operator	typename
do	private	union
double	protected	unsigned
dynamic_cast	public	using
else	register	virtual
enum	reinterpret_cast	void
explicit	return	volatile
export	short	wchar_t
extern	signed	while

Tabela 2.2. Alternativne oznake operatora

and	bitand	compl	not_eq	or_eq	xor_eq
and_eq	bit_or	not	or	xor	

Vodeći računa o pomenutim ograničenjima, možemo pustiti mašti na volju i svoje promenljive i funkcije zvati kako hoćemo. Jasno je da je zbog razumljivosti kôda bolje davati razumljiva imena promenljivama, odnosno dodeljivati im imena koja odražavaju njihovu stvarnu funkciju, npr.

kamata

Racun

rezultat

a izbegavati identifikatore poput

Snezana_I_Sedam_Patuljaka

FrankieGoesToHollywood

moja_privatna_promenljiva

Primetimo da C++ ne dozvoljava korišćenje naših dijakritičkih znakova (č, ć, ž, š, đ) u identifikatorima. Čak i ako bismo koristili neki kompajler koji ih podržava, treba ih izbegavati zbog prenosivosti kôda. Takođe, veoma je važno znati da C++ razlikuje mala i velika slova (tj. jezik je *case-sensitive*). Zbog toga identifikatori

tripraseta
triPraseta
TRIPRASETA

predstavljaju tri različita imena. Ne postoji ograničenje dužine identifikatora, mada neki kompajleri razlikuju samo nekoliko prvih znakova (pri čemu je pojam “nekoliko” prilično neodređen). Jasno je da dugačka imena treba izbegavati i zbog sopstvene udobnosti, naročito kada se radi o funkcijama i promenljivama koje se često koriste. U Tabeli 2.3 prikazano je nekoliko primera dobrih i loših identifikatora.

Tabela 2.3. Primeri identifikatora.	
Dobri	Nedozvoljeni
kamata	7Up
velikiCeoBroj	Ana Marija
poluprecnik	Ana-Marija
ana_marija	_system

2.2. Specijalni znakovi

U jeziku C++ postoji nekoliko specijalnih znakova (*escape sequence*) koji se koriste kada je potrebno na ekranu odštampati neki znak koji ima specijalnu namenu u jeziku, npr. #. Svi specijalni znakovi započinju obrnutom kosom crtom (\, backslash) koju nikako ne treba mešati sa kosom crtom (/). U Tabeli 2.4 prikazani su specijalni znakovi i njihova namena.

Tabela 2.4: Neki specijalni znaci		
Specijalni znak	Naziv	Opis
\n	newline	Pomera kursor u sledeći red
\t	tab	Pomera kursor na sledeći tabulator
\a	alarm	Računar se oglašava
\b	backspace	Vraća kursor za jednu poziciju unazad
\r	carriage return	Pomera kursor na početak tekućeg reda
\\	backslash	Prikazuje obrnutu kosu crtu
\'	single quote	Prikazuje apostrof
\"	double quote	Prikazuje navodnik
\?	question mark	Prikazuje znak pitanja

2.3. Ugrađeni tipovi podataka

Bez obzira na jezik u kome je pisan, svaki program sastoji se od niza naredbi koje menjaju vrednosti objekata koji se nalaze u memoriji računara. Da programer tokom pisanja programa ne bi pamtio memorijske adrese, svi programski jezici omogućuju da se vrednosti objekata dohvataju preko simboličkih adresa. Tip objekta, između ostalog, određuje i raspored bitova prema kojem je objekat sačuvan u memoriji.

C++ nudi ugrađene tipove podataka koji odgovaraju celim brojevima, realnim brojevima u pokretnom zarezu (floating-point) i logičkim vrednostima. To su tipovi vrednosti sa kojima se najčešće radi u programima. Kao što ćemo kasnije videti, C++ omogućuje i konstruisanje mnogo složenijih tipova, kao što su nabranja, strukture i klase, ali se i oni mogu razložiti na osnovne (proste) tipove.

Još jedan razlog zbog koga su tipovi podataka toliko važni za C++ programiranje jeste to što su osnovni tipovi čvrsto povezani sa osnovnim blokovima sa kojima računar radi: bajtovima i rečima. Jezik C++ omogućuje rad sa istim tipovima podataka sa kojima radi i procesor, što omogućuje pisanje veoma efikasnog, sistemskog kôda. C++ podržava sedam osnovnih tipova podataka prikazanih u sledećoj tabeli:

Tabela 2.5: Sedam osnovnih tipova jezika C++	
Tip	Značenje
char	znak
w_char	“široki” znak
int	ceo broj
float	decimalni broj
double	decimalni broj u dvostrukoj preciznosti
bool	logička vrednost
void	bez vrednosti

C++ dozvoljava proširivanje osnovnih tipova navođenjem modifikatora ispred naziva tipa podataka. Modifikator menja značenje osnovnog tipa tako da bude pogodniji za specifične načine korišćenja. Postoji četiri modifikatora tipa: signed, unsigned, long i short. Svi osnovni tipovi su standardno označeni (signed). Tabela 2.6 prikazuje sve dozvoljene kombinacije osnovnih tipova, kao i garantovan minimalni opseg za svaki tip prema ANSI/ISO C++ standardu.

Stvarni opseg tipa u jeziku C++ zavisi od konkretne implementacije. Pošto C++ standard zadaje samo minimalni opseg koji tip podataka mora da podrži, u dokumentaciji kompajlera uvek treba proveriti stvarne opsege.

Tabela 2.6: Opseg osnovnih C++ tipova prema ANSI/ISO standardu

Tip	Veličina u bajtovima	Opseg definisan ANSI/ISO standardom
bool	1	true ili false
char	1	-128 do 127
unsigned char	1	0 do 255
signed char	1	-128 do 127
int	4	-2 147 483 648 do 2 147 483 647
unsigned int	4	0 do 4 294 967 295
signed int	4	isto kao int
short int	2	-32 768 do 32 767
unsigned short int	2	0 do 65 535
signed short int	2	isto kao short int
long int	4	-2 147 483 648 do 2 147 483 647
signed long int	4	isto kao long int
unsigned long int	4	0 do 4 294 967 295
float	4	$\pm 3,4 \times 10^{\pm 38}$ sa tačnošću od približno 7 cifara
double	8	$\pm 1,7 \times 10^{\pm 308}$ sa tačnošću od približno 15 cifara
long double	8	isto kao double

Kada se u C++ programu uvodi neka promenljiva, mora se deklarirati kog je ona tipa. Pošto ne može da postoji promenljiva za koju se ne zna kog je tipa, kaže se da je C++ strogo tipiziran jezik. Za razliku od programskog jezika C u kome sve deklaracije moraju da budu na početku programa ili funkcije, u jeziku C++ ne postoji takvo ograničenje, pa se deklaracije promenljivih mogu nalaziti bilo gde u programu, ali pre prvog korišćenja promenljive. Promenljiva se *deklariše* navođenjem tipa i imena:

```
int tezina; //deklaracija celobrojne promenljive
unsigned short int dani; //deklaracija pozitivne celobrojne promenljive
```

Tip promenljive određuje koliko bajtova u memoriji će za nju biti odvojeno. Ukoliko se prilikom deklaracije promenljivoj dodeli i početna vrednost pomoću operatora dodele (=), kaže se da je promenljiva i *inicijalizovana*:

```
int tezina = 70; //deklaracija i inicijalizacija celobrojne promenljive
```

Dakle, pod *deklarisanjem* promenljive podrazumeva se definisanje imena i tipa promenljive, dok se pod *definisanjem* promenljive podrazumeva da se za nju odvajaju prostor u memoriji. Osim nekih specijalnih slučajeva o kojima će biti reči u nastavku

knjige, svaka deklaracija u jeziku C++ istovremeno je i definicija, odnosno čim se promenljiva deklariše, za nju se automatski rezerviše prostor u memoriji.

2.3.1. Promenljive i konstante

Promenljiva se tako zove zato što njena vrednost može da se menja. *Konstante* (ili literali) su podaci čija se vrednost tokom izvršavanja programa ne može menjati, odnosno vrednosti koje predstavljaju “same sebe”. U Tabeli 2.7 prikazano je nekoliko vrednosti konstanti. U nastavku ćemo se detaljnije pozabaviti osobinama svih osnovnih tipova podataka ugrađenih u jezik, gde ćemo objasniti i označavanje konstanti za sve tipove pojedinačno.

Tabela 2.7: Primeri konstanti (literala)	
Tip	Primeri literala
char	'A', 'Z', '8', '*'
int	-77, 65, 0x9FE
unsigned int	10U, 64000U
long	-77L, 65L, 12345L
unsigned long	5UL
float	3.14f
double	1.414
bool	true, false

2.3.2. Celobrojni podaci (int)

U računarima koji koriste aritmetiku komplementa dvojke (a takvi su gotovo svi), ceo broj (tip int) imaće opseg od najmanje -32 768 do najviše 32 767. Iako nije sigurno da tip int zauzima 4 bajta, u svim slučajevima, međutim, opseg tipa short int biće podskup opsega za int, koji će sa svoje strane biti podskup opsega za tip long int. Dakle, tip int i tip long int mogu da imaju isti opseg, ali int ne može da ima veći opseg od tipa long int.

Standardno su svi celobrojni tipovi označeni (signed). Modifikator unsigned treba koristiti samo za pozitivne podatke, npr. ako promenljiva treba da sadrži godine starosti ili težinu. Pri tom treba voditi računa da kompajler ne prijavljuje grešku ako se promenljivoj koja je deklarirana kao unsigned dodeli neka negativna vrednost, već o tome mora da vodi računa programer. Prilikom deklaracije, može se i izostaviti int, kao u sledećem primeru:

```
unsigned sobe = 32;
long apartmani = 300;
```

Celobrojne konstante su standardno tipa int:

```
sobe = 300;
apartmani = 32;
```

Ako želimo celobrojnu konstantu tipa `long`, na kraj konstante treba dodati `L` ili `l`; za neoznačenu `U` ili `u`, a ovi simboli se mogu i kombinovati:

```
spratovi = 32L;  
sobe = 300UL;  
apartmani = 10u;
```

Ako tip literala nije zadat, C++ kompajler standardno celobrojnoj vrednosti dodeljuje najmanji celobrojni tip u koji ona može da stane, a to je po pravilu tip `int`. C++ podrazumeva da su celobrojne konstante predstavljene u decimalnom brojnom sistemu. Heksadecimalne konstante počinju znakom `0x`, a oktalne konstante znakom `0`:

```
int tucePatuljaka = 0x0C; //12 decimalno  
int SnezanaIPatuljci = 010; //8 decimalno
```

Sve konstante koje počinju nulom kompajler tretira kao oktalne brojeve, što znači da će nakon prevođenja `010` i `10` biti dva različita broja, pa o tome treba voditi računa.

2.3.3. Znakovni podaci (`char`)

Tip `char` služi za smeštanje znakova, ali je to u suštini celobrojni tip podataka i može se koristiti za čuvanje “malih” brojeva. Razlog zbog koga se za čuvanje znakova koristi celobrojni tip jeste to što su znakovi u računaru predstavljeni brojevima. Najčešće korišćen metod za predstavljanje znakova jeste ASCII kôd (Dodatak A). Znakovna promenljiva se deklarise i inicijalizuje na sledeći način:

```
char slovo = 'A';  
slovo = 65; //ASCII kod slova A
```

Znakovne konstante navode se pod jednostrukim navodnicima (npr. `'A'`). Kada se znak smešta u memoriju, zapravo se smešta njegov numerički kôd; kada se od računara zatraži da ispiše znak na ekranu, on prikazuje znak koji odgovara tom kôdu.

```
//program koji demonstrira blisku vezu  
//izmedju znakova i celih brojeva  
#include <iostream>  
using namespace std;  
  
int main() {  
    char slovo;  
    slovo = 65;  
    cout << slovo << endl;  
    slovo = 66;  
    cout << slovo << endl;  
    slovo = 'C';  
    cout << slovo << endl;  
    return 0;  
}
```

Rezultat izvršavanja:

A
B
C

C++ podržava još jedan tip literala osim konstanti ugrađenih tipova, a to je string. String konstanta je skup znakova između navodnika. Na primer, “ovo je test” je string. U prethodnim primerima programa viđali smo stringove u naredbama `cout`. Treba zapamtiti da iako C++ dozvoljava definisanje string konstanti, on nema ugrađen tip podataka string. Umesto toga, kao što ćete saznati u nastavku knjige, stringovi su u jeziku C++ podržani kao nizovi znakova (u biblioteci klasa ugrađenoj u jezik takođe postoji klasa string, o kojoj će kasnije biti reči.)

U tip `wchar_t` smeštaju se znaci koji pripadaju većim skupovima od engleske abecede, tj. ASCII kôda (kao što je npr. Unicode). Mnogi jezici, kao što je kineski, imaju veliki broj znakova koji ne može da se kodira pomoću samo 8 bitova koliko staje u tip `char`. Zbog toga je u C++ dodat tip `wchar_t` koji u ovoj knjizi nećemo koristiti, ali ćete se sa njim sigurno susresti u programima namenjenim za inostrana tržišta.

2.3.4. Brojevi u pokretnom zarezu (float, double)

Vrednosti koje nisu celobrojne predstavljaju se kao brojevi u pokretnom zarezu (floating point). Broj u pokretnom zarezu može se predstaviti kao decimalna vrednost (npr. 112.5), ili u eksponencijalnoj notaciji. U eksponencijalnoj notaciji, decimalni deo ispred simbola E (ili e) množi se sa deset na stepen (eksponent) zadat nakon slova E. Na primer, 1.125e2 znači 1.125×10^2 , a to je 112.5. Praznine unutar broja (npr. iza predznaka, ili između cifara i slova) nisu dozvoljene. Konstanta u pokretnom zarezu mora da sadrži decimalnu tačku, eksponent, ili oba. Ako nema ni tačke ni eksponenta, numerička vrednost se tumači kao ceo broj. Za brojeve u pokretnom zarezu koriste se tipovi `float` i `double`; razlika između ova dva tipa je u opsegu koji mogu da sadrže. U tip `double` može da stane približno deset puta veći broj nego u tip `float`. Zbog tog se tip `double` češće koristi. Još jedan razlog za češće korišćenje tipa `double` jeste to što većina matematičkih funkcija u C++ biblioteci koristi `double` vrednosti.

```
float pi = 3.1415;  
float naelektrisanjeElektrona = 1.6e-19;  
double masaSunca = 2E30;
```

2.3.5. Logički tip (bool)

Tip `bool` type je relativno nov dodatak u jezik C++. U njemu se čuvaju logičke vrednosti (tj. tačno/netačno). C++ definiše dve logičke konstante, `true` i `false`, i to su jedine dve vrednosti koje tip `bool` može da ima.

Bilo koja vrednost različita od nule u jeziku C++ tumači se kao tačno (`true`), a vrednost nula tumači se kao netačno (`false`); ovo svojstvo je naročito korisno u logičkim izrazima. Kada se koristi u izrazima koji nisu logički, važi obrnuto: `true` se konvertuje u 1, a `false` u 0. Automatska konverzija vrednosti nula i vrednosti različitih od nule u logičke ekvivalente posebno je važna u kontrolnim naredbama, o čemu će biti reči u Poglavlju 3.

```
#include <iostream>

using namespace std;

int main() {
    bool b = false;
    cout << "b je " << b << endl;
    b = true;
    cout << "b je " << b << endl;
    return 0;
}
```

Rezultat izvršavanja:

```
b je 0
b je 1
```

2.4. Operatori

2.4.1. Operator dodele

Operator dodele smo već koristili u prethodnim primerima programa, a sada ćemo se njime detaljnije pozabaviti. Operator dodele (assignment operator) u jeziku C++ radi slično kao i u drugim programskim jezicima i ima opšti oblik

```
promenljiva = izraz;
```

što znači da se vrednost izraza dodeljuje promenljivoj. Operator dodele se može koristiti i lančano, kao u sledećem primeru:

```
int x, y, z;
x = y = z = 100;
```

Ova naredba dodeljuje vrednost 100 promenljivama `x`, `y` i `z` u jednom koraku. To je moguće zato što je rezultat operatora dodele izraz sa njegove desne strane. Tako je rezultat naredbe `z = 100` zapravo 100, taj rezultat se zatim dodeljuje promenljivoj `y`, i na kraju se rezultat izraza `y = 100` (što je opet 100) dodeljuje promenljivoj `x`.

Operatorom dodele menja se vrednost objekta, pri čemu tip objekta ostaje nepromenjen. Najčešći operator dodele je znak jednakosti (`=`) kojim se objektu sa leve

strane dodeljuje neka vrednost sa desne strane. Ako su objekat sa leve strane i vrednost sa desne strane različitih tipova, vrednost se svodi na tip objekta prema definisanim pravilima konverzije, što će biti objašnjeno u odeljku 2.5. Očigledno je da se s leve strane operatora dodele mogu nalaziti samo promenljivi objekti, tzv. *lvrednosti* (engl. *lvalues*, skraćenica od *left-hand side values*). Pri tom treba znati da se ne može svakoj lvrednosti dodeljivati nova vrednost, jer se neke promenljive mogu deklarirati i tako da budu konstantne, pa će pokušaj njihove promene kompajler prepoznati kao grešku. Sa desne strane operatora dodele mogu biti i lvrednosti i konstante.

```
3.1415 = pi; //greska!
int i;
i = i + 5;
int j = 5;
```

2.4.2. Aritmetički operatori

Operatori +, -, * i / u jeziku C++ imaju isto značenje kao u algebri i mogu se primeniti na bilo koji ugrađen numerički tip podataka (kao i na podatke tipa char). Mogu se podeliti na unarne, koji deluju na samo jedan operand, i binarne, koji rade sa dva operanda.

Operator % (modulo) daje ostatak celobrojnog deljenja. Kada se operator / primeni na podatak tipa int, eventualni ostatak deljenja se odseca; na primer, rezultat operacije 10/3 daće 3. Ostatak celobrojnog deljenja može se dobiti operacijom modulo (%). Na primer, 10 % 3 daje 1. U jeziku C++, operator % može da se primeni samo na celobrojne operande, ne i na podatke u pokretnom zarezu.

Tabela 2.8: Aritmetički operatori

Operator	Značenje	Tip
+	sabiranje	binarni
-	oduzimanje	binarni
*	množenje	binarni
/	deljenje	binarni
%	modulo	binarni
++	inkrementiranje	unarni
-	dekrementiranje	unarni

Operatori inkrementiranja i dekrementiranja

Operator ++ uvećava vrednost celobrojne promenljive za jedan, što se zove inkrementiranje. Operator - smanjuje vrednost celobrojne promenljive za 1, tj. dekrementira je.

```
int i = 0;
++i;
cout << i; //ispisuje 1
--i;
cout << i; //ispisuje 0
```

$x++$ je isto što $i x = x + 1$, a $x--$ je isto što $i x = x - 1$. Treba uočiti razliku kada je operator inkrementiranja (dekrementiranja) napisan ispred operatora ili iza njega. Kada je napisan ispred operanda, radi se o prefiksnom obliku; tada će se vrednost promenljive prvo inkrementirati (dekrementirati), a zatim će se raditi sa promenjenom vrednošću. Ako je operator iza promenljive, prvo će se pročitati vrednost promenljive, a tek zatim će ona biti promenjena.

U prethodnom primeru svejedno je da li su operatori napisani u prefiksnom ili u postfixnom obliku. Međutim, kada je inkrementiranje ili dekrementiranje deo složenijeg izraza, to postaje važno. Na primer:

```
int x = 1, y;
y = ++x; //y je 2, x je 1
y = x--; //y je 2, x je 1
```

Skraćeni operatori dodele

C++ je od jezika C nasledio i skraćeno pisanje naredbi. Opšti oblik skraćene naredbe je:

```
levo op= desno;
```

gde je op neki od sledećih operanada:

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

Neke od ovih operatora smo već videli, a sa nekima ćemo se susresti u nastavku poglavlja. Značenje ove notacije je isto kao naredba:

```
levo = levo op desno;
```

Tako su, na primer, sledeće tri naredbe funkcionalno identične:

```
x = x + 1;
x++;
x+=1; //skracena notacija
```

Skraćeni operatori dodele imaju dve prednosti. Prvo, kompaktniji su od odgovarajućih standardnih naredbi. Drugo, često daju efikasniji izvršni kôd, jer se operand proverava samo jednom. Zbog toga se ovakav oblik naredbi dodele često sreće u profesionalno napisanim C++ programima.

2.4.3. Relacioni i logički operatori

Relacioni operatori omogućuju poređenje logičkih vrednosti i određivanje vrednosti koja je veća, manja, jednaka ili različita. Svi relacioni operatori su binarni, tj. zahtevaju dva operanda i pridružuju se sleva udesno. U Tabeli 2.9 prikazani su relacioni operatori i njihovo značenje:

Tabela 2.9: Relacioni operatori

Izraz	Značenje
$x > y$	Da li je x veće od y?
$x < y$	Da li je x manje od y?
$x \geq y$	Da li je x veće ili jednako y?
$x \leq y$	Da li je x manje ili jednako y?
$x == y$	Da li je x jednako y?
$x != y$	Da li je x različito od y?

Treba zapamtiti suštinsku razliku između jednostrukog znaka jednakosti (=) koji je simbol za dodelu, i dvostrukog znaka jednakosti (==) koji je simbol za poređenje. Rezultat logičke operacije može da bude tačan ili pogrešan. Netačno poređenje (kada je rezultat relacione operacije false) uvek se predstavlja kao 0, dok je vrednost tačnog poređenja (true) broj 1.

```
//program koji prikazuje vrednosti stanja true i false
#include <iostream>

using namespace std;

int main() {
    int tacnaVrednost, pogresnaVrednost, x=5, y=10;
    tacnaVrednost = x < y;
    pogresnaVrednost = x == y;
    cout << "Tacno je: " << tacnaVrednost << endl;
    cout << "Pogresno je: " << pogresnaVrednost << endl;
    return 0;
}
```

Rezultat izvršavanja:

Tacno je 1

Pogresno je 0

Logički operatori povezuju nekoliko relacionih izraza u jedan, ili invertuju logiku izraza. U Tabeli 2.10 prikazani su logički operatori po redosledu prioriteta:

Tabela 2.10: Logički operatori

!	NOT	Logička negacija
&&	AND	Logičko i
	OR	Logičko ili

Logička negacija je unarni operator koji menja logičku vrednost promenljive (iz true u false ili obrnuto). Logičko i je binarni operator koji kao rezultat daje true

samo ako su oba operanda `true`. Logičko ili je takođe binarni operator; da bi dao tačnu vrednost, dovoljno je da barem jedan od operandi ima vrednost `true`. Logički operatori najčešće se koriste u naredbama za grananje toka programa, pa ćemo ih kasnije detaljnije proučiti.

2.4.4. Definisanje sinonima za tipove podataka: `typedef`

Ključna reč `typedef` omogućuje definisanje sopstvenih naziva za postojeće tipove podataka. Na primer, pomoću ključne reči `typedef` mogli bismo da uvedemo naziv `VelikiCeoBroj` kao zamenu za tip `long int` pomoću sledeće deklaracije:

```
typedef long int VelikiCeoBroj;
```

Na ovaj način `VelikiCeoBroj` postaje alternativni specifikator tipa za `long int`, pa možemo da deklariramo promenljivu `mojBroj` kao `long int` pomoću sledeće deklaracije:

```
VelikiCeoBroj mojBroj = 0L;
```

Nema razlike između ove deklaracije i one koja koristi ime ugrađenog tipa. I dalje može da se piše:

```
long int mojBroj = 0L;
```

Dakle, čak i ako se definiše ime tipa kao što je `VelikiCeoBroj`, i dalje mogu da se koriste oba specifikatora za deklarisanje različitih promenljivih koje će na kraju biti istog tipa. Iako je korišćenje reči `typedef` u funkciji sinonima za postojeći tip na prvi pogled trivijalno, ono je važno za uprošćavanje složenijih deklaracija, što čini kôd mnogo čitljivijim. Takođe, ako je isti program potrebno optimizovati i ponovo kompajlirati za različite platforme, preporučljivo je definisati sinonime za ugrađene tipove pomoću ključne reči `typedef` (na primer, ako za celobrojne promenljive na jednoj platformi treba da se koristi tip `int`, a na drugoj `long`, tip svih celobrojnih promenljivih deklarisanih u programu može se lako promeniti na jednom mestu - samo u deklaraciji `typedef`).

2.4.5. Operator `sizeof`

Već smo pomenuli da se veličina memorijskog prostora i opseg vrednosti koji određeni tip zauzima menjaju u zavisnosti od kompajlera i platforme. Najvažniji podaci o celobrojnim opsezima mogu se naći u zaglavlju `climits`, a za decimalne brojeve u zaglavlju `cmath`. Na primer, u zaglavlju `climits` definisane su konstante `INT_MIN` i `INT_MAX` sa najmanjom i najvećom vrednošću tipa `int`.

U jeziku C++ postoji i poseban operator `sizeof` koji prikazuje dužinu određenog tipa u bajtovima. Njegovo korišćenje ilustruje sledeći primer:

```
#include <iostream>
#include <climits>
#include <cfloat>

using namespace std;

int main() {
    cout << "Najmanji int: " << INT_MIN << endl;
    cout << "Najveci int: " << INT_MAX << endl;
    cout << "Najmanji double: " << DBL_MIN << endl;
    cout << "Najveci float: " << FLT_MAX << endl;
    cout << "Tip double na ovom racunaru zauzima " << sizeof(double)
    << " bajtova" << endl;
    cout << "Tip long zauzima " << sizeof(long) << " bajta" << endl;
    return 0;
}
```

Rezultat izvršavanja:

```
Najmanji int: -2147483648
Najveci int: 2147483648
Najmanji double: 2.22507e-308
Najveci float: 3.40282e+038
Tip double na ovom racunaru zauzima 8 bajtova
Tip long zauzima 4 bajta
```

2.4.6. Operator nabranjanja (,)

Operator nabranjanja (ili razdvajanja) koristi se za razdvajanje izraza u naredbama. Navedeni izrazi se izvršavaju postepeno, sleva udesno, a rezultat operatora je vrednost poslednjeg izraza u nabranjanju. Prilikom korišćenja operatora zarez u složenijim naredbama treba biti vrlo oprezan, jer mu je prioritet vrlo nizak.

```
#include <iostream>

using namespace std;

int main() {
    long num1=0, num2=0, num3=0, num4=0;
    num4=(num1=10, num2=20, num3=30);
    cout << "Vrednost niza izraza je vrednost " <<
    " poslednjeg izraza sa desne strane: " << num4 << endl;
    return 0;
}
```

Rezultat izvršavanja:

Vrednost niza izraza je vrednost poslednjeg izraza sa desne strane: 30

2.4.7. Osnovne operacije ulaza i izlaza

Ulaz i izlaz podataka u jeziku C++ zasnovani su na apstraktnom pojmu tokova podataka (stream), kojima ćemo se detaljnije baviti u Poglavlju 8.

U prethodnim primerima videli smo kako se objekat izlaznog toka `cout` koristi za ispis podataka na konzolu pomoću operatora umetanja (insertion operator) `<<`. Podaci se šalju u izlazni tok (output stream), odnosno čitaju iz ulaznog toka (input stream). Odgovarajući objekat ulaznog toka za čitanje podataka sa tastature zove se `cin`. Znači se sa tastature u objekat ulaznog toka `cin` učitavaju se pomoću operatora izdvajanja (`>>`):

```
cin >> promenljiva;
```

Operator izdvajanja (extraction operator) “pokazuje” u smeru toka podataka: u ovom slučaju, iz objekta ulaznog toka `cin` na promenljivu koja prima podatke. Ako ima praznina, one se preskaču, a prva vrednost koja se otkuca na tastaturi biće dodeljena promenljivoj jer se naredbe ulaza izvršavaju sleva udesno. Operacija učitavanja iz toka završava se kada se pritisne taster Enter. Operacije učitavanja iz ulaznog toka automatski rade sa svim promenljivama ugrađenih tipova. Za formatiranje ulaza i izlaza koriste se tzv. manipulatori i o tome ćemo detaljnije učiti u Poglavlju 8.

2.4.8. Prioritet i asocijativnost operatora

U Tabeli 2.11 prikazan je prioritet operatora. Mnoge od njih još nismo objasnili, ali ćemo se sa njima susresti u nastavku knjige.

Operatori su u tabeli navedeni u grupama po opadajućem prioritetu. Vodoravne linije u tabeli označavaju promenu u prioritetu operatora.

Ako u izrazu nema zagrada, operacije istog prioriteta izvršavaju se redosledom koji je definisan njihovom asocijativnošću, tj. pridruživanjem. Tako, ako je pridruživanje “sleva”, onda se prvo izvršava operacija koja je prva gledano sa leve strane, a zatim redom ostale kako se kreće udesno. To znači da se izraz kao što je $a + b + c + d$ izvršava kao da je napisano $((a + b) + c) + d$ pošto se binarni operator sabiranja + pridružuje sleva. U slučajevima kada isti operator ima i unarni i binarni oblik, unarni oblik je uvek višeg prioriteta.

Prefiksni unarni operatori i operatori dodele pridružuju se zdesna ulevo. Svi ostali operatori pridružuju se sleva udesno. Na primer,

$x - y - z$ znači $(x - y) - z$ jer se operator `-` pridružuje sleva udesno, ali

$x = y = z$ znači $x = (y = z)$ jer se operator `=` pridružuje zdesna ulevo.

Prioritet operatora može se uvek promeniti pomoću zagrada. Pošto jezik C++ ima mnogo operatora, ponekad nije lako odrediti redosled izvršavanja operacija i zato zagrade treba koristiti i kada nisu neophodne jer poboljšavaju čitljivost kôda.

Tabela 2.11: Prioritet i asocijativnost operatora.

Operator	Opis
::	Razrešenje dosega (scope)
.	Pristup podatku članu
->	Dereferenciranje i pristup podatku članu
[]	Indeksiranje niza
()	Poziv funkcije
++	Inkrementiranje
--	Dekrementiranje
typeid	Identifikacija tipa tokom izvršavanja
const_cast	Ukidanje nepromenljivosti tipa
dynamic_cast	Konverzija tipa tokom izvršavanja
static_cast	Konverzija tipa tokom prevođenja
reinterpret_cast	Neproverena konverzija tipa
!	Logička negacija
~	Bitska negacija
+ (unarni) - (unarni)	Promena znaka
* (unarni)	Dereferenciranje pokazivača
& (unarni)	Adresa promenljive
new	Dinamička alokacija memorije
delete	Oslobađanje dinamičke memorije
sizeof	Veličina promenljive ili tipa
. * (unarni)	Pristup podatku članu
->*	Dereferenciranje i pristup podatku članu
*	Množenje
/	Deljenje
%	Modulo (ostatak deljenja)
+ -	Sabiranje i oduzimanje
<< >>	Pomeranje za jedan bit ulevo ili udesno
< <= > >=	Relacioni operatori
== !=	Operatori jednakosti
&	Bitsko i (AND)
^	Bitsko isključivo ili (XOR)
	Bitsko ili (OR)
&&	Logičko i

Tabela 2.11: Prioritet i asocijativnost operatora.	
	Logičko ili
: ?	Uslovni izraz
= *= /= %= += -=	Operatori dodele
&= ^= >= <=< >>=	Operator nabravanja
,	

2.5. Konverzija tipova (cast)

Kada u izrazu dodele učestvuju operatori različitih tipova, vrednost desne strane (izraza) biće konvertovana u tip leve strane (tip promenljive). U slučaju izračunavanja, pravila za konverziju tipova su složenija. Izračunavanja u jeziku C++ mogu se izvoditi samo sa vrednostima istog tipa. Kada se napiše izraz u kome se nalaze promenljive ili konstante različitih tipova, za svaku operaciju kompajler mora da konvertuje tip jednog od operandata tako da odgovara onom drugom. To se zove *konverzija tipa* (casting). Na primer, ako želimo da saberemo vrednost tipa double sa vrednošću tipa int, celobrojna vrednost se prvo konvertuje u tip double, a zatim se obavlja sabiranje. Naravno, sama promenljiva koja sadrži vrednost koja se konvertuje ostaje neizmenjena.

Postoje pravila koja upravljaju izborom operandata koji će biti konvertovan u operacijama. Bilo koji izraz koji se izračunava raščlanjuje se u niz operacija između dva operandata. Na primer, izraz $2*3-4+5$ se raščlanjuje na $2*3$ što daje 6, $6-4$ daje 2, i na kraju $2+5$ daje 7. Zbog toga se pravila koja upravljaju konverzijom definišu samo na nivou para operandata. Za bilo koji par operandata različitih tipova proveravaju se sledeća pravila konverzije i to u redosledu kojim su navedena:

1. Ako je bilo koji operand tipa long double, drugi se konvertuje u long double.
2. Ako je bilo koji operand tipa double, drugi se konvertuje u double.
3. Ako je bilo koji operand tipa float, drugi se konvertuje u float.
4. Bilo koji operand tipa char, signed char, unsigned char, short ili unsigned short konvertuje se u int.
5. Ako je bilo koji operand tipa unsigned long, drugi se konvertuje u unsigned long.
6. Ako je jedan operand tipa long a drugi tipa unsigned int, oba operandata se konvertuju u tip unsigned long.
7. Ako je bilo koji operand tipa long, drugi se konvertuje u long.

Iako ova pravila na prvi pogled izgledaju složeno, princip je da se promenljiva čiji je opseg više ograničen uvek konvertuje u tip šireg opsega. Kada u izrazu učestvuju operandi različitih tipova, svi se konvertuju u tip najvećeg operandata (to je tzv. unapređivanje tipa).

```
int i = 3;
float f = 0.5;
cout << i * f << endl; //ispisuje 1.5
```

U ovom slučaju na ekranu se ispisuje 1.5 jer je tip `float` složeniji od tipa `int`. Celobrojnu promenljivu `i` kompajler pre prevođenja konvertuje u tip `float` (prema pravilu broj 3), sprovodi množenje dva broja u pokretnom zarezu i daje rezultat koji je tipa `float`. Da smo rezultat množenja pre ispisa dodelili nekoj celobrojnoj promenljivoj, kao u sledećem primeru:

```
i *=f;
cout << i; //ispisuje 1
```

dobili bismo celobrojni rezultat, tj. 1, jer se decimalni deo rezultata gubi prilikom dodele.

Problem konverzije najizraženiji je kod celobrojnog deljenja, što zbog nepažnje često prouzrokuje probleme.

```
int brojilac = 1;
int imenilac = 4;
float razlomak = brojilac / imenilac;
cout << razlomak << endl; //ispisuje 0
```

Iako smo rezultat deljenja dodelili promenljivoj tipa `float`, ta dodela tokom izvršavanja sledi tek nakon što je operacija deljenja dva cela broja završena. Pošto su `i` brojilac i `imenilac` celobrojne promenljive, izvodi se celobrojno deljenje koje kao rezultat daje 0. Da bi se izbegli ovakvi problemi, sve konstante u pokretnom zarezu treba pisati sa decimalnom tačkom:

```
float diskutabilniRazlomak = 1 / 4.; //daje 0
float razlomak = 1. / 4.; //daje 0.25
```

Izraz se može "primorati" da bude određenog tipa pomoću eksplicitne konverzije tipa (`cast`). Najopštiji oblik konverzije tipa je:

```
static_cast<tip> (izraz)
```

Ovde je tip ciljni tip podataka u koji želimo da konvertujemo izraz. Eksplicitna konverzija tipa često se koristi kada je u pitanju nasleđeni kôd, a želimo da izbegnemo upozorenja prevodioca.

```
float pi = 3.1415926;
cout << "celi deo broja pi je: " <<
static_cast<int>(pi) << endl; //ispisuje 3
//i ova sintaksa je podrzana, ali se ne preporučuje
cout << "celi deo broja pi je: " << (int) pi << endl;
```

Operatori konverzije su unarni i imaju isti prioritet kao i drugi unarni operatori.

2.6. Simboličke konstante

U programima se često koriste promenljive čija vrednost tokom izvršavanja ne treba da se menja. To mogu biti matematičke konstante, ali i neki parametri poput

maksimalne dužine niza i sl. Da bi se sprečilo slučajno menjanje takvih promenljivih, one se deklariraju kao konstantne pomoću ključne reči `const`. Tada ih kompajler proverava i prijavljuje grešku prilikom pokušaja da se one u programu izmene.

```
const double pi = 3.141592653;
```

```
pi = 2 * pi; //greska!
```

Konstantna promenljiva prilikom definicije mora da bude i inicijalizovana:

```
const int mojaKonstanta; //greska!
```

Drugi pristup definisanju konstantnih promenljivih nasleđen je iz jezika C i koristi preprocesorsku direktivu `#define` (Dodatak B):

```
#define PI 3.141592653
```

U ovom slučaju kompajler će pre prevođenja svako pojavljivanje konstante `PI` u programu zameniti odgovarajućom vrednošću.

```
double precnik = 2;
```

```
double obimKrug = precnik * PI;
```

```
PI = 2 * PI; //greska!
```

Na prvi pogled nema razlike između ova dva pristupa, jer će i prilikom korišćenja preprocesorske direktive kompajler javiti grešku zato što se konstantna vrednost mora naći sa leve strane operacije dodele, što nije dozvoljeno. Međutim, ako se konstanta definiše pomoću direktive `#define`, njen naziv neće postojati u simboličkoj tabeli koju generiše kompajler, pa neće biti vidljiv tokom debugovanja programa, što je veliki nedostatak. Zbog toga se preporučuje C++ stil definisanja konstantnih promenljivih pomoću ključne reči `const`.

Pitanja

1. Koja od sledećih imena su prihvatljiva u jeziku C++?

- a) danUNedelji
- b) 3dGrafik
- c) _broj_zaposlenog
- d) Jun1997
- e) Mesavina#3

2. Ispravno deklarisanе promenljive su:

- a) `int ime prezime;`
- b) `int ime_prezime;`
- c) `int ImePrezime;`
- d) `int 5deset;`
- e) `int prezime_25;`
- f) `int _ime;`

3. U kojim slučajevima su ispravno napisane naredbe za deklarisanje promenljivih:

- a) `int duzina; int sirina;`
- b) `int duzina, sirina;`
- c) `int duzina; sirina;`
- d) `int duzina, int sirina;`

4. Reči `main` i `Main` su istog značenja u jeziku C++ ?

- a) tačno
- b) pogrešno

5. Definicija promenljive:

- a) rezerviše prostor u memoriji za promenljivu
- b) inicijalizuje promenljivu
- c) određuje samo tip promenljive

6. Tip podataka definiše:

- a) skup mogućih vrednosti tipa
- b) skup različitih podataka tipa
- c) skup dozvoljenih operacija nad vrednostima tipa

7. Celobrojni tip vrednosti možemo predstaviti sa:

- a) `int`
- b) `short`
- c) `char`
- d) `long`
- e) `unsigned`
- f) sve od navedenog

8. Promenljive čija je vrednost inicijalizovana su:

- a) `int a;`
- b) `int b = 2;`
- c) `float d;`

9. Promenljiva mora da bude deklarisan pre nego što se upotrebi.

- a) tačno
- b) pogrešno

10. Promenljiva može da bude deklarisan bilo gde u C++ programu.

- a) tačno
- b) netačno

11. Nazivi promenljivih mogu da počinju brojem.

- a) tačno
- b) netačno

12. Ako promenljiva treba da sadrži vrednosti od 32 do 6000, koji tip podataka bi bio najpogodniji?

- a) `short int`
- b) `int`
- c) `long`

13. Tip svake promenljive u jeziku C++ mora biti poznat tokom kompajliranja.

- a) tačno
- b) pogrešno

14. Tip `char` se može koristiti i kao "mali" `int`.

- a) tačno
- b) pogrešno

15. Tip `float` je precizniji od tipa `double`.

- a) tačno
- b) pogrešno

16. Koje vrednosti može da ima promenljiva tipa `bool`?

- a) `true` i `false`
- b) 0 i 1

17. Svaka vrednost različita od nule tumači se kao `true`.

- a) tačno
- b) pogrešno

18. Sledeći podaci:

72

'A'

"Hello world"

2.87

su primeri:

- a) promenljivih
- b) literala ili konstanti
- c) stringova
- d) ničega od navedenog

19. Pravilno deklarisanе i inicijalizovane promenljive su:

- a) `Int a = 5;`
- b) `double b = "abc";`
- c) `int f = 6, t = 5;`
- d) `char c = 18.25;`

20. Koje su od sledećih službene reči u jeziku C++:

- a) `public`
- b) `static`
- c) `main`
- d) `class`
- e) tačno

21. Koji od sledećih tipova zahteva najmanje memorije za predstavljanje celih brojeva:

- a) `int`
- b) `short`
- c) `char`
- d) `float`

22. Sledeća operacija dodele je važeća: `72 = iznos.`

- a) tačno
- b) netačno

23. Koji tip kompajler dodeljuje literalu `12.2`?

- a) `float`
- b) `double`
- c) `int`

24. String konstanta se piše pod navodnicima.

- a) tačno
- b) pogrešno

25. Heksadecimalne konstante počinju sa:

- a) 0
- b) 0x
- c) x
- d) 16

26. Koje od sledećih nisu validne naredbe dodele?

- a) `ukupno = 9;`
- b) `72 = iznos;`
- c) `zarada = 12000;`
- d) `slovo := 'D';`

27. Kojom se od sledećih naredbi ispravno dodeljuje vrednost 17 promenljivoj `x`?

- a) `x = 17;`
- b) `x := 17;`
- c) `x == 17;`
- d) `17 = x;`

28. Operator logičke negacije (!) je:

- a) unarni
- b) binarni
- c) ternarni
- d) ništa od navedenog

29. Koja od sledećih je string konstanta?

- a) `"H"`
- b) `'H'`
- c) `H`

30. Koja od sledećih je znakovna konstanta?

- a) `"H"`
- b) `'H'`
- c) `H`

31. Pravilno prihvatanje ulaza sa konzole u C++ je:

- a) `cout<<`
- b) `cout>>`
- c) `cin>>`
- d) `cin<<`

32. Ispravno napisana naredba za izlaz na konzolu je:

- a) `cout>>`
- b) `cout<<`
- c) `cin<<`

33. Deklarisana je i inicijalizovana promenljiva `int a = 1;` ; posle naredbe `a++;` njena vrednost je:

- a) 3
- b) 1
- c) 2

34. Deklarisana je i inicijalizovana promenljiva `int a = 10;` ; posle naredbe `a += 3;` njena vrednost je:

- a) 7
- b) 10
- c) 13

35. Redosled izvršavanja operatora je:

- a) unapred definisan
- b) nije bitan
- c) može se menjati zagradama
- d) sve navedeno

36. Izraz `a = b` je:

- a) dodela vrednosti promenljive `b` promenljivoj `a`
- b) poređenje vrednosti promenljivih `a` i `b`
- c) dodela vrednosti promenljive `a` promenljivoj `b`

37. Izraz `x++` može se napisati i kao:

- a) `x = x + 1`
- b) `x = x - 1`
- c) `x == x`

38. Kolika je vrednost promenljive `j` nakon izvršavanja sledećeg koda?

```
int i, j;
```

```
i=5;
```

```
j=5+--i;
```

- a) 10
- b) 9
- c) 11

39. Šta radi sledeći deo kôda?

```
int i=3, j=5, temp;
```

```
temp=i;
```

```
i=j;
```

```
j=temp;
```

- a) Vrednost promenljive j dodeljuje promenljivoj i
- b) Vrednost promenljive i dodeljuje promenljivoj j
- c) Međusobno zamenjuje vrednosti promenljivama i i j

40. % je operator:

- a) celobrojnog deljenja
- b) ostatka celobrojnog deljenja
- c) izračunavanja procenta

41. Izraz $x=x+12$ može se napisati i kao $x+=12$

- a) tačno
- b) pogrešno

42. Koji od sledećih operatora su logički?

- a) `&&`
- b) `##`
- c) `||`
- d) `//`
- e) `!`

43. Ako se na promenljivu c primeni operator inkrementiranja u prefiksnoj formi, onda se to piše kao:

- a) `c++`
- b) `++c`
- c) `c = c+1`

44. Operator `sizeof`:

- a) vraća veličinu tipa u bitovima
- b) vraća veličinu tipa u bajtovima
- c) služi za preimenovanje tipova

45. Operator `typedef`:

- a) vraća veličinu tipa u bitovima
- b) vraća veličinu tipa u bajtovima
- c) služi za preimenovanje tipova

46. Kolika je vrednost promenljive broj4 nakon sledećih naredbi:

```
int broj1=0, broj2=0, broj3=0, broj4=0;
```

```
broj4 = (broj1=10, broj2=20, broj3=30);
```

a) 10

b) 20

c) 30

47. Šta će ispisati sledeći kôd?

```
int a = 10;
```

```
float b = 20.;
```

```
cout << a / b;
```

a) 0

b) 0.5



Glava 3

Naredbe za kontrolu toka programa

Izvršavanje iole korisnih programa nikada nije pravolinijsko, već se često javlja potreba da se delovi kôda ponavljaju, ili da se u zavisnosti od nekog uslova izvršava određeni deo kôda, a da se drugi deo preskače. Grananje toka programa i ponavljanje delova kôda omogućuju posebne naredbe koje je C++ preuzeo iz jezika C.

3.1. Blokovi naredbi

Delovi programa koji se uslovno izvode ili čije se izvršavanje ponavlja grupišu se u blokove naredbi. Nekoliko naredbi se grupiše unutar vitičastih zagrada, i postaje blok naredbi ili složena naredba. Primer za to su naredbe funkcije `main()`, koje se uvek nalaze između para otvorenih i zatvorenih zagrada. Blok naredbi se može posmatrati i kao jedna složena naredba; gde god je u jeziku C++ dozvoljeno pisanje naredbe, može se napisati i blok naredbi omeđen zgradama. Važno svojstvo blokova jeste to da su promenljive koje su definisane unutar bloka vidljive samo u njemu. Zbog toga će se program:

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

kompajlirati bez grešaka, ali ako se deklaracija promenljive izmesti izvan bloka, kompajler će prijaviti grešku:

```
int main() {  
    {  
        int a = 1;  
    }  
    cout << a << endl; //greska! a vise ne postoji  
    return 0;  
}
```

Promenljive koje su deklarisanе unutar bloka zovu se *lokalne promenljive* za taj blok. Ograničenje oblasti važenja promenljive zove se *doseg* (scope) i omogućuje da se unutar blokova deklariraju promenljive sa istim imenom koje je već upotrebljeno izvan bloka. Lokalna promenljiva će unutar bloka zakloniti promenljivu istog imena koja je već deklarirana izvan bloka:

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

Opseg važnosti (doseg) promenljivih kasnije ćemo detaljnije proučiti. Za sada, zapamtimo i da ako se blok u naredbama za kontrolu toka sastoji samo od jedne naredbe, onda se vitičaste zagrade mogu i izostaviti.

3.2. Naredba if

Kompletan oblik naredbe `if` je:

```
if (izraz)
    naredba;
else
    naredba;
```

Deo `else` u naredbi nije obavezan. Iza ključnih reči `if` i `else` mogu se nalaziti i blokovi naredbi, koji se smeštaju unutar vitičastih zagrada.

Ako je izraz tačan, izvršiće se naredbe iza `if`; u suprotnom, izvršiće se blok iza `else`, ako postoji. Nikako se ne može desiti da budu izvršene i naredbe iza `if` i naredbe iza `else`. Sledeći program koristi naredbu `if else` da bi odredio neke osobine celog broja koji se unosi sa tastature:

```
//grananje naredbom if else
#include <iostream>

using namespace std;
```

```
int main() {
    int a;
    cout << "Unesite ceo broj: ";
    cin >> a;
    if(a<0)
        cout << "uneti broj je negativan" << endl;
    if(a%2)
        cout << "uneti broj je neparan";
    else
        cout << "uneti broj je paran";
    cout << endl;
    return 0;
}
```

Rezultat izvršavanja:

unesite ceo broj: -5
uneti broj je negativan
uneti broj je neparan

Uslovni izraz koji kontroliše `if` može da bude bilo koji važeći C++ izraz koji kao rezultat daje logičke vrednosti `true` ili `false`, što početnike ponekad zna da zbuni. U izrazu ne moraju da se koriste isključivo relacioni i logički operatori, ili operandi tipa `bool`. Kao što smo već pominjali, vrednost 0 se automatski konvertuje u `false`, a sve vrednosti različite od nule se konvertuju u `true`. Zbog toga bilo koji izraz čiji je rezultat 0 ili neka vrednost različita od nule može da se koristi za kontrolisanje naredbe `if`. Kao primer, proučimo program koji učitava dva cela broja sa tastature i prikazuje rezultat celobrojnog deljenja. Da bi se izbegla greška deljenja nulom, koristi se naredba `if` kojom se upravlja pomoću drugog unetog broja (delioca):

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

int main() {
    int a, b;
    cout << "Unesite deljenik: ";
    cin >> a;
    cout << "Unesite delilac: ";
    cin >> b;
    if(b)
        cout << "Rezultat je " << a/b;
    else
        cout << "Deljenje nulom nije dozvoljeno";
}
```

```

    cout << endl;
    return 0;
}

```

Rezultat izvršavanja:

Unesite deljenik: 50
Unesite delilac: 10
Rezultat je: 10
Unesite deljenik: 50
Unesite delilac: 0
Deljenje nulom nije dozvoljeno.

Uočite da se unesena vrednost delioca `b` koristi kao uslov u `if` grananju. Ako je `b` nula, onda uslov koji kontroliše grananje ima vrednost `false` i izvršava se `else` deo. U suprotnom, ako `b` nije nula, izvršava se deljenje (deo `if`). Nije neophodno da se uslov piše kao `if (b == 0)`.

3.2.1. Ugnježdene naredbe `if`

Ugnježdene naredbe `if` je naredba `if` koja se nalazi unutar druge naredbe `if` ili `else`. Ugnježđivanje `if` naredbi veoma se često koristi u programiranju, a najvažnije je zapamtiti da se deo `else` uvek odnosi na najbližu `if` naredbu unutar istog bloka u kome se nalazi `else`, a koja već nije povezana sa nekom `else` naredbom. Evo primera koji to objašnjava:

```

if(i) {
    if(j)
        rezultat = 1;
    if(k)
        rezultat = 2;
    //ovaj else odnosi se na if(k)
    else
        rezultat = 3;
}
//ovaj else odnosi se na if(i)
else
    rezultat = 4;

```

3.2.2. If else if lestvica

Česta konstrukcija u programiranju zasnovana na ugnježđenim naredbama `if` jeste tzv. `if-else-if` lestvica, koja izgleda ovako:

```
if (uslov)
    naredba;
else if (uslov)
    naredba;
else if (uslov)
    naredba;
...
else
    naredba;
```

Uslovni izrazi proveravaju se s vrha naniže; čim se nađe na uslov koji ima vrednost `true`, izvršava se naredba povezana sa njim, a ostatak lestvice se preskače. Ako nijedan uslov nema vrednost `true`, izvršava se poslednja `else` naredba. Poslednji `else` ima ulogu podrazumevanog (default) uslova, jer se izvršava ako nijedan izraz nije tačan. Ako poslednji `else` ne postoji, a svi ostali uslovi imaju vrednost `false`, ne dešava se ništa. Primer programa koji računa diskriminantu kvadratne jednačine ilustruje korišćenje `if else if` lestvice:

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

int main() {
    float a, b, c;
    cout << "Unesite koeficijente kvadratne jednacine" << endl;
    cin >> a >> b >> c;
    float diskriminanta = b * b - 4 * a * c;
    cout << "Kvadratna jednacina ima ";
    if(diskriminanta > 0)
        cout << "dva realna korena";
    else if(diskriminanta < 0)
        cout << "dva kompleksna korena";
    else
        cout << "dvostruki realan koren";
    cout << endl;
    return 0;
}
```

Rezultat izvršavanja:

Unesite koeficijente kvadratne jednacine:

1
3
5

Kvadratna jednacina ima dva kompleksna korena.

3.2.3. Uslovni izraz (operator ? :)

Iako ne spada u naredbe za kontrolu toka programa, uslovni operator je sličan naredbi `if else` pa ćemo ga ovde ukratko objasniti. Sintaksa uslovnog operatora je:

```
uslov ? izraz1 : izraz2;
```

Ovaj operator zove se ternarni, jer zahteva tri operanda. Ako `uslov` ima vrednost `true`, izvršava se `izraz1`, a u suprotnom, izvršava se `izraz2`. U sledećem primeru:

```
x = (x > 0) ? x : -x; //apsolutna vrednost x
```

izračunava se apsolutna vrednost promenljive `x`. Ako je `x` pozitivan, izračunava se prvi izraz, tj. `x` ostaje nepromenjen. Ako je `x` negativan, izvršava se drugi izraz i menja se znak. Uslovni operator treba koristiti samo za vrlo jednostavna ispitivanja, kada ceo kôd staje u jedan red, jer u suprotnom kôd postaje nečitljiv.

3.3. Naredba switch

Druga naredba koja u jeziku C++ omogućuje izbor jeste naredba `switch`. Ona omogućuje višestruko grananje, odnosno izbor između nekoliko mogućnosti. Iako se isti efekat može postići i nizom ugnježđenih naredbi `if`, u mnogim slučajevima korišćenje naredbe `switch` je efikasnije. Opšti oblik naredbe je sledeći:

```
switch (izraz) {
    case (konstanta1):
        niz naredbi
        break;
    case (konstanta2):
        niz naredbi
        break;
    ...
    default:
        niz naredbi
}
```

Vrednost izraza, koji mora biti celobrojan, redom se upoređuje sa nizom konstanti; kada se utvrdi jednakost, izvršava se naredba (ili blok naredbi) pridružena ispunjenom uslovu. Često je izraz koji kontroliše naredbu `switch` neka celobrojna promenljiva. Konstante u delu `case` moraju da budu celobrojne. Mogu da postoje i `case` delovi bez pratećeg niza naredbi. Niz naredbi iza grananja `default` izvršava se ako nijedan prethodni uslov nije ispunjen. Blok `default` može se staviti bilo gde u naredbi `switch`, ali se zbog preglednosti po pravilu stavlja na kraj. Takođe, on nije obavezan; ako ne postoji, ne dešava se ništa.

Naredba `switch` razlikuje se od grananja `if` po tome što može da proverava samo jednakost (tj. da li postoji poklapanje vrednosti izraza i konstanti u delovima `case`),

dok u naredbi if uslovni izraz može da bude bilo kog tipa. Sve konstante u delovima case moraju da budu različite (naravno, ako se naredbe switch ugnježđuju, mogu imati iste vrednosti konstanti, ali je to vrlo redak slučaj). Naredba switch je obično efikasnija od ugnježđenih naredbi if. Naredba break koja sledi iza niza naredbi pridruženih grananju case tehnički nije obavezna, ali se skoro uvek koristi. Kada se završi izvršavanje niza naredbi pridruženog grananju case, naredba break prouzrokuje izlazak iz naredbe switch i nastavak izvršavanja programa iza nje. Kada na kraju grananja ne bi bilo naredbe break, nastavilo bi se izvršavanje svih preostalih case grananja do kraja naredbe switch, a to obično nije ono što želimo.

```
#include <iostream>

using namespace std;

int main() {
    int dan;
    cout << "Koji je danas dan po redu u nedelji? ";
    cin >> dan;
    switch(dan) {
        case(1):
            cout << "ponedeljak" << endl;
            break;
        case(2):
            cout << "utorak" << endl;
            break;
        case(3):
            cout << "sreda" << endl;
            break;
        case(4):
            cout << "cetvrtak" << endl;
            break;
        case(5):
            cout << "petak" << endl;
            break;
        case(6):
        case(7):
            cout << "vikend!" << endl;
            break;
        default:
            cout << "nema toliko dana u nedelji!" << endl;
    }
    return 0;
}
```

Rezultat izvršavanja:

**Koji je danas dan po redu u nedelji? 6
vikend!**

3.4. Petlja for

Često je u programima potrebno ponavljati delove kôda. Ako je broj ponavljanja unapred poznat, najpogodnije je koristiti petlju `for` čija je opšta sintaksa:

```
for(inicijalizacija; izraz; inkrement) blok naredbi;
```

Inicijalizacija je obično naredba dodele kojom se postavlja početna vrednost promenljive koja kontroliše petlju, tj. koja se ponaša kao brojač. Izraz je uslovni izraz koji određuje da li će petlja biti ponovljena, a inkrement definiše korak za koji se kontrolna promenljiva petlje menja u svakoj iteraciji (ponavljanju). Ova tri važna dela `for` petlje moraju biti razdvojena tačkom-zarezom. Petlja `for` nastaviće da se izvršava sve dok je rezultat kontrolnog izraza `true`. Kada taj uslov postane `false`, izvršavanje petlje će biti prekinuto i nastaviće se od prve naredbe iza bloka naredbi.

Često je promenljiva koja kontroliše `for` petlju potrebna samo unutar petlje. Zbog toga se ona najčešće i deklarise u delu `for` petlje koji obavlja inicijalizaciju. Ipak, treba imati na umu da je tada doseg (scope) kontrolne promenljive samo `for` petlja, te da se ona izvan nje ne vidi.

```
//deklaracija kontrolne promenljive  
//u inicijalizacionom delu for petlje  
for(int i=0; i<10; i++)
```

Kao primer proučimo program koji koristi petlju `for` za ispis kvadratnog korena brojeva između 1 i 99.

```
#include <iostream>  
#include <cmath>  
using namespace std;  
int main() {  
    int broj;  
    double kv_koren;  
    for(broj=1; broj<100; broj++) {  
        kv_koren = sqrt(static_cast<double>(broj));  
        cout << kv_koren << endl;  
    }  
    return 0;  
}
```

U ovom primeru, kontrolna promenljiva petlje zove se `broj`. Ovaj program koristi standardnu funkciju `sqrt()` za izračunavanje kvadratnog korena. Pošto argument

ove funkcije mora biti tipa `double`, neophodno je konvertovati brojač `for` petlje koji je celobrojan u tip `double`. Takođe, potrebno je navesti i zaglavlje `<cmath>` u kome se nalaze matematičke funkcije.

Petlja `for` može se izvršavati „unapred“ ili „unazad“, tj. kontrolna promenljiva se može povećavati ili smanjivati za bilo koji iznos. Na primer, sledeći program ispisuje brojeve od 50 do -50, u koracima koji se smanjuju za po 10:

```
for(int i=50; i>=50; i-=10)
    cout << i << endl;
```

Važno je zapamtiti da se uslovni izraz `for` petlje uvek proverava na početku. To znači da se telo `for` petlje neće izvršiti nijednom ako je vrednost kontrolnog izraza na početku `false`. Evo i primera:

```
for(int brojac=10; brojac < 5; brojac++)
    cout << brojac; //ova naredba nikad se neće izvršiti
```

Ova petlja se neće izvršiti nijednom jer je vrednost njene kontrolne promenljive `brojac` veća od 5 kada se prvi put uđe u petlju.

Naredba `for` je jedna od najkorisnijih u jeziku C++ jer ima mnogo varijacija. Na primer, dozvoljava korišćenje nekoliko kontrolnih promenljivih:

```
int x, y;
for(x=0, y=10; x <=y; ++x, --y)
    cout << x << ' ' << y << endl;
```

U ovom primeru zarezi razdvajaju dve naredbe za inicijalizaciju i dva izraza za inkrementiranje; to je neophodno da bi kompajler razumeo da postoje dve naredbe inicijalizacije i dve naredbe inkrementiranja. Takođe, za razliku od većine drugih programskih jezika, C++ omogućuje da delovi petlje `for` budu izostavljeni, kao u sledećem primeru:

```
//for petlja bez inkrementa
for(int i=0; i != 123 ; ) {
    cout << "unesite ceo broj: ";
    cin >> i;
}
```

U ovom primeru deo za inkrementiranje je prazan. Kad god se izvrši petlja, proverava se da li kontrolna promenljiva ima vrednost 123, i ništa više. Petlja se završava kada korisnik programa preko tastature unese 123.

Još jedna moguća varijacija `for` petlje jeste da se deo inicijalizacije izmesti izvan petlje, kao u sledećem primeru:

```
//for petlja koja ispisuje brojeve od 1 do 10
int i = 1; //i je inicijalizovan van for petlje
for( ; i <=10; ) {
```

```

    cout << i << ' ';
    i++;
}

```

Ovde je deo za inicijalizaciju prazan, a promenljiva *x* je inicijalizovana pre ulaska u petlju. Izmeštanje inicijalizacije izvan petlje obično se radi kada se početna vrednost dobija složenim postupkom koji nema veze sa telom *for* petlje. Primetite takođe da je u ovom primeru deo inkrementiranja kontrolne promenljive smešten unutar tela *for* petlje.

Moguće je napisati i *for* petlju koja nema telo, kao u sledećem primeru:

```

//for petlja koja sabira brojeve od 1 do 10
int j=0;
for(int i=0; i < 10; j+=++i);
cout << j << endl; //ispisuje 55

```

Beskonačna petlja (tj. petlja koja se nikada ne završava) u jeziku C++ piše se ovako:

```

for ( ; ; ) {
    //beskonacna petlja
}

```

Iako postoje određeni poslovi koji ne treba nikada da se završe (npr. izvršavanje komandnog procesora u operativnim sistemima), većina „beskonačnih petlji“ zapravo su obične petlje sa specijalnim uslovom završetka. Za izlazak iz „beskonačne“ petlje koristi se naredba *break* o kojoj će biti reči u nastavku ovog poglavlja.

3.5. Petlja while

Druga vrsta petlje podržana u jeziku C++ je *while*, čiji je opšti oblik

```
while(izraz) blok naredbi;
```

Izraz definiše uslov koji kontroliše petlju i može da bude bilo koji validan izraz. Petlja se izvršava sve dok izraz ima vrednost *true*; kada postane *false*, izvršavanje programa nastavlja se u redu odmah iza petlje. Petlja *while* je najkorisnija za ponavljanje delova kôda čiji broj ponavljanja nije unapred poznat. Kao primer, proučimo program koji računa faktorijel broja koji se unosi preko tastature. Zanimljivo je da nema suštinske razlike između petlji *for* i *while*, jer se svaka *while* petlja uz malo prilagođavanje može napisati kao petlja *for*:

```

//izracunavanje faktorijela
#include <iostream>
#include <cmath>

```

```
using namespace std;

int main() {
    int broj;
    int faktorijel=1;
    int i=2;
    cout << "Unesite ceo broj: ";
    cin >> broj;
    while(i<=broj) {
        faktorijel *=i;
        i++;
    }
    cout << "Faktorijel broja " << broj << " je: " <<
        faktorijel << endl;
    return 0;
}
```

Rezultat izvršavanja:

Unesite ceo broj: 5
Faktorijel broja 5 je 120

3.6. Petlja do-while

Često je potrebno da se određena operacija obavi, pa da se u zavisnosti od njenog rezultata odlučuje da li će se ona ponavljati ili ne. Za tu svrhu služi petlja do-while, čija je sintaksa:

```
do
    blok naredbi
while(izraz);
```

Sledeći primer koristi do-while petlju za izvršavanje sve dok korisnik ne unese broj 15:

```
//primer petlje do while
#include <iostream>

using namespace std;
```

```
int main() {  
    const int TAJNI_BROJ = 15;  
    int broj = 0;  
    do {  
        cout << "Unesite tajni broj: ";  
        cin >> broj;  
    } while (broj != TAJNI_BROJ);  
    cout << "Pogodili ste tajni broj!" << endl;  
    return 0;  
}
```

Rezultat izvršavanja:

Unesite tajni broj: 6
Unesite tajni broj: 20
Unesite tajni broj: 15
Pogodili ste tajni broj!

Treba zapamtiti suštinsku razliku između petlji `while` i `do-while`, a to je da se petlja `do-while` sigurno izvršava barem jednom, jer se uslov petlje proverava na dnu, za razliku od petlje `while` kod koje se uslov proverava na vrhu pa može da se desi da se uopšte ne izvrši.

3.7. Naredbe `break` i `continue`

Naredbu `break` smo već susreli u grananju `switch`, ali se ona može koristiti i za prekid izvršavanja petlji `for`, `while` i `do while`. Ova naredba prouzrokuje prekid izvršavanja petlje i skok na prvu naredbu iza petlje. U sledećem primeru `for` petlja se prekida kada vrednost kontrolne promenljive `x` postane 3:

```
for(int x=10; x > 0; x--) {  
    if(x == 3)  
        cout << "odbrojavanje završeno!";  
    break;  
}
```

Kada su petlje ugnježdene (odnosno, kada se jedna petlja nalazi unutar druge), `break` prekida samo unutrašnju petlju. Naredba `break` je najkorisnija kada postoji neki specijalan uslov koji trenutno prekida ponavljanje naredbi, a najčešće se koristi za izlazak iz "beskonačnih" petlji.

Moguće je prekinuti i samo jedan ciklus (iteraciju) petlje, što se postiže pomoću naredbe `continue`. Ova naredba preskače kompletan kôd do kraja tekuće iteracije

petlje i prenosi izvršavanje na proveru uslova koji kontroliše petlju. Na primer, sledeći kôd ispisuje parne brojeve između 1 i 100:

```
for(int x=0; x <= 100; x++) {  
    if(x%2) //preskace neparne brojeve  
        continue;  
    else  
        cout << x << endl;  
}
```

U petljama `while` i `do while`, naredba `continue` prenosi kontrolu direktno na proveru kontrolne promenljive, a zatim se izvršavanje petlje nastavlja.

Naredbe `break` i `continue` u programima treba koristiti samo kada su zaista neophodne, jer narušavaju strukturiranost programa.

3.8. Ugnježdene petlje

Petlje se mogu nalaziti jedna unutar druge, što se veoma često koristi u programiranju. Dobar primer za primenu ugnježđenih petlji je digitalni sat, sa poljima koja prikazuju sate, minute i sekunde. Vrednost za sate promeni se jednom za vreme dok se vrednost za minute promeni 60 puta, a za isto vreme se vrednost za sekunde promeni 3600 puta. U nastavku je dat primer ugnježđenih `for` petlji za digitalni sat:

```
//primer ugnjezdjenih for petlji  
#include <iostream>  
  
using namespace std;  
  
int main() {  
    for(int sat=0; sat<24; sat++) {  
        for(int minut=0; minut<60; minut++) {  
            for(int sekund=0; sekund<60; sekund++) {  
  
                cout << sat << ":" << minut << ":" << sekund << endl;  
            }  
        }  
    }  
    return 0;  
}
```

Rezultat izvršavanja:

```
0:0:0  
0:0:1  
0:0:2  
...
```

23:59:58

23:59:59

3.9. Naredba bezuslovnog skoka (goto)

Naredba `goto` u jeziku C++ služi za bezuslovni skok na lokaciju koja je navedena u nastavku iza ključne reči `goto`. Iako je naredba `goto` gotovo sasvim proterana iz upotrebe zbog toga što čini kôd nečitljivim, a nije ni neophodna jer se uvek može pronaći elegantnije rešenje, ipak se ponekad koristi za kontrolu programa. Da bi se koristila, zahteva da se naredba na koju se “skače” označi identifikatorom iza koga sledi dvotačka, kao u sledećem primeru:

```
if(a ==0) goto deljenjeNulom;
//naredbe
deljenjeNulom:
    cout << "deljenje nulom nije dozvoljeno!";
```

Pitanja

1. Blok koda odvaja se zagradama:

- a) ()
- b) []
- c) { }

2. Doseg (scope) promenljive ograničen je na blok u kome je definisana.

- a) tačno
- b) pogrešno

3. Promenljiva `x` nakon naredbe `x = (10 > 11) ? 10 : 11;` ima vrednost:

- a) 10
- b) 11
- c) 0

4. Delovi petlje `for` mogu biti prazni.

- a) tačno
- b) pogrešno

5. Petlja `for(i=0;i<=20;i++)` ima:

- a) 20 ciklusa
- b) 21 ciklus
- c) 19 ciklusa

6. Petlja `for(i=1;i<=20;i++)` ima:

- a) 20 ciklusa
- b) 21 ciklus
- c) 19 ciklusa

7. Šta radi sledeći deo kôda: `for( ; ; )`

- a) pogrešan kôd
- b) beskonačna petlja

8. Koliko je `ch` u sledećem primeru?

```
int i=7;  
char ch;  
ch =(i==3) ? 'A' : 'B';
```

- a) 'B'
- b) 'A'
- c) 7

9. Izraz koji kontroliše grananje tipa `switch` mora da bude logičkog tipa.

- a) tačno
- b) pogrešno

10. Naredba u jeziku C++ koja služi za bezuslovan skok je:

- a) `break`
- b) `continue`
- c) `goto`

11. Telo petlje `while` može biti prazno.

- a) tačno
- b) pogrešno

12. Telo petlje `do while` izvršava se barem jednom.

- a) tačno
- b) pogrešno

13. U telu petlje mogu se deklarirati promenljive.

- a) tačno
- b) pogrešno

14. U ugnježdjenoj petlji, naredba `break` prekida samo petlju u kojoj se nalazi.

- a) tačno
- b) pogrešno

15. Jedno od ograničenja petlje `for` jeste da se u delu za inicijalizaciju može inicijalizovati samo jedna promenljiva.

- a) tačno
- b) pogrešno

16. Naredba `continue` služi za prelazak na sledeći ciklus petlje.

- a) tačno
- b) pogrešno

17. Koju vrednost ima promenljiva `y` posle izvršavanja sledećeg kôda:

```
int x=3, y=3;
switch(x+3) {
    case(6):
        y =0;
    case(7):
        y =1;
    default:
        y++;
}
```

- a) 1
- b) 2
- c) 3
- d) 4

18. Šta se prikazuje na ekranu kao rezultat izvršavanja ovog kôda:

```
char ch='a';
switch(ch) {
    case 'a':
    case 'A':
        cout<<ch;
        break;
    case 'b':
    case 'B':
```

```
        cout<<ch;
        break;
    case 'c':
    case 'C':
        cout<<ch;
        break;
    case 'd':
    case 'D':
        cout<<ch;
}

```

- a) a
- b) A
- c) Aa

19. Sledeći deo kôda:

```
int x;
double d =1.5;
switch(d) {
    case 1.0:
        x =1;
    case 1.5:
        x=2;
    case 2.0:
        x=3;
}

```

je

- a) pogrešan, jer nedostaju naredbe break
- b) pogrešan, jer nedostaje naredba default
- c) pogrešan jer kontrolna promenljiva d u naredbi switch ne može biti tipa double
- d) ispravan

20. Šta se prikazuje na ekranu kao rezultat sledećeg kôda:

```
int k=20;
while (k>0)
    cout <<k;

```

- a) kôd je pogrešan i neće se izvršavati
- b) neće se prikazati ništa
- c) stalno će se prikazivati 20 u beskonačnoj petlji

21. Šta se prikazuje na ekranu kao rezultat izvršavanja sledećeg kôda:

```
int i=1;
do {
    i++; } while(i<5);
cout<<"i= "<< i;
```

a) ništa se neće prikazati

b) stalno će se prikazivati i=1 u beskonačnoj petlji

c) i=5

22. Šta se prikazuje na ekranu kao rezultat izvršavanja sledećeg kôda:

```
int zbir = 0;
for(int i=0;i<10;i++)
    zbir +=i;
cout <<zbir;
```

a) 10

b) 11

c) 45

23. Da li će izvršavanje sledećeg dela kôda biti beskonačno?

```
int stanje = 10;
while(true) {
    if(stanje<9)
        break;
    stanje = stanje - 9;
}
```

a) da

b) ne

Glava 4

Nizovi, stringovi i pokazivači

U ovom poglavlju bavićemo se nizovima, stringovima i pokazivačima. Iako na prvi pogled izgleda da su to tri nepovezane teme, nije tako. U jeziku C++ nizovi, stringovi i pokazivači su toliko povezani da razumevanje jednog pojma veoma olakšava rad sa drugima.

Niz (array) je skup promenljivih istog tipa kojima se pristupa preko zajedničkog imena. Nizovi mogu imati samo jednu ili više dimenzija, mada se najčešće koriste jednodimenzionalni nizovi. Nizovi su pogodan način za pravljenje liste povezanih promenljivih, a najčešće se koriste nizovi znakova.

U jeziku C++ ne postoji ugrađen tip string, već su stringovi implementirani kao nizovi znakova. Ovakav pristup nudi veću slobodu i prilagodljivost u odnosu na jezike u koje je ugrađen tip string.

Pokazivač je promenljiva koja sadrži neku memorijsku adresu. Pokazivači se najčešće koriste za pristupanje drugim objektima u memoriji, a često su ti objekti nizovi. U stvari, pokazivači i nizovi su u mnogo čvršćoj vezi nego što se na prvi pogled čini, o čemu će biti reči u nastavku poglavlja.

4.1. Jednodimenzionalni nizovi

Jednodimenzionalni niz je skup povezanih promenljivih istog tipa. Nizovi se često koriste u programiranju, a u jeziku C++ opšti oblik deklaracije niza je

```
tip ime_niza [dimenzija];
```

Tip deklarise osnovni tip elemenata koji sačinjavaju niz. Dimenzija određuje broj elemenata koje niz može da sadrži. Na primer, sledeća naredba deklarise niz koji sadrži deset celih brojeva:

```
int primer[10];
```

Pojedinačnim elementima niza pristupa se preko indeksa, koji određuje poziciju elementa u nizu. U jeziku C++, indeks prvog elementa niza je 0. Pošto niz iz primera ima deset elemenata, vrednosti njihovih indeksa su od 0 do 9. Elementu niza pristupa se preko njegovog indeksa koji se navodi u uglastim zagradama. Tako je prvi element niza `primer[0]`, a poslednji je `primer[9]`.

Nizovi se u programiranju često koriste zato što olakšavaju rad sa povezanim promenljivama. Na primer, sledeći program pravi niz od deset elemenata, svakom od njih dodeljuje vrednost, a zatim prikazuje sadržaj elemenata niza:

```
#include <iostream>
using namespace std;
int main() {
    int i;
    int primer[10]; //rezervise prostor za 10 celih brojeva
    //inicijalizacija elemenata niza
    for(i=0; i<10;i++)
        primer[i] = i;
    //prikaz elemenata niza
    for(i=0; i<10;i++)
        cout << "primer[" << i << "] = " << primer[i] << endl;
    return 0;
}
```

Rezultat izvršavanja:

```
primer[0] = 0;
primer[1] = 1;
primer[2] = 2;
...
primer[9] = 9;
```

U jeziku C++ elementi niza smeštaju se u susedne memorijske lokacije. Prvi element nalazi se na najnižoj adresi, a poslednji na najvišoj.

Direktna dodela jednog niza drugom nije dozvoljena, već se vrednosti moraju dodeljivati element po element:

```
int a[10], b[10];
a = b; //greska!
for(int i=0; i<10; i++)
    a[i]=b[i]; //ispravno kopiranje niza
```

Jezik C++ ne proverava granice niza (bounds checking), što znači da programera ništa ne sprečava da "prekorači" kraj niza. Drugim rečima, ako je indeks poslednjeg elementa niza N, može da se indeksira element sa indeksom većim od N a da to ne izazove nikakve greške tokom kompajliranja ili izvršavanja programa. Na primer, kompajler će prevesti i izvršiti sledeći kôd a da se ni najmanje ne pobuni:

```
int krah[10], i;
for(i=0; i<100; i++) krah[i]=i;
```

U ovom slučaju petlja će se ponoviti 100 puta, iako niz `krah` ima samo deset elemenata. To će prouzrokovati menjanje memorijskih lokacija koje nisu dodeljene nizu `krah`. Prekoračivanje granica niza može da prouzrokuje katastrofalne posledice. Ako se prekoračenje granice desi u operaciji dodele, menja se memorija u kojoj se mogu naći neke druge promenljive, a ako se desi tokom čitanja, program će raditi sa neispravnim podacima. Međutim, kada bi proverila granica niza bila ugrađena u jezik (kao npr. u Javi), C++ kôd ne bi bio toliko brz i efikasan. Od programera se očekuje da bude oprezan, tj. da deklarira niz tako da sadrži potreban broj elemenata i da ne dozvoli prekoračivanje granice niza.

4.2. Višedimenzionalni nizovi

C++ podržava i višedimenzionalne nizove, a najjednostavniji oblik je dvodimenzionalni niz. To je zapravo niz jednodimenzionalnih nizova. Na primer, dvodimenzionalni celobrojni niz `dvaD` dimenzija 10 i 20 deklarira se kao:

```
int dvaD[10][20];
```

Za razliku od nekih drugih programskih jezika koji koriste zarez za razdvajanje dimenzija niza, u jeziku C++ svaka dimenzija stavlja se u sopstvene uglaste zagrade. Slično, elementima niza se pristupa tako što se indeksi navode u sopstvenim uglastim zagradama (npr. `dvaD[3][5]`). U sledećem primeru kreira se i inicijalizuje dvodimenzionalni niz brojevi:

```
const int BROJ_REDOVA = 3;
const int BROJ_KOLONA = 4;
//deklaracija niza sa 3 reda i 4 kolone
int brojevi[BROJ_REDOVA][BROJ_KOLONA];
for(int red=0; red<BROJ_REDOVA; red++) {
    for(int kolona=0; kolona<BROJ_KOLONA; kolona++) {
        brojevi[red][kolona] = 4*red + kolona + 1;
    }
}
```

U ovom primeru, `brojevi[0][0]` imaće vrednost 1, `brojevi[0][1]` vrednost 2, `brojevi[0][2]` vrednost 3 itd. Vrednost elementa `brojevi[2][3]` biće 12.

Dvodimenzionalni nizovi čuvaju se u matrici red-kolona, u kojoj prvi indeks označava red, a drugi kolonu. To znači da kada se elementima dvodimenzionalnog niza pristupa na način na koji se stvarno čuvaju u memoriji, desni indeks se menja brže od levog.

	0	1	2	3	levi indeks (red)
desni indeks (kolona)	0	1	2	3	4
	1	5	6	7	8
	2	9	10	11	12

Treba imati na umu da se ukupna memorija potrebna za sve elemente niza određuje tokom prevođenja, i da je ta memorija rezervisana za niz dogod on postoji. U slučaju dvodimenzionalnog niza, u memoriji se odvaja sledeći broj bajtova:

broj redova $\times$ broj kolona $\times$ sizeof(tip elemenata niza).

Recimo, za najčešći slučaj računara na kome tip `int` zauzima 4 bajta, celobrojni niz sa 10 redova i pet kolona zauzimao bi 200 bajtova.

Nizovi mogu imati i više od dve dimenzije, i tada se deklariraju kao:

```
tip ime_niza [dimenzija1][dimenzija2]...[dimenzijaN];
```

Međutim, nizovi sa više od tri dimenzije izuzetno se retko koriste u praksi, jer zauzimaju mnogo memorije (na primer, četvorodimenzionalni niz znakova dimenzija $100 \times 60 \times 90 \times 40$ zauzimao bi preko 20 MB u memoriji), pa bi “pojeo” slobodnu memoriju za ostatak programa.

4.3. Stringovi

Najčešće se jednodimenzionalni nizovi koriste za čuvanje znakova. Jezik C++ podržava dve vrste niza znakova (stringova). Prvi je niz znakova koji se završava nulom (null-terminated string), nasleđen iz jezika C. On sadrži znakove koji sačinjavaju string, i na kraju null (odnosno nulu, tj. znak `\0`). Ova vrsta stringova se često koristi zato što je veoma efikasna i programeru omogućuje detaljnu kontrolu nad operacijama sa stringovima. Druga vrsta je klasa `string` koja se nalazi u biblioteci klase jezika C++. Treba zapamtiti da u jeziku C++ string nije osnovni (ugrađen tip). U ovom poglavlju pozabavićemo se stringovima koji se završavaju nulom (C stringovima).

Kada se deklariraju niz znakova koji se završava nulom, on treba da se deklariraju tako da bude za jedan duži od broja znakova u stringu koji treba da sadrži. Na primer, ako želimo da deklariramo niz `str` u koji treba da stane string od deset znakova, deklariramo ga kao:

```
char str[11];
```

Zadavanje dimenzije 11 rezerviše prostor za nulu na kraju niza znakova. U Poglavlju 2 pominjali smo da C++ dozvoljava definisanje string konstanti, tj. niza znakova koji se pišu pod navodnicima (npr. `"cao"`, `"Ja volim C++"`, `"Tri praseta"`, `""`). Na kraj string konstanti nije neophodno “ručno” dodavati nulu, jer C++ kompajler to radi automatski. Tako će string “Tri praseta” u memoriji izgledati ovako:

T	r	i		p	r	a	s	e	t	a	\0
---	---	---	--	---	---	---	---	---	---	---	----

String `""` se zove null string, jer sadrži samo nulu, a koristan je za predstavljanje praznog stringa.

4.3.1. Učitavanje stringa sa tastature

Najlakši način za čitanje stringa unetog preko tastature jeste korišćenje niza znakova u naredbi `cin`, kao u sledećem primeru:

```
#include <iostream>

using namespace std;

int main() {
    char str[80];
    cout << "Unesite string: ";
    cin >> str;
    cout << "Uneli ste: " << str << endl;
    return 0;
}
```

Rezultat izvršavanja:

Unesite string: Ovo je string
Uneli ste: Ovo

Kada program ponovo prikaže učitani string, prikazaće samo reč "Ovo", a ne celu rečenicu, zato što U/I sistem prestaje sa učitavanjem stringa kada naiđe na prvu belinu (whitespace). Pod belinama podrazumevamo razmake, tabulatore i znakove za nov red. Taj problem rešićemo upotrebom funkcije `gets()` koja je takođe deo standardne C++ biblioteke. Da bismo učitali string, pozvaćemo funkciju `gets()` kojoj ćemo kao argument proslediti ime niza bez indeksa, a kao rezultat taj niz će sadržati string učitani sa tastature. Funkcija `gets()` nastaviće da učitava znake (uključujući i beline) sve dok se ne unese carriage return (znak za nov red). Funkcija `gets()` nalazi se u zaglavlju `<cstdio>` pa se ono mora navesti:

```
#include <iostream>

using namespace std;

int main() {
    char str[80];
    cout << "Unesite string: ";
    gets(str);
    cout << "Uneli ste: " << str << endl;
    return 0;
}
```

Uočite da se u naredbi `cout` može direktno koristiti `str`. U opštem slučaju, ime niza znakova koji sadrži string može se koristiti svuda gde može da se koristi string konstanta. Takođe, zapamtite da ni `cin` ni `gets()` ne proveravaju granice niza koji prihvata unos sa tastature. Ako korisnik unese string koji je duži od dimenzije niza, memorija će biti izmenjena. Kasnije ćemo upoznati alternativu za `gets()` kojom se izbegava taj problem.

4.3.2. Funkcije C++ biblioteke za stringove

U C++ je ugrađen veliki broj funkcija za rad sa stringovima, a najčešće korišćene su prikazane u Tabeli 4.1:

Tabela 4.1: Najčešće korišćene funkcije za rad sa stringovima.	
Funkcija	Opis
<code>strcpy(s1, s2)</code>	Kopira string <code>s2</code> u string <code>s1</code> . Niz <code>s1</code> mora da bude dovoljno dugačak, jer će se u suprotnom biti prekoračena granica niza.
<code>strcat(s1, s2)</code>	Dodaje <code>s2</code> na kraj <code>s1</code> ; <code>s2</code> ostaje neizmenjen. <code>s1</code> mora da bude dovoljno dugačak da u njega stane sadrži prvobitni sadržaj i ceo niz <code>s2</code> .
<code>strcmp(s1, s2)</code>	Poredi stringove <code>s1</code> i <code>s2</code> ; ako su jednaki, vraća 0, ako je <code>s1 > s2</code> (po leksikografskom redu) vraća 1, u suprotnom vraća negativan broj.
<code>strlen(s)</code>	Vraća dužinu stringa <code>s</code> .

Standardna C++ biblioteka nudi i ugrađene funkcije za rad sa znakovima, definisane u zaglavlju `<cctype>`; Tabela 4.2 prikazuje one koje se najčešće koriste.

Tabela 4.2: Najčešće korišćene funkcije za rad sa znakovima.	
Funkcija	Opis
<code>toupper(c)</code>	prevodi znak u veliko slovo (uppercase)
<code>tolower(c)</code>	prevodi znak u malo slovo (lowercase)
<code>isupper(c)</code>	true ako je znak veliko slovo
<code>islower(c)</code>	false ako je znak veliko slovo
<code>isalnum(c)</code>	true ako je znak alfanumerički
<code>isdigit(c)</code>	true ako je znak cifra
<code>isspace(c)</code>	true ako je znak razmak

Sledeći primer ilustruje korišćenje funkcija `strcpy` i `toupper` za prevođenje stringa u sva velika slova:

```
#include <iostream>
#include <cstring>
#include <cctype>

using namespace std;

int main() {
    char str[80];
    strcpy(str, "Ovo je test!");
    for(int i=0; str[i]; i++)
        str[i] = toupper(str[i]);
    cout << str << endl;
    return 0;
}
```

Rezultat izvršavanja:

OVO JE TEST!

Uočite da je uslov završetka petlje `for` napisan kao `str[i]`; kada se stigne do poslednjeg znaka niza, on će imati vrednost nula, što se automatski tumači kao `false` i izvršavanje petlje se prekida. Nema potrebe da se uslov petlje piše kao `str[i] == 0`.

4.4. Inicijalizacija nizova

Nizovi se mogu inicijalizovati slično drugim promenljivama:

```
tip ime_niza[dimenzija] = {skup_vrednosti};
```

`Skup_vrednosti` je niz vrednosti razdvojenih zarezima istog tipa kao što je tip niza. Prva vrednost dodeljuje se prvom elementu niza, druga drugom i tako redom. U sledećem primeru inicijalizuje se celobrojni niz `kvadratni_metri`:

```
int kvadratni_metri[5] = {30, 50, 70, 80, 100};
```

Ovo znači da će `kvadratni_metri[0]` imati vrednost 30, a `kvadratni_metri[4]` će imati vrednost 100. Nizovi znakova koji sadrže stringove mogu se kraće inicijalizovati na sledeći način:

```
char ime_niza[dimenzija] = "string";
```

Na primer, ovako se niz `str` inicijalizuje stringom "C++":

```
char str[4] = "C++";
```

što je isto kao kada bismo napisali

```
char str[4]={'C', '+', '+', '\0'};
```

Pošto se stringovi moraju završavati nulom, niz znakova koji se deklarira mora da bude dovoljno dugačak da u njega stane i nula na kraju. Zbog toga je dimenzija niza str u ovom primeru 4, iako "C++" ima samo tri znaka. Kada se za inicijalizaciju koriste string konstante, kompajler automatski dodaje završnu nulu.

4.4.1. Inicijalizacija nizova bez dimenzija

Kada se niz istovremeno deklarira i inicijalizuje, moguće je izostaviti dimenziju niza i ostaviti kompajleru da je automatski odredi. Kompajler će odrediti dimenziju prebrojavanjem inicijalizacionih vrednosti i kreiraće niz koji je dovoljno dugačak da te vrednosti u njega stanu. Na primer,

```
int vrednosti[] = {2, 3, 4};
```

pravi niz brojeva sa tri elementa koji sadrže vrednosti 2, 3 i 4. Pošto dimenzija niza nije eksplicitno zadata, niz kao što je vrednosti zove se niz bez dimenzije (un-sized array). Nizovi bez dimenzija su prilično korisni. Na primer, pretpostavimo da koristimo inicijalizaciju niza koji treba da sadrži tabelu sa Web adresama, npr.

```
char e1[15] = "www.google.com";
char e2[15] = "www.amazon.com";
char e3[18] = "www.microsoft.com";
```

Očigledno je da je vrlo zamorno ručno prebrojavati znakove da bi se odredila odgovarajuća dimenzija niza, a velika je i šansa da se pogreši u brojanju. Bolje je prepustiti kompajleru da prebrojava znakove:

```
char e1[] = "www.google.com";
char e2[] = "www.amazon.com";
char e3[] = "www.microsoft.com";
```

Osim što je manje zamorna, inicijalizacija nizova bez dimenzija omogućuje da se bilo koji string izmeni, a da se usput ne zaboravi na promenu dimenzije niza. Inicijalizacije nizova bez dimenzija nisu ograničene samo na jednodimenzionalne nizove. U slučaju višedimenzionalnih nizova, dozvoljeno je izostaviti (samo) prvu dimenziju. Na taj način mogu se napraviti tabele promenljivih dužina, tako da kompajler automatski odvoji dovoljno mesta u memoriji za njih. Na primer, ovde je niz kvadrati deklarisan kao niz bez dimenzije:

```
//niz koji sadrzi brojeve i njihove kvadrate
int kvadrati[][2] = { 2, 4
                     3, 9,
                     4, 16,
                     5, 25 }
```

Prednost ovakve deklaracije je u tome što se tabela može skraćivati ili produžavati bez menjanja dimenzija niza.

4.5. Nizovi stringova

Specijalan oblik dvodimenzionalnog niza je *niz stringova*. Takav niz se veoma često koristi u programiranju (npr. modul baza podataka koji tumači ulazne komande upoređuje ih sa nizom stringova koji sadrži važeće komande).

Da bi se napravio niz stringova u jeziku C++ koristi se dvodimenzionalni niz znakova, u kome dimenzija levog indeksa određuje broj stringova, a dimenzija desnog indeksa zadaje maksimalnu dužinu stringa koju može da sadrži (uključujući i nulu na kraju). Na primer, sledeća naredba deklarise i inicijalizuje niz od 7 stringova, od kojih svaki može da sadrži maksimalno 10 znakova (sa nulom na kraju kao jedanaestim znakom).

```
char daniUNedelji[7][11] = { "Ponedeljak", "Utorak", "Sreda",
    "Cetvrtak", "Petak", "Subota", "Nedelja" };
```

Pojedinačnim stringovima unutar niza pristupa se navođenjem samo levog indeksa:

```
daniUNedelji[3] //cetvrtak
```

Da bi se pristupilo pojedinačnim znakovima unutar trećeg stringa, koristi se sledeća sintaksa:

```
daniUNedelji[3][2] //slovo t u stringu cetvrtak
```

Sledeći program ilustruje kako se niz stringova može iskoristiti za pravljenje vrlo jednostavnog telefonskog imenika. Dvodimenzionalni niz sadrži parove imena i brojeva; da bi se prikazao broj, potrebno je uneti ime.

```
//jednostavan telefonski imenik
#include <iostream>

using namespace std;

int main() {
    char str[80];
    int i;
    //definicija niza od 10 stringova
    //u svaki moze da stane 79 znakova
    char brojevi[10][80] = {
        "Pera", "065-2341123",
        "Deki", "063-5654667",
        "Sandra", "062-32563434",
        "Laza", "061-3453453"
    };
    cout << "Unesite ime:";
    cin >> str;
    for(i=0;i<10;i+=2) {
        if(!strcmp(brojevi[i], str)) {
```

```

        cout << "Broj je: " << brojevi[i+1] << endl;
        break;
    }
}
if(i==10)
    cout << "Broj nije pronadjen." << endl;
return 0;
}

```

Rezultat izvršavanja:

Unesite ime: Deki
Broj je: 063-1123400
Unesite ime: Filip
Broj nije pronadjen.

Obratite pažnju na uslov poredjenja u for petlji: `if(!strcmp(brojevi[i], str))`. Funkcija `strcmp()` poredi dva znakovna niza, i ako ustanovi da su jednaki, vraća nulu. Zbog toga se u uslovu poredjenja mora koristiti operator negacije (!). Ako se ustanovi da su string unesen preko tastature i neki element niza znakova sa parnim indeksom jednaki (parni indeksi zato što se poredi po imenima, a ne po brojevima), onda se dohvata sledeći element sa neparnim indeksom u naredbi `brojevi[i+1]`, a to je broj koji odgovara imenu. Ako se ime ne pronade, indeks niza će biti 10 (obratite pažnju i da je brojač `for` petlje `i` deklarisan van petlje da bi bio dostupan i izvan nje).

4.6. Pokazivači

Pokazivači su jedna od najsnažnijih mogućnosti jezika C/C++, ali njihovo savladavanje je obično problematično, naročito za početnike u programiranju. Iako ih je lako pogrešno upotrebiti, oni su veoma bitni za programiranje u jeziku C++ jer omogućuju dinamičku alokaciju memorije i druge funkcionalnosti koje ćemo detaljno proučiti u nastavku knjige. U ovom poglavlju objasnićemo osnove rada sa pokazivačima.

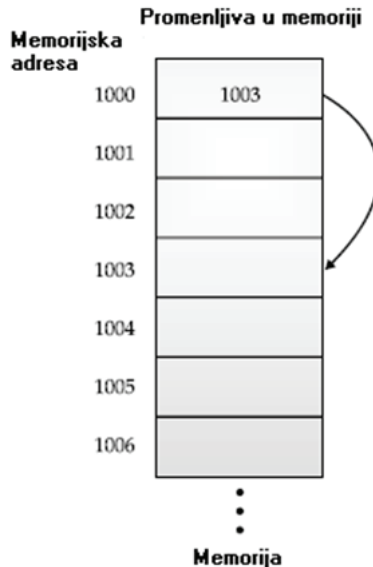
Pokazivač (pointer) je promenljiva koja sadrži memorijsku adresu. Svaka promenljiva ima adresu, a to je broj koji određuje njen položaj u radnoj memoriji računara (Slika 4.1). Najčešće se na adresi koju sadrži pokazivač nalazi neka druga promenljiva, pa se kaže da "pokazivač pokazuje na promenljivu". Na primer, ako `x` sadrži adresu `y`, kaže se da "x pokazuje na y". Pošto memorijska adresa nije običan ceo broj, ni pokazivač ne može biti predstavljen kao obična promenljiva, već se deklarise na poseban način:

```
tip *promenljiva;
```

Ovde je `tip` osnovni tip pokazivača, tj. tip podataka na koji on pokazuje. Na primer, ako želimo da deklariramo promenljivu `ip` kao pokazivač na ceo broj (tip `int`), upotrebićemo deklaraciju:

```
int *ip; //pokazivac na ceo broj
```

Kada se u naredbi deklaracije ispred imena promenljive nalazi zvezdica, ta promenljiva postaje pokazivač.



Slika 4.1: Izgled memorije u računaru.

4.6.1. Pokazivač NULL

Kada se pokazivač deklarise, a pre nego što mu se dodeli adresa na koju će da pokazuje, on će sadržati proizvoljnu vrednost. Ako pokušate da upotrebite pokazivač pre nego što mu se dodeli vrednost, to će vrlo verovatno “srušiti” program jer može doći do nasumičnog prepisivanja sadržaja memorije. Iako ne postoji siguran način za zaobilazanje neinicijalizovanog pokazivača, C++ programeri su usvojili postupak kojim se izbegava većina grešaka vezana za pokazivače koji nisu inicijalizovani. Po tom dogovoru, ako pokazivač sadrži vrednost NULL (nula), onda ne pokazuje ni na šta. Tako, ako se svim pokazivačima koji se trenutno ne koriste dodeli vrednost NULL i u programu se proverava se da li je sadrže, može se izbeći slučajno pogrešno korišćenje neinicijalizovanog pokazivača. To je dobra programerska praksa koje se treba pridržavati. Preprocesorska direktiva u zaglavlju `iostream` definiše vrednost konstante NULL kao 0 i ona se najčešće koristi za inicijalizaciju pokazivača koji ne pokazuju ni na šta. Sledeći kôd proverava da li je vrednost pokazivača NULL:

```
int *pbroj;
//pokazivac koji ne pokazuje ni na sta
pbroj = NULL;
pbroj = 0; //isto sto i pbroj = NULL
```

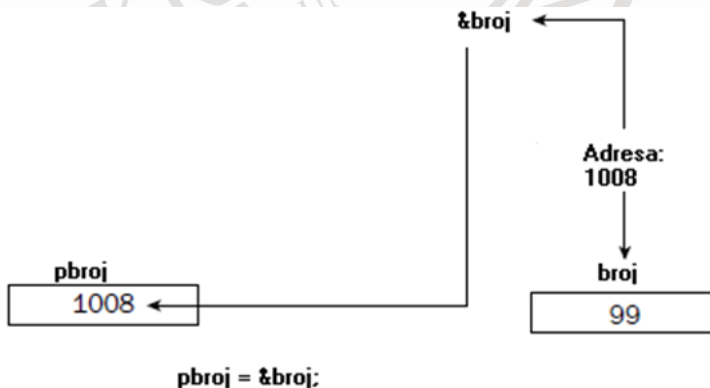
```
//provera da li pokazivac pokazuje na nesto
if(!pbroj)
    cout << "pbroj ne pokazuje ni na sta" << endl;
```

4.6.2. Operacije sa pokazivačima

Postoje dva specijalna operatora koji se koriste sa pokazivačima: * i &. Operator adresiranja (&) je unarni operator koji vraća memorijsku adresu svog operanda. Na primer, naredba `pbroj = &broj;` pokazivačkoj promenljivoj `pbroj` dodeljuje memorijsku adresu promenljive `broj`, odnosno adresu lokacije u memoriji na kojoj se nalazi promenljiva `broj`. Dakle, operator & daje adresu promenljive ispred koje se nalazi.

```
int broj;
int *pbroj;
//vrednost pokazivaca je adresa promenljive
pbroj = &broj;
```

Da bismo bolje razumeli ovu dodelu, pretpostavimo da se promenljiva `broj` nalazi na adresi 1008 i da ima vrednost 99; nakon dodele pomoću operatora & iz prethodnog primera, promenljiva `pbroj` imaće vrednost 1008 (tj. imaće vrednost adrese u memoriji na kojoj se nalazi promenljiva `broj`), kao što je ilustrovano na Slici 4.2.



Slika 4.2: Simbolički prikaz operatora adresiranja.

Operator indirekcije (*) je takođe unarni, a daje vrednost promenljive koja se nalazi na adresi koju sadrži operand. Ovaj operator se zove i operator dereferenciranja jer vraća vrednost promenljive na koju pokazuje pokazivač.

```
int broj = 99;
int vrednost;
int *pbroj;
pbroj = &broj; //pbroj pokazuje na broj
vrednost = *pbroj; //vrednost je 99
```

Ako promenljiva broj ima vrednost 99 kao u prethodnom primeru, onda se pomoću operatora * iz memorije dohvata vrednost na koju pokazuje pbroj, a to je u našem primeru 99.

Oba pokazivačka operatora (& i *) su višeg prioriteta od svih aritmetičkih operatora osim unarnog minusa. Korišćenje pokazivača često se zove i indirekcija jer se promenljivoj pristupa indirektno, tj. preko pokazivača.

```
int *pbroj = NULL;
int broj1 = 55, broj2 = 99;
pbroj = &broj1; //pbroj sada pokazuje na broj1
(*pbroj)+=11; //broj 1 je sada 66
pbroj = &broj2; //pbroj sada pokazuje na broj2
broj1 = (*pbroj) * 10; //broj1 postaje 990
```

4.6.3. Osnovni tip pokazivača

U prethodnom primeru videli smo da je promenljivoj moguće dodeliti vrednost indirektno, preko pokazivača. Međutim, kako C++ zna koliko bajtova treba da kopira sa adrese na koju pokazuje pokazivač? Osnovni tip pokazivača određuje tip podataka sa kojima on može da radi. U prethodnim primerima koristili smo pokazivače na cele brojeve (tip int*). U tom slučaju, ako pretpostavimo da tip int zauzima 4 bajta, kopiraće se 4 bajta podataka počev od adrese na koju pokazuje pokazivač i dodeliće se odgovarajućoj promenljivoj. Kada bismo koristili pokazivače na double (tip double*), kopiralo bi se 8 bajtova, jer je toliko potrebno za tip double.

Važno je obezbediti da pokazivač uvek pokazuje na odgovarajući tip podataka. Ako se pokazivač deklarise kao tip int*, kompajler pretpostavlja da svaki objekat na koji on pokazuje mora biti ceo broj. Ako nije tako, nevolja je na vidiku. Na primer, sledeći kôd je pogrešan:

```
int *p;
double f;
p = &f; //greska!
```

jer se pokazivač na tip double ne može dodeliti pokazivaču na int. Odnosno, &f kreira pokazivač na double, ali je p pokazivač na int. Iako dva pokazivača moraju da pokazuju na isti tip da bi se mogli dodeljivati jedan drugome, to ograničenje se može zaobići konverzijom tipova (iako to skoro nikad nije pametno, jer osnovni tip pokazivača, tj. tip objekta na koji treba da pokazuje određuje kako će se kompajler ponašati prema stvarnim objektima na koje pokazuje).

4.6.4. Dodela vrednosti promenljivoj preko pokazivača

Pokazivač se može koristiti sa leve strane operacije dodele za dodelu vrednosti sa desne strane lokaciji na koju pokazivač pokazuje. Ako je `p` pokazivač na `int`, onda sledeća naredba dodeljuje vrednost 101 lokaciji, tj. promenljivoj na koju pokazuje `p`:

```
*p = 101;
```

Za inkrementiranje ili dekrementiranje vrednosti na koju pokazuje pokazivač, može se upotrebiti naredba

```
(*p)++;
```

Zagrade su neophodne jer je operator `*` nižeg prioriteta od operatora `++`. Sledeći primer ilustruje dodelu vrednosti promenljivoj preko pokazivača.

```
int *pbroj = NULL;
int broj = 10;
pbroj = &broj; //pbroj pokazuje na broj
cout << broj << endl; //ispisuje 10
(*pbroj)++; //broj se inkrementira preko pokazivaca
cout << broj << endl; //ispisuje 11
```

4.6.5. Pokazivačka aritmetika

Pokazivači se mogu koristiti u većini C++ izraza, ali uz poštovanje nekih posebnih pravila. Sa pokazivačima se koriste samo četiri aritmetička operatora: `++`, `--`, `+` i `-`. Rezultat operacija sa pokazivačima direktno zavisi od njegovog osnovnog tipa.

Pretpostavimo da je `p` pokazivač na `int` čija je trenutna vrednost 2000 (tj. sadrži adresu 2000). Pod pretpostavkom da tip `int` zauzima 4 bajta, nakon izraza `p++`; sadržaj pokazivača `p` biće 2004, a ne 2001, zato što kad god se `p` inkrementira, pokazuje na sledeći ceo broj. Isto važi i za dekrementiranje; ako `p` ima vrednost 2000, nakon naredbe `p--`; imaće vrednost 1996.

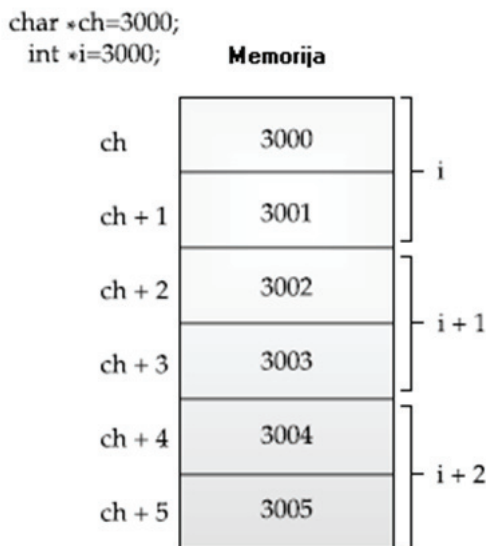
Kad god se pokazivač inkrementira, pokazivaće na lokaciju sledećeg elementa svog osnovnog tipa. Ako se radi o pokazivaču na znakove, operacije inkrementiranja i dekrementiranja izgledaće "uobičajeno" jer tip `char` zauzima jedan bajt (Slika 4.3).

Na pokazivaće se takođe može primenjivati sabiranje i oduzimanje. Na primer, izraz `p = p + 9`; pomera `p` na deveti element njegovog osnovnog tipa u odnosu na onaj na koji trenutno pokazuje.

4.6.6. Pokazivači i nizovi

U jeziku C++ pokazivači i nizovi su blisko povezani, i često se mogu uporedo koristiti. Proučimo sledeći primer:

```
char str[80];
char *p;
p = str;
```



Slika 4.3: Aritmetičke operacije sa pokazivačima zavise od njegovog osnovnog tipa.

Ovde je `str` niz od 80 znakova, a `p` je pokazivač na znakove. U trećem redu, pokazivaču `p` se dodeljuje adresa prvog elementa niza `str` (odnosno, posle ove dodele, `p` će pokazivati na `str[0]`). To je zato što u jeziku C++ ime niza (bez indeksa) zapravo pokazivač na prvi element niza. Treba ipak zapamtiti da je ime niza konstantan pokazivač i nijedna naredba ne može da mu promeni vrednost (npr. ne može se pomeriti inkrementiranjem).

Kad god se u izrazima koristi ime niza bez indeksa, dobija se pokazivač na prvi element niza. Pošto nakon dodele `p = str;` pokazivač `p` pokazuje na početak niza `str`, nadalje `p` može da se koristi za pristup elementima niza `str`. Na primer, ako želimo da pristupimo petom elementu niza `str`, možemo da upotrebimo sintaksu `str[4]` ili `*(p+4)`. Pokazivaču se dodaje 4 jer trenutno pokazuje na prvi element niza `str`.

Zagrade oko izraza `p+4` su neophodne zato što je operator `*` većeg prioriteta od operatora `+`. Kada ih ne bi bilo, prvo bi se pronašla vrednost na koju pokazuje `p` (prvi element niza), a zatim bi se ona sabrala sa 4.

C++ tako nudi dva načina za pristupanje elementima niza: pokazivačku aritmetiku i indeksiranje. Pokazivačka aritmetika je često brža od indeksiranja, naročito ako se elementima niza pristupa redom. Obično je potrebno više mašinskih instrukcija za indeksiranje niza nego za rad preko pokazivača. Zbog toga se u profesionalno napisanom C++ kôdu češće koriste pokazivači. Početnicima je obično lakše da koriste indeksiranje dok u potpunosti ne savladaju rad sa pokazivačima.

Kao što smo upravo videli, nizu je moguće pristupati preko pokazivačke aritmetike. Ali, važi i obrnuto: pokazivač je moguće indeksirati kao da se radi o nizu. Ako je

p pokazivač na prvi element niza znakova, onda je naredba `p[i]` funkcionalno identična sa `*(p+i)`.

Kao primer navodimo program koji mala slova stringa prevodi u velika i obrnuto. U prvom slučaju program je napisan korišćenjem tehnike indeksiranja nizova:

```
#include <iostream>

using namespace std;

int main() {
    char str[] = "Ovo Je Test";
    for(int i=0; str[i]; i++) {
        if(isupper(str[i]))
            str[i] = tolower(str[i]);
        else if(islower(str[i]))
            str[i] = toupper(str[i]);
    }
    cout << str << endl;
    return 0;
}
```

Rezultat izvršavanja:

oVO JE tEST

Program koristi bibliotečke funkcije `isupper` i `islower` da bi odredio da li je znak malo ili veliko slovo. Funkcija `isupper` vraća `true` kada joj je argument veliko slovo, a `islower` vraća `true` kada je argument malo slovo. U `for` petlji `str` se indeksira, proverava se da li je znak malo ili veliko slovo i obrće se. Petlja se završava kada se dođe do kraja stringa, tj. do nule na kraju niza; pošto se 0 tumači kao `false`, petlja se prekida.

Isti ovaj program napisan pomoću pokazivača izgleda ovako:

```
#include <iostream>

using namespace std;

int main() {
    char str[] = "Ovo Je Test";
    char *p = str;
    while(*p) {
        if(isupper(*p))
            *p = tolower(*p);
    }
}
```

```
        else if(islower(*p))
            *p = toupper(*p);
        p++;
    }
    cout << str << endl;
    return 0;
}
```

U ovoj verziji, `p` se postavlja na početak niza `str`. Zatim se unutar petlje `while` slovo na koje pokazuje `p` proverava i menja, a nakon toga se `p` inkrementira. Petlja se završava kada `p` stigne do znaka nula kojim se završava niz `str`.

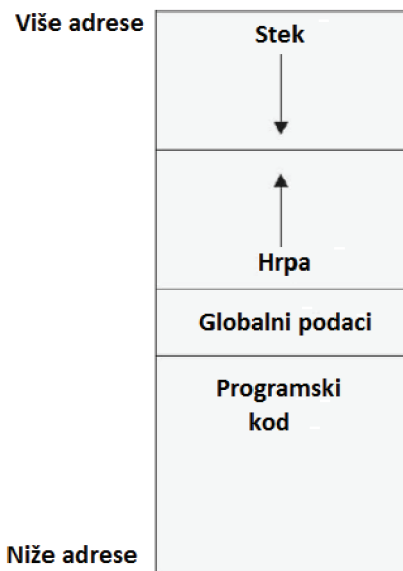
4.7. Dinamička alokacija memorije

Promenljive se mogu kreirati i uništavati i tokom izvršavanja programa. Često se dešava da ne znamo tačno koliko nam je promenljivih potrebno. Pretpostavimo na primer da hoćemo da napišemo program koji računa srednju ocenu proizvoljnog broja testova. Kako ćemo smestiti rezultate pojedinačnih testova u memoriju ako ne znamo tačno koliko ima testova? C++ omogućuje “usputno” definisanje promenljivih, što se zove dinamička alokacija memorije, a moguće je samo uz korišćenje pokazivača.

Dinamička alokacija memorije znači da program tokom izvršavanja traži od računara da odvoji deo neiskorišćene memorije da bi u njega smestio promenljive određenog tipa. Pretpostavimo da nam je potrebna promenljiva tipa `int`. Program će od računara zatražiti da odvoji prostor za ceo broj, a računar će programu vratiti početnu memorijsku adresu gde će broj biti smešten. Memoriji koja je odvojena (alocirana) program može da pristupi samo preko adrese, pa je za njeno korišćenje neophodan pokazivač.

Dinamička alokacija se često koristi za podršku strukturama kao što su povezane liste, stabla i sl. Naravno, dinamičku alokaciju možete da upotrebite kad god vam se učini da bi bila pogodna, i ona se koristi u većini realnih programa. Memorija koja se dinamički alokira uzima se iz oblasti koja se zove hrpa (heap). Hrpa se razlikuje od drugih zona memorije koje koristi C++ program, kao što su memorija za statičke i lokalne promenljive (Slika 4.4). Ako se promenljiva deklarise kao statička ili globalna (deklarisanjem van svih funkcija), prostor za promenljivu se alokira iz zone globalnih podataka, i to pre nego što izvršavanje programa počne. Tek kada se funkcija `main()` završi, računar dealocira prostor za statičke i globalne promenljive. Automatska (lokalna) promenljiva, tj. promenljiva deklarisanu unutar bloka, kreira se u zoni zvanoj stek kada program ulazi u blok, a uništava se kada program napušta blok. Prostor za lokalne promenljive se alokira i dealocira tokom izvršavanja programa, i to se de-

šava automatski (programer nema kontrolu nad alokacijom i dealokacijom lokalnih promenljivih).



Slika 4.4: Simbolički prikaz korišćenja memorije u C++ programima.

Dinamičko alociranje memorije zahteva se operatorom `new`. Operator `new` alocira memoriju i vraća pokazivač na promenljivu. Operator `delete` oslobađa memoriju koja je prethodno alocirana pomoću operatora `new`. Opšta sintaksa ovih operatora je:

```
pokazivac = new tip;
delete pokazivac;
```

`pokazivac` je promenljiva koja prihvata adresu alocirane promenljive. Na primer, ako želimo da alociramo prostor za jednu celobrojnu promenljivu, to ćemo učiniti naredbom:

```
int *p = new int;
```

Dinamičko alociranje pojedinačnih promenljivih nema mnogo smisla, već se ono prvenstveno koristi za dinamičko kreiranje nizova. Niz od 100 celih brojeva može se dinamički alocirati naredbom:

```
int *p = new int[100];
```

Ako nema dovoljno memorije koja bi se alocirala, operator `new` vraća vrednost 0 (`NULL`), pa u programu to treba uvek proveriti:

```
int *p = new int[100];
if(!p) //isto sto i if (p == NULL)
    cout << "greska u alociranju memorije!";
```

Kada program završi sa korišćenjem dinamički alocirane memorije, treba je osloboditi pomoću operatora `delete`. Operand za `delete` mora da bude pokazivač koji je korišćen za alokaciju operatorom `new`.

```
delete p; //ako p pokazuje na promenljivu  
delete [] p; //ako p pokazuje na niz
```

Pitanja

- Indeks prvog elementa niza u jeziku C++ je:
 - 0
 - 1
- Jezik C++ proverava granice niza.
 - tačno
 - pogrešno
- Naredba `int broj[10]={0};`
 - inicijalizuje sve elemente niza na vrednost 0
 - predstavlja prazan niz
 - deklariše samo jedan element u nizu
- Naredba `cout << niz[2];`
 - prikazuje vrednost trećeg elementa niza
 - prikazuje vrednost drugog elementa niza
 - prikazuje vrednost indeksa drugog elementa niza
- Naredba `cin >> niz[1];`
 - prikazuje vrednost prvog elementa niza
 - prikazuje vrednost drugog elementa niza
 - prihvata vrednost za drugi element niza sa standardnog ulaza
- Niz je skup objekata:
 - različitog tipa
 - istog tipa
 - mešovito tipa
- Niz po imenu `brojevi` koji sadrži 31 element tipa `short int` deklariše se kao:
 - `short int brojevi[31];`
 - `brojevi short[31];`
 - `short int brojevi(31);`
 - `short brojevi[31];`

8. Koje je ime trećeg elementa u nizu `a`?

- a) `a(2)`
- b) `a[2]`
- c) `a[3]`
- d) `a(3)`

9. Naredba `for` mora se upotrebiti za inicijalizaciju svih elemenata niza na 0.

- a) tačno
- b) pogrešno

10. Za sabiranje elemenata dvodimenzionalnog niza moraju se upotrebiti ugnježdene `for` petlje.

- a) tačno
- b) pogrešno

11. C-string je niz znakova koji:

- a) počinje od početka reda
- b) završava se znakom `\0`
- c) ima najmanje jedan znak

12. Standardna biblioteka funkcija koje rade sa C stringovima je:

- a) `iostream`
- b) `cstdlib`
- c) `cstring`

13. Funkcija `strcmp`:

- a) komprimuje niz
- b) poredi dva stringa
- c) kompajlira niz

14. Funkcija `tolower(ch)`:

- a) ispisuje znak na ekranu malim fontom
- b) konvertuje znak u malo slovo
- c) konvertuje znak u veliko slovo

15. Funkcija `toupper(ch)`:

- a) ispisuje znak na ekranu većim fontom
- b) konvertuje znak u malo slovo
- c) konvertuje znak u veliko slovo

16. Kada ustanovi da su dva stringa jednaka, funkcija strcmp vraća:

- a) 0
- b) 1
- c) pozitivan broj
- d) negativan broj

17. Dimenzija niza znakova u koji treba da stane string "singidunum" je:

- a) 9
- b) 10
- c) 11

18. Funkcija koja sa tastature učitava niz znakova koji sadrži razmake je:

- a) cin
- b) gets

19. Funkcija koja vraća dužinu stringa je:

- a) strcpy
- b) strlen
- c) strcmp

20. String konstanta se piše pod navodnicima.

- a) tačno
- b) pogrešno

21. Naredba `str[] = "nizZnakova";` je ispravna.

- a) tačno
- b) pogrešno

22. Pokazivač je promenljiva:

- a) čija je vrednost adresa druge promenljive
- b) koja pokazuje na funkciju
- c) koja pokazuje na kraj programa

23. Svakom bajtu u memoriji dodeljuje se jedinstvena adresa.

- a) tačno
- b) pogrešno

24. Ime niza je pokazivač na:

- a) sve članove niza
- b) veličinu niza
- c) prvi element niza

25. Adresa na koju pokazuje ime niza može se promeniti.

- a) tačno
- b) pogrešno

26. Gde je greška u sledećem delu kôda ?

```
int* p = &i;  
int i;
```

- a) u prvom redu
- b) u drugom redu
- c) u redosledu deklarisanja promenljivih

27. Na šta pokazuje pokazivač ako ima vrednost 0?

- a) na nultu memorijsku lokaciju
- b) na prvi član niza
- c) ni na šta

28. Šta radi sledeći deo koda?

```
int a[3]={1,2,3};  
int i;  
for(i=0;i<3;++i)  
    cout<<*(a+i);
```

- a) sabira članove niza
- b) ispisuje vrednosti pokazivača na članove niza
- c) ispisuje članove niza

29. Pokazivač može da pokazuje ni na šta.

- a) tačno
- b) pogrešno

30. Ako želim da pokazivač p pokazuje na promenljivu x, naredba kojom ću to postići je:

- a) p=x;
- b) p=&x;
- c) p=*x;
- d) *p=x;

31. Šta će se ispisati na ekranu posle sledećih naredbi?

```
int *p;  
int x = 100;  
p=&x;  
cout <<(*p);
```

- a) 0
- b) 100
- c) memorijska adresa promenljive x

32. Kolika je vrednost promenljive x nakon sledećih naredbi?

```
int *p, x;  
x=100;  
p = &x;
```

- ```
p++;
```
- a) 100
  - b) 101
  - c) 104

33. Kolika je vrednost promenljive x nakon sledećih naredbi?

```
int *p, x;
x=100;
p = &x;
(*p)++;
```

- a) 100
- b) 101
- c) 104

34. Ako je brojevi ime niza, onda se brojevi [2] preko pokazivača može napisati kao:

- a) \*(brojevi+2)
- b) &(brojevi+2)
- c) brojevi+2

35. Moguće je definisati niz pokazivača.

- a) tačno
- b) pogrešno

36. U radu sa pokazivačima koriste se operatori:

- a) &
- b) .
- c) \*
- d) &&

37. Jedini ceo broj koji se može direktno dodeliti pokazivaču je 0.

- a) tačno
- b) pogrešno

38. Povratna vrednost operatora `new` je:

- a) pokazivač
- b) `void`
- c) `int`
- d) ništa od navedenog

39. Operator `delete` oslobađa memoriju koju je prethodno rezervisao operator `new`.

- a) tačno
- b) pogrešno

40. Nemoguće je kreirati promenljive tokom izvršavanja programa.

- a) tačno
- b) pogrešno



## Glava 5

# Funkcije

*Funkcija* je potprogram koji sadrži nekoliko C++ naredbi i izvodi određeni zadatak. Svi programi koje smo dosad pisali koristili su jednu funkciju `main()`. Funkcije su gradivni blokovi jezika C++ jer je program zapravo skup funkcija, a sve naredbe koje nešto "rade" nalaze se u funkcijama. Dakle, funkcije sadrže naredbe koje su izvršni deo programa. Iako veoma jednostavni programi kao što su primeri u ovoj knjizi imaju samo funkciju `main()`, većina komercijalnih programa ih ima na stotine. Sve C++ funkcije imaju isti oblik:

```
povratni_tip ime (lista parametara) {
 // telo funkcije
}
```

Povratni\_tip zadaje tip podataka koje funkcija vraća, što može da bude bilo koji tip podataka osim niza. Ako funkcija ne vraća nikakvu vrednost, njen povratni tip mora da bude deklarisan kao `void`. Ime funkcije može da bude bilo koji važeći identifikator koji se već ne koristi u programu. Lista parametara je niz parova tip - identifikator razdvojenih zarezima. Parametri su promenljive koje prihvataju vrednosti stvarnih argumenata koji se prosleđuju funkciji prilikom njenog pozivanja. Ako funkcija nema parametre, onda je lista parametara prazna. Telo funkcije okružuju vitičaste zagrade, a čine ga naredbe koje definišu šta funkcija radi. Funkcija se završava i vraća u deo kôda iz koga je pozvana kada izvršavanje stigne do zatvorene zagrade.

Pošto su sve funkcije istog oblika, one liče na funkciju `main()` koju smo već koristili. Sledeći primer sadrži dve funkcije: `main()` i `funkcija()`.

---

```
#include <iostream>

using namespace std;

//prototip funkcije
void funkcija();

int main() {
 cout << "u funkciji main" << endl;
 funkcija(); //poziv funkcije
```

```
 cout << "ponovo u mainu" << endl;
 return 0;
}

void funkcija() {
 cout << "u funkciji" << endl;
}
```

Rezultat izvršavanja:

```
u funkciji main
u funkciji
ponovo u mainu
```

---

Izvršavanje započinje u funkciji `main()`, tj. izvršava se prva naredba `cout`. Zatim funkcija `main()` poziva funkciju `mojaFunkcija()`. Primetite da se funkcija poziva navođenjem zagrada iza imena. U ovom slučaju poziv funkcije je naredba i zato mora da se završi tačkom-zarezom. Zatim se u funkciji `mojaFunkcija()` izvršava naredba `cout` i izvršavanje se vraća u `main()` kada se stigne do zatvorene zagrade `}`. U funkciji `main()` izvršavanje se nastavlja u redu kôda odmah ispod poziva `mojaFunkcija()`. Na kraju `main()` izvršava drugu naredbu `cout` i program se završava.

Način na koji se `mojaFunkcija()` poziva i kako se iz nje vraća je specifičan primer postupka koji se primenjuje na sve funkcije. U opštem slučaju, da bi se funkcija pozvala, navodi se njeno ime iza koje slede zagrade. Nakon pozivanja funkcije izvršavanje "skače" u funkciju i nastavlja se unutar nje. Kada se funkcija izvrši, izvršavanje programa se vraća pozivaocu u naredbi koja se nalazi odmah iza poziva funkcije. Primetite da je u prethodnom programu ispred funkcije `main()` navedena naredba:

```
void mojaFunkcija(); //prototip funkcije
```

Iako ćemo se na prototipe funkcije kasnije vratiti da bismo ih detaljnije proučili, sada treba da znamo da prototip funkcije deklarise funkciju pre njene definicije. Prototip omogućuje kompajleru da pre prvog poziva funkcije zna njen povratni tip, broj i tipove eventualnih parametara. Pošto kompajler mora da zna te informacije pre prvog poziva funkcije, u ovom slučaju prototip mora da se navede pre funkcije `main()`. Ključna reč `void` ispred prototipa funkcije `mojaFunkcija()` i njene definicije znači da `mojaFunkcija()` ne vraća nikakvu vrednost; povratni tip takvih funkcija u jeziku C++ je `void`.

## 5.1. Argumenti funkcije

Funkciji je moguće proslediti jednu ili više vrednosti. Vrednost koja se prosleđuje funkciji zove se *argument*. Kada se kreira funkcija koja prihvata jedan argument ili više njih, moraju se deklarirati i promenljive koje prihvataju te argumente, a one se

zovu *parametri funkcije*. U sledećem primeru definiše se funkcija `kutija()` koja računa zapreminu kutije i prikazuje rezultat, a ima tri parametra.

---

```
//program koji ilustruje pozivanje funkcije
#include <iostream>

using namespace std;

//prototip funkcije
void kutija(int duzina, int sirina, int visina);

int main() {
 kutija(2, 3, 4);
 return 0;
}

void kutija(int duzina, int sirina, int visina) {
 cout << "Zapremina kutije je " <<
 duzina * sirina * visina << endl;
}
```

Rezultat izvršavanja:

**Zapremina kutije je: 24**

---

Kad god se pozove funkcija `kutija()`, ona će izračunati zapreminu množenjem vrednosti koje joj se prosleđuju kao parametri: dužine, širine i visine. Deklaracije parametara međusobno su razdvojene zarezima, a navedene su unutar zagrada iza imena funkcije. Takva sintaksa se koristi za sve funkcije sa parametrima. Da bi se pozvala funkcija `kutija()`, moraju se zadati tri argumenta, na primer `kutija (2, 3, 4);` ili `kutija(8, 6, 9);`

Vrednosti navedene između zagrada su argumenti koji se prosleđuju funkciji `kutija()`, pri čemu se vrednost svakog argumenta kopira redom u odgovarajući parametar. To znači da se u pozivu `kutija(2, 3, 4)`, vrednost 2 kopira u dužinu, 3 u širinu, a 4 u visinu. Kada se argumenti prosleđuju funkciji, prvom parametru se dodeljuje vrednost prvog argumenta itd. (vrednosti se prosleđuju isključivo na osnovu pozicije).

## 5.2. Povratak iz funkcije

U prethodnim primerima funkcija se vraćala na red u kome je pozvana kada je izvršavanje stiglo do zatvorene zagrade u funkciji. Iako je to prihvatljivo za neke funkcije, nije primenljivo na sve. Često je potrebno precizno kontrolisati kada se funkcija

“vraća” i za tu namenu služi naredba `return`. Naredba `return` ima dva oblika: jedan koji vraća vrednost i drugi koji ne vraća ništa. Za početak razmotrimo naredbu `return` koja ne vraća vrednost. Ako je povratni tip funkcije `void` (što znači da ona ne vraća ništa), onda se koristi oblik

```
return;
```

Kada se naiđe na naredbu `return`, izvršavanje funkcije se prekida i vraća na mesto odakle je funkcije pozvana. Sav kôd koji se nalazi iza naredbe `return` se ignoriše, kao u sledećem primeru:

---

```
#include <iostream>

using namespace std;

void funkcija(); //prototip funkcije

int main() {
 cout << "Pre poziva: " << endl;
 funkcija();
 cout << "Posle poziva " << endl;
 return 0;
}

void funkcija() {
 cout << "U funkciji" << endl;
 return;
 cout << "ovo nece biti prikazano";
}
```

Rezultat izvršavanja:

```
pre poziva
u funkciji
posle poziva
```

---

Funkcija može da ima nekoliko naredbi `return`. Ipak, povratak iz funkcije na više mesta može da naruši njenu strukturu i da oteža razumevanje kôda, pa više naredbi `return` u istoj funkciji treba koristiti samo kada olakšavaju njeno razumevanje.

Funkcija može i da vraća vrednost pozivaocu, pa naredba `return` služi i za dobijanje informacija od funkcije. Za vraćanje vrednosti koristi se drugi oblik naredbe `return`:

```
return vrednost;
```

Ovde je vrednost ono što funkcija vraća kao rezultat, a mora biti istog tipa kao što je i tip funkcije. Ako se tip funkcije i tip vrednosti iza naredbe `return` ne poklapaju, pojaviće se greška tokom kompajliranja. Tip funkcije može da bude bilo koji važeći

C++ tip podataka osim niza. U sledećem primeru funkcija `kutija()` je prepravljena tako da vraća zapreminu.

---

```
#include <iostream>

using namespace std;

//prototip funkcije
int kutija(int duzina, int sirina, int visina);

int main() {
 cout << "Zapremina kutije je: " << kutija(2, 3, 4) << endl;
 return 0;
}

int kutija(int duzina, int sirina, int visina) {
 return duzina * sirina * visina;
}
```

Rezultat izvršavanja:

**Zapremina kutije je: 24**

---

Ako se funkcija čiji povratni tip nije `void` vraća zato što je izvršavanje stiglo da zatvorene zagrade, vraća se nedefinisana (nepoznata) vrednost, ali dobra praksa u programiranju nalaže da bilo koja ne-void funkcija treba da vrati vrednost putem eksplicitne naredbe `return`.

### 5.2.1. Funkcija `exit()`

C++ program prestaje da se izvršava kada stigne do kraja funkcije `main()`, ili do naredbe `return` u funkciji `main()`. Kada se druge funkcije izvrše, program se ne završava, već se kontrola izvršavanja vraća na naredbu odmah iza poziva funkcije. Ponekad je poželjno ili čak neophodno da se program završi u nekoj funkciji koja nije `main()`. Da bi se to postiglo, koristi se funkcija `exit()` koja prekida izvršavanje programa, bez obzira na to koja funkcija ili kontrolni mehanizam se izvršavaju. Da bi se koristila funkcija `exit()` neophodno je navesti zaglavlje `cstdlib`.

```
#include <iostream>
#include <cstdlib>

using namespace std;

//prototip funkcije
void funkcija();
```

```

int main() {
 funkcija();
}

void funkcija() {

 cout << "ovaj program završava se pomocu funkcije exit" << endl;
 cout << "cao!" << endl;
 exit(0);
 cout << "ova poruka nikada neće biti prikazana " <<
 "jer je program već završen.";
}

```

Funkcija `exit()` ima jedan celobrojni argument koji se koristi za prosleđivanje izlaznog kôda operativnom sistemu; vrednost 0 standardno označava uspešan završetak. U zaglavlju `cstdlib` definisane su dve konstante, `EXIT_SUCCESS` i `EXIT_FAILURE` koje se mogu koristiti ako se zna da neće biti posebne provere rezultata programa.

### 5.3. Doseg promenljivih (scope)

Dosad smo koristili promenljive bez formalnog objašnjenja gde mogu da se deklariraju, koliko ostaju važeće i koji delovi programa mogu da im pristupaju. Ta svojstva određena su pravilima za *doseg* (scope) promenljivih u jeziku C++, koja upravljaju vidljivošću i životnim vekom promenljivih. Iako je sistem dosega u jeziku C++ prilično složen, izdvajaju se dva osnovna tipa: lokalni i globalni dosegi. U ovom odeljku pozabavićemo se razlikama između promenljivih deklariranih u lokalnom i globalnom dosegu i videćemo u kakvoj su oni vezi sa funkcijama.

#### 5.3.1. Lokalni dosegi

Lokalni opseg važnosti se definiše u bloku (podsetimo se iz Poglavlja 3 da su početak i kraj bloka kôda definisani zagradama). Kad god se započne nov blok, stvara se nova oblast važenja promenljivih (dosegi). Promenljiva koja je definisana unutar bloka zove se lokalna promenljiva. Sve promenljive definisane unutar bloka su lokalne za taj blok, tj. nisu vidljive niti dostupne izvan njega. U suštini, kada se deklarira lokalna promenljiva, ona se “lokalizuje” i sprečava se njeno neovlašćeno korišćenje i menjanje.

Lokalne promenljive “žive” samo dok se izvršava blok koda u kome su definisane. One nastaju unutar bloka i nestaju kada se blok završi. Najčešće korišćen blok koda u kome se definišu lokalne promenljive jeste funkcija: pošto se telo funkcije nalazi između zagrada, funkcija jeste blok, pa njenim naredbama i podacima ne može da se pristupi izvan nje (nije moguće, na primer, upotrebiti naredbu `goto` da bi se “uskočilo” usred neke funkcije).

---

```
#include <iostream>

using namespace std;

//prototip funkcije
void funkcija();

int main() {
 int promenljiva = 10;
 cout << "vrednost promenljive u mainu je: " <<
 promenljiva << endl;
 funkcija();
 cout << "vrednost promenljive u mainu " <<
 "nakon poziva funkcije je: " << promenljiva << endl;
 return 0;
}

void funkcija() {
 int promenljiva = 88;
 cout << "vrednost promenljive u funkciji je: " <<
 promenljiva << endl;
}
```

Rezultat izvršavanja:

**vrednost promenljive u mainu je: 10**

**vrednost promenljive u funkciji je 88**

**vrednost promenljive u mainu nakon poziva funkcije je: 10**

---

Uobičajeno je da se sve promenljive potrebne u funkciji deklariraju na početku tela funkcije, prvenstveno zbog toga što funkcija postaje čitljivija. Međutim, to ne mora obavezno da bude tako: lokalna promenljiva može se deklarirati bilo gde unutar bloka, ali ta deklaracija mora da se nalazi pre njenog prvog korišćenja.

Parametri funkcije su unutar dosega funkcije, tj. oni su lokalni za funkciju i osim što prihvataju vrednosti argumenata, ponašaju se kao i sve druge lokalne promenljive.

Pošto se lokalna promenljiva pravi i uništava pri svakom ulasku i izlasku iz bloka u kome je deklarirana, njena vrednost neće biti zapamćena između dva izvršavanja. Naročito je važno razumeti korišćenje lokalnih promenljivih u funkciji: one se prave pri ulasku u funkciju, a uništavaju po izlasku, što znači da ne mogu da zadrže vrednost između dva poziva funkcije. Ako se lokalna promenljiva inicijalizuje, onda se inicijalizacija ponavlja pri svakom izvršavanju bloka, što ilustruje sledeći primer.

---

```

#include <iostream>

using namespace std;

//prototip funkcije
void funkcija();

int main() {
 for(int i=0; i<3; i++)
 funkcija();
 return 0;
}

void funkcija() {
 //promenljiva se inicijalizuje pri svakom ulasku u funkciju
 int promenljiva = 88;
 cout << "promenljiva: " << promenljiva << endl;
 promenljiva++; //ovo nema nikakvog efekta
}

```

Rezultat izvršavanja:

**promenljiva: 88**  
**promenljiva: 88**  
**promenljiva: 88**

---

### 5.3.2. Globalni doseg

Globalni doseg (global scope) je oblast deklarisanja koja se nalazi izvan svih funkcija. Globalne promenljive dostupne su u celom programu, tj. njihova oblast važnosti je ceo kôd programa, i zadržavaju svoju vrednost tokom celog izvršavanja programa. To znači da je njihova oblast važenja (doseg) ceo program. Globalne promenljive deklariraju se izvan svih funkcija.

Sledeći program ilustruje korišćenje globalne promenljive broj `ac` koja je deklarirana izvan svih funkcija, pre funkcije `main()`. Pošto se svaka promenljiva mora deklarirati pre prvog korišćenja, najbolje je da se globalne promenljive deklariraju na samom početku programa, izvan svih funkcija, pre funkcije `main()`, da bi bile dostupne celom programu.

---

```

#include <iostream>

using namespace std;

void funkcija1();
void funkcija2();

```

```
int brojac; //globalna promenljiva

int main() {
 for(int i=0; i<10; i++) {
 brojac = i * 2; //ovo se odnosi na globalni brojac
 funkcija1();
 }
 return 0;
}

void funkcija1(){
 //pristup globalnom brojacu
 cout << "brojac: " << brojac << endl;
 funkcija2();
}

void funkcija2(){
 //ovaj brojac je lokalni za funkciju2
 int brojac;
 for(brojac=0; brojac<3; brojac++)
 cout << ",";
}
```

Rezultat izvršavanja:

```
brojac: 0
...brojac: 2
...brojac: 4
...brojac: 6
...
...brojac: 18
```

Globalne promenljive inicijalizuju se kada program počne da se izvršava. Ako nije navedena vrednost za inicijalizaciju globalne promenljive, ona se inicijalizuje na 0. Globalne promenljive smeštaju se u posebnu oblast memorije rezervisanu za tu namenu. Korisne su kada se isti podatak koristi u više funkcija, ili kada neka promenljiva treba da zadrži vrednost tokom celokupnog izvršavanja programa.

Ipak, korišćenje globalnih promenljivih treba izbegavati iz više razloga:

- zauzimaju memoriju sve vreme tokom izvršavanja programa, a ne samo onda kada su potrebne
- ako se globalna promenljiva koristi u funkciji, ona postaje manje samostalna jer se oslanja na promenljivu koja je definisana izvan nje
- globalne promenljive prouzrokuju greške u programima koje se teško otkrivaju, zbog promena vrednosti na raznim mestima u programu.

### 5.3.3. Statičke promenljive

Umesto korišćenja globalnih promenljivih u funkcijama kada je potrebna promenljiva koja pamti vrednosti između različitih poziva iste funkcije bolje je koristiti statičke promenljive. Deklaracija statičke promenljive postiže se dodavanjem modifikatora `static` ispred tipa promenljive:

```
static int brojac;
```

Statička promenljiva inicijalizuje se samo jednom, pri prvom izvršavanju funkcije, pamti svoje vrednosti tokom celog izvršavanja programa, a dostupna je u bloku u kome je deklarirana. Sledeći kôd ilustruje korišćenje statičke promenljive za brojanje koliko je puta funkcija pozvana.

---

```
#include <iostream>

using namespace std;

void zapamti(); //prototip funkcije

int main() {
 for(int i=0; i<=3; i++)
 zapamti();
 return 0;
}

void zapamti() {
 static int brojac = 0;
 cout << "Ovo je " << ++brojac << " put da sam pozvana" << endl;
}
```

Rezultat izvršavanja:

**Ovo je 1. put da sam pozvana**  
**Ovo je 2. put da sam pozvana**  
**Ovo je 3. put da sam pozvana**  
**Ovo je 4. put da sam pozvana**

---

Kada se modifikator `static` primeni na globalnu promenljivu, pravi se globalna promenljiva koja je poznata samo datoteci u kojoj je deklarirana. To znači da i pored toga što je promenljiva globalna, funkcije izvan datoteke u kojoj je deklarirana ne mogu da joj promene vrednost. Generalno, modifikator `static` služi za ograničavanje vidljivosti promenljive i za sprečavanje sporednih efekata u velikim programima.

## 5.4. Imenici

*Imenici* (namespaces) su noviji dodatak u standardnom jeziku C++ i omogućuju kreiranje oblasti važenja (dosega, scope) za globalne identifikatore da bi se izbegli sukobi, tj. preklapanja. Pre uvođenja mehanizma imenika, u jeziku C++ je postojala zbrka sa nazivima promenljivih i funkcija. Na primer, ako biste u svom programu definisali funkciju `toupper()`, u zavisnosti od njenih parametara moglo bi se desiti da preklopi funkciju `toupper()` standardne biblioteke, pošto su oba imena bila u istom globalnom imeniku.

Imenik omogućuje korišćenja istog imena u različitim kontekstima bez sukoba. Novi imenik se deklarise navođenjem ključne reči `namespace` ispred deklarativne oblasti:

```
namespace ime {
 // deklaracije
}
```

Sve što je deklarirano unutar imenika ima njegov doseg (scope).

U svim programima dosada koristili smo naredbu  
`using namespace std;`

Direktiva `using` uvozi sva imena iz imenika u datoteku, pa svim identifikatorima koji su definisani u imeniku nadalje može da se pristupi direktno. Bez direktive `using`, mora da se koristi operator razrešenja dosega (`::`), kao u sledećem primeru:

```
#include <iostream>

namespace imenik {
 int vrednost = 0;
}

int main() {
 std::cout << "unesite ceo broj: ";
 std::cin >> imenik::vrednost;
 std::cout << std::endl << "uneli ste: " <<
 imenik::vrednost << std::endl;
 return 0;
}
```

Imenik se ne može deklarirati unutar funkcije, već mu je namena suprotna: u njemu treba da se nađu funkcije, globalne promenljive i druga imena. Funkcija `main()` ne sme da se nađe u nekom imeniku jer je kompajler ne bi prepoznao.

Specijalna vrsta imenika, tzv. bezimeni imenik (unnamed namespace) omogućuje definisanje identifikatora koji su vidljivi samo unutar datoteke:

```
namespace {
 // deklaracije
}
```

Unutar datoteke koja sadrži bezimeni imenik identifikatori se mogu koristiti direktno, bez kvalifikacije, a van te datoteke su nepoznati. Zbog toga bezimene imenike treba koristiti umesto statičkih globalnih promenljivih.

## 5.5. Nizovi i pokazivači kao argumenti funkcija

Kada se funkcijama umesto prostih ugrađenih tipova kao što su `int` ili `double` prosleđuju pokazivači i nizovi kao argumenti, potrebno je poznavati mehanizam njihovog prenošenja u funkciju. Kada se kao argument prosleđuje pokazivač, parametar funkcije se mora deklarirati kao pokazivački tip, što ilustruje sledeći primer:

```
//prosleđjivanje pokazivaca funkciji
#include <iostream>

using namespace std;

void funkcija(int *p);
int main() {
 int i=0;
 int *p; p=&i; //p pokazuje na i
 //prosleđjivanje pokazivaca kao argumenta funkcije
 funkcija(p);
 cout << i << endl; //i je sada 100
 return 0;
}

void funkcija(int *p) {
 //promenljivoj na koju pokazuje p dodeljuje se 100
 *p = 100;
}
```

`funkcija()` prihvata jedan parametar koji je pokazivač na `int`. U funkciji `main()`, pokazivaču `p` dodeljuje se adresa celobrojne promenljive `i`, a zatim se `funkcija()` poziva sa pokazivačem `p` kao argumentom. Kada pokazivački parametar prihvati `p`, i on će pokazivati na promenljivu `i` iz funkcije `main()`. Dodela `*p = 100;` zapravo je dodela vrednosti 100 promenljivoj `i` preko pokazivača. U opštem slučaju, korišćenje pokazivačke promenljive `p` nije neophodno; umesto toga, funkcija bi se mogla pozvati kao `funkcija(&i)`, čime bi joj se prosledila adresa promenljive `i`.

Veoma je važno shvatiti da kada funkcija kao parametar ima pokazivač, zapravo se u njoj radi sa promenljivom na koju taj pokazivač pokazuje, što znači da funkcija može da promeni vrednost objekta na koji taj pokazivač pokazuje.

Kada je parametar funkcije niz, prilikom stvarnog poziva sa argumentima prosleđuje se adresa prvog elementa niza, a ne ceo niz. Pošto je ime niza pokazivač na prvi element niza, to znači da se zapravo prosleđuje pokazivač na niz, pa funkcija može da promeni sadržaj niza koji se koristi kao stvarni argument prilikom poziva.

---

```
#include <iostream>

using namespace std;

void prikaziNiz(int brojevi[10]);

//prosledjivanje niza funkciji
int main() {
 int t[10], i;
 //inicijalizacija niza
 for(i=0;i<10;i++)
 t[i]=i;
 prikaziNiz(t);
 return 0;
}

void prikaziNiz(int brojevi[10]) {
 for(int i=0;i<10;i++)
 cout << brojevi[i] << " ";
 cout << endl;
}
```

Rezultat izvršavanja:

**0123456789**

---

Postoje tri načina da se parametar funkcije deklarira tako da može da primi niz. Prvi je da se parametar deklarira kao niz istog tipa i dimenzija kao onaj koji će se koristiti za poziv funkcije, kao u prethodnom primeru:

```
void prikaziNiz(int brojevi[10]);
```

Iako je parametar `brojevi` deklarisan kao niz od deset celih brojeva, C++ kompajler će ga automatski konvertovati u pokazivač na `int`. Drugi način je da se parametar deklarira kao niz bez dimenzija:

```
void prikaziNiz(int brojevi[]);
```

Pošto C++ ne proverava granice niza, stvarne dimenzije niza nisu bitne za parametar (ali su za program jako bitne). I deklaraciju ovog tipa kompajler takođe konvertuje u pokazivač na `int`.

Treći način deklarisanja, koji se i najčešće koristi u profesionalnim C++ programima, jeste da se parametar odmah deklarira kao pokazivač:

```
void prikaziNiz(int *brojevi);
```

Važno je zapamtiti da se u slučaju korišćenja niza kao argumenta funkcije, toj funkciji prosleđuje adresa prvog elementa niza. To znači da će kôd funkcije potencijalno moći da izmeni stvarni sadržaj niza koji se koristi za pozivanje funkcije. Na

primer, u sledećem programu funkcija `kub()` svaki element niza zamenjuje njegovom kubnom vrednošću. U pozivu funkcije `kub()` adresa niza se prosleđuje kao prvi argument, a njegova dimenzija kao drugi.

---

```
#include <iostream>

using namespace std;

void prikaziNiz(int brojevi[10]);
void kub(int*, int);

int main() {
 int t[10], i;
 //inicijalizacija niza
 for(i=0; i<10;i++)
 t[i] = i+1;
 cout << "originalan niz: ";
 prikaziNiz(t);
 kub(t, 10);
 cout << "izmenjen niz: ";
 prikaziNiz(t);
 return 0;
}

//funkcija za prikaz niza
void prikaziNiz(int brojevi[10]) {
 for(int i=0; i<10; i++)
 cout << brojevi[i] << " ";
 cout << endl;
}

//primer funkcije koja menja vrednost niza
void kub(int *p, int n) {
 while(n) {
 *p = *p * *p * *p;
 n--;
 p++;
 }
}
```

Rezultat izvršavanja:

**originalan niz: 1 2 3 4 5 6 7 8 9 10**

**izmenjen niz: 1 8 27 64 125 216 343 512 729 1000**

---

## 5.6. Reference

U opštem slučaju, programski jezici podržavaju dva načina za prenos argumenata u funkciju. Prvi se zove *prenos po vrednosti* (pass-by-value) i sastoji se u kopiranju vrednosti argumenta u parametre funkcije. To znači da promene parametra unutar funkcije nemaju nikakvog efekta na argument koji se koristi za poziv funkcije. Ovo je standardan način prenosa argumenata u funkciju u jeziku C++. Drugi pristup je *prenos po adresi* ili prenos po referenci (pass-by-reference). Kod tog pristupa, u parametar se kopira adresa stvarnog argumenta, a ne njegova vrednost. Unutar funkcije ta adresa se koristi za posredan pristup stvarnom argumentu sa kojim je funkcija pozvana. To znači da promene parametra unutar funkcije menjaju stvarni argument sa kojim je pozvana. Iako je standardan način prenosa parametara u jeziku C++ po vrednosti, moguće je "ručno" ostvariti prenos po adresi, tako što se kao parametri funkcije zadaju pokazivači, o čemu smo diskutovali u prethodnom odeljku. Kao primer prenosa po adresi, pogledajmo program koji koristi funkciju `swap()` za zamenu vrednosti dve promenljive na koje pokazuju argumenti funkcije.

---

```
#include <iostream>

using namespace std;

//prototip funkcije koja zamenjuje vrednosti
//svojih argumenata
void swap(int *x, int *y);

int main() {
 int i=10, j=20;
 cout << "pocetne vrednosti: " << i << ", " << j << endl;
 //funkciji swap prosledjuju se adrese argumenata
 swap(&i, &j);
 cout << "posle zamene: " << i << ", " << j << endl;
 return 0;
}

void swap(int *x, int *y) {
 int temp;
 temp = *x; //sacuvaj vrednost na adresi x
 *x = *y; //stavi y u x
 *y = temp; //stavi x u y
}
```

Rezultat izvršavanja:

**pocetne vrednosti: 10, 20**

**posle zamene: 20, 10**

---

Funkcija `swap()` deklarira dva pokazivačka parametra, `x` i `y` koji se koriste za zamenjivanje vrednosti promenljivih na koje pokazuju argumenti koji se prosleđuju funkciji. Pošto `swap()` očekuje da primi dva pokazivača, kada se poziva kao argumenti se moraju proslediti adrese promenljivih koje treba da se zamene.

### 5.6.1. Reference u funkcijama

Pristup "ručnog" prosleđivanja adrese preko pokazivača je prilično nezgodan, iz dva razloga. Prvo, primorava nas da sve operacije obavljamo preko pokazivača, a drugo, ne smemo zaboraviti da prilikom pozivanja funkcije prosledimo adrese promenljivih (umesto vrednosti). Umesto "ručnog" poziva po adresi pozivanjem funkcije sa argumentima pokazivačima, može se C++ kompajleru reći da automatski koristi prenos po adresi. To se postiže tako što se kao argumenti funkcije navode *reference*.

Kada je parametar funkcije referenca, njoj se automatski prosleđuje adresa argumenta. Unutar funkcije, operacije sa parametrom se automatski dereferenciraju, što znači da se ne radi sa adresom promenljive nego sa njenom vrednošću. Referenca se deklarira tako što se ispred imena promenljive navodi `&`, kao u sledećem primeru:

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

//prototip funkcije ciji su argumenti reference
void swap(int &x, int &y);

int main() {
 int i=10, j=20;
 cout << "pocetne vrednosti: " << i << ", " << j << endl;
 //funkciji swap prosledjuju se direktno argumenti
 swap(i, j);
 cout << "posle zamene: " << i << ", " << j << endl;
 return 0;
}

void swap(int &x, int &y) {
 int temp;
 temp = x; //sacuvaj vrednost na adresi x
 x = y; //stavi y u x
 y = temp; //stavi x u y
}
```

Uz korišćenje referenci, u funkciji nije potrebno koristiti operator dereferenciranja (\*) kao što je slučaj sa pokazivačima. Takođe, treba znati da operacije koje se izvode nad parametrom koji je referenca utiču na samu vrednost parametra, a ne na njegovu adresu.

Funkcija čiji je argument referenca poziva se na isti način kao da se radi o prenosu po vrednosti. Primetite da je u prethodnom primeru funkcija `swap()` deklarirana tako

da prihvata reference kao argumente, a u redu gde se poziva, tj. u naredbi `swap(i, j)`; prosleđuju joj se stvarni argumenti, a ne njihove adrese. Parametar se inicijalizuje adresom argumenta, bez ikakvog kopiranja vrednosti.

### 5.6.2. Samostalne reference

Referenca se koristi kao alias (drugo ime) za promenljivu, tj. može se koristiti kao alternativni način za pristup promenljivoj. Prilikom deklaracije, nezavisna referenca se uvek inicijalizuje i tokom celog svog životnog veka je vezana za objekat kojim je inicijalizovana. Reference se ne mogu "preusmeravati" na druge promenljive kao pokazivači. Ipak, reference se retko koriste samostalno; njihova najvažnija upotreba je u parametrizaciji funkcija.

```
long broj = 10;
//definisanje reference na promenljivu broj
long &rbroj = broj;
cout << "broj pre: " << broj << endl; //ispisuje 10
rbroj+=10; //povecava se broj preko reference
cout << "broj posle: " << broj << endl; //ispisuje 11
```

### 5.6.3. Pokazivači i reference

Iako između pokazivača i referenci ima dosta sličnosti, treba imati na umu i sledeće veoma bitne razlike:

- pokazivač se može preusmeriti tako da pokazuje na neki drugi objekat (promenljivu), a referenca ne može
- pokazivač može da ne pokazuje ni na šta, a referenca od početka do kraja svog životnog veka pokazuje na jedan isti objekat
- pristup objektu preko pokazivača obavlja se pomoću operatora `*`, dok je preko reference pristup neposredan
- moguće je napraviti nizove pokazivača, ali ne i nizove referenci
- ne može se napraviti referenca na referencu

### 5.6.4. Reference kao povratni tip funkcija

Kada je povratni tip referenca, treba voditi računa o životnom veku promenljive povezane sa referencom. Pokazivač na lokalnu promenljivu, kao ni referenca na lokalnu promenljivu u funkciji nikada ne smeju da budu povratna vrednost funkcije. Pošto referenca nikada ne postoji samostalno, već je uvek alias neke druge promenljive, objekat s kojim je povezana mora da postoji i nakon završetka izvršavanja funkcije. Referenca kao argument funkcije pravi se pri svakom pozivu funkcije, i uništava po završetku izvršavanja funkcije. To znači da u sledećem primeru promenljiva i koja

je povratna vrednost funkcije neće postojati kada se funkcija vrati na mesto gde je pozvana:

```
//funkcija ciji je povratni tip referenca na int
int& f() {
 int i = 10;
 return i; //promenljiva i vise ne postoji kada se funkcija završi!
}
```

### 5.6.5. Korišćenje modifikatora const u funkcijama

Kada se modifikator const navede ispred imena parametra funkcije, to znači da odgovarajući argument nikako ne sme biti promenjen. Kompajler proverava da li se argument unutar funkcije menja i upozorava na to. Korišćenjem referenci može se izbeći nepotrebno kopiranje argumenata prilikom poziva funkcije, a ako se one koriste uz modifikator const, obezbeđuje se i dodatna zaštita od slučajne izmene argumenta.

---

```
#include <iostream>
using namespace std;
int incr10(const int &broj);
int main() {
 const int primer = 20;
 cout << "pre poziva funkcije: " << primer << endl;
 cout << "vrednost koja se vraća iz funkcije: " <<
 incr10(primer) << endl;
 cout << "posle poziva funkcije: " << primer << endl;
 return 0;
}

int incr10(const int &broj) {
 //return broj+=10; ovo bi bila greska
 return broj + 10;
}
```

Rezultat izvršavanja:

**pre poziva funkcije: 20**

**vrednost koja se vraća iz funkcije: 30**

**posle poziva funkcije: 20**

---

Kada bi se u prethodnom primeru koristila naredba `return broj+=10;` posredno bi se promenio objekat povezan sa referencom (tj. promenljiva broj) koja se prosleđuje kao argument, što bi bila greška jer je funkcija deklarisana sa parametrom koji

je `const`. Zbog toga povratna vrednost funkcije mora biti privremeni objekat koji se kreira naredbom `broj+10`.

## 5.7. Prototipovi funkcija

Prototipove funkcija već smo pomenuli na početku ovog poglavlja, a sada ćemo ih detaljno proučiti. Sve funkcije u jeziku C++ moraju da budu deklarisanе pre prvog korišćenja. Od *prototipa funkcije* kompajler dobija sledeće tri informacije:

- koji je povratni tip funkcije
- koliko ima argumenata
- kog su tipa argumenti

U sledećem primeru naveden je prototip funkcije koja očekuje pokazivač kao argument:

```
void funkcija(int *p);
int main() {
 int x = 10;
 funkcija(x); //greska!
}
```

Prototipovi omogućuju kompajleru da obavi tri važne operacije:

1. kažu kompajleru kakvu vrstu koda treba da generiše prilikom poziva funkcije. Kompajler se različito ophodi prema različitim povratnim tipovima funkcija.
2. omogućuju pronalaženje i prijavljivanje svih nevažećih konverzija tipova između tipova argumenata sa kojima se funkcija poziva i tipova parametara sa kojima je funkcija definisana
3. omogućuju otkrivanje neslaganja između broja argumenata korišćenih u pozivu funkcije i broja parametara sa kojima je funkcija deklarisanа u prototipu

Opšti oblik prototipa funkcije je isti kao i njena definicija, s tim što nema tela:

```
povratni_tip ime (tip param1, tip param2, ... tip paramN);
```

Korišćenje imena parametara u prototipu nije obavezno, ali je preporučljivo zbog čitljivosti.

Obično je lakše i bolje deklarirati prototip za svaku funkciju koja se koristi u programu umesto proveravati da li je svaka funkcija definisana pre prvog korišćenja, naročito u velikim programima u kojima se teško prati međusobno pozivanje funkcija. Ako se funkcija i definiše pre prvog pozivanja u programu, onda ta definicija služi i kao prototip.

*Zaglavlja* (header files) sadrže prototipove svih funkcija koje čine standardnu C++ biblioteku (zajedno sa različitim vrednostima i definicijama koje koriste te funkcije). Na početku datoteke koja poziva neku standardnu funkciju uvek se navodi zaglavlje u kome se ona nalazi da bi bio dostupan prototip te funkcije iz zaglavlja.

Funkcija `main()` je posebna zato što je to prva funkcija koja se poziva prilikom izvršavanja programa. C++ svaki program započinje pozivom funkcije `main()`, bez obzira gde se ona u programu nalazi. Program može i mora da ima samo jednu funkciju `main()` i za nju ne postoji prototip jer je ugrađena u jezik C++.

## 5.8. Preklapanje funkcija

U jeziku C++ može da postoji nekoliko funkcija istog imena, sa različitim brojem i(li) tipovima argumenata; to se zove *preklapanje funkcija* (function overloading). Ova mogućnost je vrlo korisna za funkcije koje obavljaju isti zadatak nad različitim tipovima podataka. Na primer, moguće je u istom programu imati dva prototipa za funkciju `min()` koja pronalazi manji od svoja dva parametra:

```
int min(int a, int b); //min za cele brojeve
char min(char a, char b); //min za znake
```

Da bi dve funkcije bile preklapljene, nije dovoljno da im se razlikuje samo povratni tip, već i tip ili broj argumenata moraju biti različiti. Kada se poziva preklapljena funkcija, kompajler na osnovu tipa argumenata poziva odgovarajuću funkciju.

Vrednost preklapanja je u tome što omogućuje korišćenje skupa funkcija slične namene sa istim imenom. U prethodnom primeru, naziv `min` označava opštu operaciju koja će biti sprovedena, a kompajleru se ostavlja da izabere pravu verziju u odgovarajućim okolnostima. Tako se pomoću mehanizma preklapanja ostvaruje polimorfizam, o kome ćemo detaljnije učiti u Poglavlju 7. Druga prednost preklapanja je u tome što omogućuje definisanje različitih verzija iste funkcije specijalizovanih za tip podataka sam kojima funkcija radi. U sledećem primeru ilustrovano je kako funkcija `min()` u slučaju celobrojnih argumenata nalazi manji broj, a znakove poredi nešto drugačije, uz zanemarivanje velikih i malih slova.

---

```
#include <iostream>

using namespace std;

//prototipovi preklapljenih funkcija
int min(int a, int b);
char min(char a, char b);

int main() {
 cout << min(2, 3) << endl;
 cout << min('a', 'B') << endl;
 return 0;
}
```

```

int min(int a, int b) {
 int min;
 (a>b) ? min=a : min=b;
 return min;
}

//min za char ignorise velika slova
char min(char a, char b) {
 char min;
 (tolower(a)>tolower(b)) ? min=a : min=b;
 return min;
}

```

Rezultat izvršavanja:

```

9
X
9

```

Kada se funkcije prelope, svaka verzija funkcije može da izvodi proizvoljne operacije, odnosno nema pravila koje bi diktiralo da preklopljene funkcije moraju da budu u neakvoj vezi. Međutim, iako mogu da se koriste ista imena za funkcije koje nisu ni u kakvoj vezi, to nije dobar stil u programiranju, jer poništava ideju preklapanja.

### 5.8.1. Podrazumevani argumenti funkcija

Parametru funkcije može da se dodeli podrazumevana (default) vrednost koja će se koristiti kada u pozivu funkcije nije naveden odgovarajući argument za taj parametar. Podrazumevani argument zadaje se slično inicijalizaciji promenljive. Na primer,

```
void mojaFunkcija(int x=0, int y= 100);
```

deklariše podrazumevane vrednosti svojih parametara na 0 i 100, redom. Sada se `mojaFunkcija()` može pozvati na jedan od sledeća tri načina:

```
mojaFunkcija(1, 2) //direktno prosledjivanje vrednosti
```

```
mojaFunkcija(10); //promenljivoj x se prosledjuje vrednost, y je 100
```

```
mojaFunkcija(); //x je 0 a y je 100
```

Podrazumevane (default) vrednosti mogu biti navedene samo jednom, i to prilikom prve deklaracije funkcije. Svi parametri koji prihvataju podrazumevane vrednosti moraju biti zadati sa desne strane parametara koji nemaju podrazumevane vrednosti; kada se počne sa dodeljivanjem podrazumevanih vrednosti, više se ne može zadavati parametar bez podrazumevane vrednosti:

```
void funkcija(int a=10, int b); //pogresno!
```

Iako su podrazumevani argumenti snažna alatka kada se ispravno koriste, često se dešava i da se zloupotrebljavaju. Njihova svrha je da omoguće funkciji da obavi

svoj zadatak efikasno i lako, ali uz zadržavanje prilagodljivosti. U skladu sa tim, svi podrazumevani argumenti trebalo bi da odražavaju način na koji se funkcija inače koristi, odnosno zamenu za to. Ako ne postoji neka određena vrednost koja bi se dodelila parametru, nema razloga da se uvodi podrazumevani parametar. Zapravo, u tom slučaju razumljivost kôda bi bila smanjena. Takođe, podrazumevani argumenti ne bi smeli da prouzrokuju štetne sporedne efekte.

---

```
#include <iostream>

using namespace std;

void saberi(int x=0, int y=100);

int main() {
 saberi(20, 30);
 saberi(20);
 saberi();
 return 0;
}

void saberi(int x, int y) {
 cout << x + y << endl;
}
```

Rezultat izvršavanja:

```
50
120
100
```

---

## 5.9. Rekurzivne funkcije

Rekurzivne funkcije su funkcije koje pozivaju same sebe. Prava svrha rekurzivnih funkcija u programiranju jeste da svedu složene probleme na problem koji se može rešiti. Pogodne su za rešavanje problema koji se elegantnije rešavaju rekurzijom, kao što je na primer izračunavanje faktoriijela.

Većina rekurzivnih funkcija može se napisati i “iterativno”, uz korišćenje petlji. Rekurzija se izvršava sporije od iteracija, zato što se pri svakom pozivu funkcije prave nove kopije argumenata i lokalnih promenljivih na steku. Treba takođe voditi računa i o “dubini” rekurzije, jer se može desiti i da se stek “istroši”. Kao primer rekurzivne funkcije navodimo program koji ispisuje string obrnutim redosledom uz korišćenje pokazivačke aritmetike.

---

```
#include <iostream>

using namespace std;

void ispisiStringNaopako(char*);

int main() {
 char str[] = "Ovo je test";
 ispisiStringNaopako(str);
 return 0;
}

void ispisiStringNaopako(char *p) {
 if(*p)
 ispisiStringNaopako(p+1);
 else
 return;
 cout << *p;
}
```

Rezultat izvršavanja:

**tset ej ovO**

---

Funkcija `obrniString()` kao argument prihvata pokazivač na tip `char`; pošto je ime niza znakova zapravo pokazivač na prvi element niza, poziv funkcije `obrniString()` iz funkcije `main()` sa argumentom tipa `char[]` je validan. Funkcija `obrniString()` poziva samu sebe, ali kao argument prosleđuje sledeći znak u nizu (pomeranjem pokazivača na sledeći znak naredbom `p+1`). Rekurzivno pozivanje se prekida kada se nađe na poslednji znak u nizu, tj. nulu; tada se izvršava `else` deo naredbe. Prilikom povratka iz funkcije izvršava se prva naredba iza mesta gde je funkcija pozvana, a to je `cout`. Tako se string ispisuje unazad.

## Pitanja

1. Pravilno deklarisana funkcija je:

- a) `main[]{}`
- b) `main{}( )`
- c) `main(){}`

2. Funkcija čiji je povratni tip `void`:

- a) vraća celobrojnu vrednost
- b) vraća vrednost tipa `void`

c) ne vraća vrednost

3. Ako je data funkcija `double nadji(int x, int y, bool b) { ... }`

- a) ime funkcije je `nadji`
- b) povratna vrednost je `int`
- c) povratna vrednost je `bool`
- d) broj parametara funkcije je tri

4. Na koji način funkcija vraća vrednost na mesto odakle je pozvana?

- a) automatski
- b) svojim imenom
- c) preko parametara u zagradi

5. Funkcija koja se poziva iz programa `main`:

- a) mora biti deklarirana u programu `main`
- b) mora biti deklarirana pre programa `main`
- c) može biti deklarirana bilo gde u programu

6. Parametri se u funkciju mogu preneti:

- a) po vrednosti ili po adresi
- b) po imenu, vrednosti ili adresi
- c) po adresi, imenu ili referenci

7. Definisanje različitih funkcija istog imena je:

- a) dozvoljeno
- b) nije dozvoljeno
- c) zavisi od prethodnog koda

8. Niz može da bude povratni tip funkcije.

- a) tačno
- b) pogrešno

9. Niz može da bude parametar funkcije.

- a) tačno
- b) pogrešno

10. U jeziku C++ postoji funkcija koja ne mora da ima prototip.

- a) tačno
- b) pogrešno

11. Standardni način prenosa argumenata u funkciju je:

- a) po vrednosti
- b) po adresi

12. Preklopljene funkcije mogu da se razlikuju samo po povratnom tipu.

- a) tačno
- b) pogrešno

13. Deklaracija funkcije `void f(int a=10, int b);` je ispravna.

- a) tačno
- b) pogrešno

14. Kada se funkciji argumenti prenose po vrednosti, ona može da ih izmeni.

- a) tačno
- b) pogrešno

15. Kada se funkciji argumenti prenose po adresi, ona može da ih izmeni.

- a) tačno
- b) pogrešno

16. Prototip funkcije sprečava njeno pozivanje sa neodgovarajućim brojem i(li) tipovima argumenata.

- a) tačno
- b) pogrešno

17. Naredba koja se koristi unutar tela funkcije za povratak iz funkcije i vraćanje rezultata je:

- a) void
- b) return
- c) public
- d) static

18. Globalne promenljive su dostupne:

- a) samo jednom delu programa
- b) celom programu

19. Opseg važenja globalne promenljive je:

- a) ceo program
- b) funkcija main
- c) ceo program posle mesta deklarisanja

20. Šta označava reč `static` ako se napiše ispred promenljive?

- a) promenljiva zadržava vrednost tokom izvršavanja programa
- b) promenljiva se ne nalazi u memoriji

21. Oblast važenja lokalne promenljive je:

- a) ceo kôd
- b) blok u kojem je definisana
- c) funkcija main

22. Promenljiva koja je deklarisanu unutar funkcije je:

- a) globalna
- b) statička
- c) lokalna

23. Statička globalna promenljiva je vidljiva samo u datoteci u kojoj je deklarisanu.

- a) tačno
- b) pogrešno

24. Kada se poziva funkcija čiji je parametar referenca, ispred naziva stvarnog argumenta funkcije treba da bude `&`.

- a) tačno
- b) pogrešno

25. Funkcija nikako ne bi trebalo da vraća referencu na lokalnu promenljivu.

- a) tačno
- b) pogrešno

26. Reference se mogu preusmeravati na druge objekte, isto kao pokazivači.

- a) tačno
- b) pogrešno

27. Korišćenje referenci olakšavanje prenos argumenata u funkciju:

- a) po vrednosti
- b) po adresi

28. Moguće je definisati niz referenci.

- a) tačno
- b) pogrešno

29. Ako je parametar funkcije konstantna referenca (const &), ona sme da promeni vrednost stvarnog argumenta sa kojim je pozvana.

- a) tačno
- b) pogrešno

30. Funkcije u jeziku C++ mogu se rekurzivno pozivati.

- a) tačno
- b) pogrešno





## Glava 6

# Klase i objekti

Dosad smo u knjizi proučavali primere programa koji nisu bili objektno orijentisani, već su koristili tehnike proceduralnog programiranja. Da bi se pisali objektno orijentisani programi, moraju se koristiti klase, i njihovi primerci, odnosno objekti. Klase i objekti su toliko važni za C++ da im je ostatak knjige u potpunosti posvećen.

### 6.1. Kako i zašto je nastalo objektno orijentisano programiranje?

Tehnike proceduralnog programiranja usredsređene su na procedure, odnosno operacije koje se obavljaju u programu. Podaci se obično čuvaju u promenljivama, a sa njima rade funkcije. Podaci i funkcije koje rade sa njima potpuno su odvojeni. Program radi tako što se promenljive prosleđuju funkcijama, koje rade nešto sa njima i vraćaju rezultat. Dakle, proceduralno programiranje je prvenstveno orijentisano na funkcije. Međutim, ovakav pristup prouzrokuje dva velika problema:

- Preterano korišćenje globalnih podataka: programeri često pribegavaju čuvanju najvažnijih podataka u globalnim promenljivama, da bi im se lako pristupalo iz svih funkcija u programu. Međutim, tako se istovremeno otvaraju vrata za nehotično uništavanje ili oštećivanje najvažnijih podataka.
- Projektovanje složenih programa koji se teško održavaju i menjaju: Iako je preporučljivo da se program učini veoma modularnim, tj. da se izdela u veliki broj logičkih funkcija, postoji granica koju je programer u stanju da sagleda. Realni programi imaju na stotine funkcija čiju interakciju je veoma teško sagledati, pa menjanje kôda postaje izuzetno teško. Pre bilo kakve izmene potrebno je razumeti kako će promena uticati na druge delove programa.

Kako su programi postajali složeniji, programeri su shvatili da im je potreban mehanizam koji će olakšati projektovanje i održavanje velikih aplikacija. Objektno orijentisan pristup su smislili programeri kao način da sebi olakšaju život, tj. kao odgovor na softversku krizu koja se ogledala u tome što se softver projektovao duže nego što je predviđeno, koštao više nego što je predviđeno i nije zadovoljavao sve postavljene zahteve.

Objektno orijentisan pristup, kako mu i ime kaže, usredsređen je na objekte koji objedinjuju podatke i funkcije koje rade sa njima. Najvažniji princip koji se koristi pri-

likom definisanja objekata jeste *apstrakcija*. Objekti su apstrakcije pojmova iz stvarnog sveta, što znači da se modeliraju tako da sadrže samo one osobine koje su bitne za program. Na primer, ako klasa Osoba modelira pojam osobe iz stvarnog sveta, programu je možda važno da zna koliko ta osoba ima godina i kako se zove, a nije bitno kog je pola.

## 6.2. Pojam klase

Mehanizam klasa omogućuje korisnicima da definišu sopstvene tipove podataka. Zbog toga se klase često nazivaju i korisnički definisani tipovi podataka. Klase se koriste kao šablon kojim se modeliraju objekti sličnih svojstava, a objekti su konkretni primeri (instance) klase. Uočavanje zajedničkih osobina (svojstava) objekata i njihovo grupisanje u klasu naziva se apstrakcija. Klasa sadrži *podatke članove* (member data) i *funkcije članice* ili metode (member functions). Podaci članovi najčešće predstavljaju unutrašnjost klase, tj. njenu realizaciju, a funkcije članice njen interfejs, odnosno ono što se može raditi sa njenim objektima.

Klasa se deklarise pomoću ključne reči `class`. Opšti oblik deklaracije klase je

```
class ime { lista članova } lista objekata;
```

Ime klase postaje novi tip koji se može koristiti za kreiranje objekata klase (odnosno promenljivih koje su tipa te klase). Iza deklaracije klase mogu se navesti nazivi objekata, ali to nije obavezno. Klasa može da sadrži privatne i javne članove; standardno su svi članovi klase privatni. To znači da im mogu pristupati isključivo drugi članovi iste klase. Na taj način postiže se OO princip kapsuliranja ili skrivanja podataka. Da bi se delovi klase učinili javnim, tj. da bi se omogućio pristup iz drugih delova programa, oni se moraju deklarirati kao javni pomoću ključne reči `public`. Klase se po pravilu projektuju tako da su podaci članovi privatni, a funkcije članice javne.

Iako nema sintaksnog pravila koje to nalaže, dobro projektovana klasa trebalo bi da definiše samo jednu logičku celinu. Na primer, u klasi koja čuva imena i telefonske brojeve ne bi trebalo da se nađu i informacije o berzi, broju sunčanih sati u godini niti bilo šta drugo. Dobro projektovane klase grupišu logički povezane informacije.

Da ponovimo: u jeziku C++, klasa kreira nov tip podataka koji se može koristiti za kreiranje objekata. Klasa je logički okvir koji definiše vezu između svojih članova. Kada se deklarise promenljiva tipa klase, zapravo se kreira objekat te klase i za njega se odvaja prostor u memoriji.

Kao primer proučićemo klasu koja čuva informacije o vozilima, kao što su automobili, kombiji i kamioni. Tu klasu nazvaćemo `Vozilo` i u nju ćemo smestiti tri podatka o vozilu: broj mesta za putnike, kapacitet rezervoara za gorivo i prosečnu potrošnju goriva.

```
class Vozilo {
 public:
 int brojMesta; //broj mesta za sedenje
 int kapacitetRezervoara; //kapacitet rezervoara u litrama
 int potrosnja; //prosecna potrosnja u litrama na 100 km
};
```

Klasa Vozilo ima tri podatka člana: putnici, rezervoar i potrosnja. Klasa Vozilo za sada nema funkcija članica; njih ćemo dodati kasnije.

Opšti oblik deklarisanja podataka članova u klasi je:

*tip promenljiva;*

Dakle, podaci članovi deklarishu se na sličan način kao "obične" promenljive. U klasi Vozilo ispred svih podataka članova nalazi se specifikator javnog pristupa (public). To omogućuje da im se pristupa iz kôda koji se nalazi izvan klase Vozilo. Definicija klase kreira nov tip podataka, koji se u ovom slučaju zove Vozilo. To ime se može koristiti za deklarisanje promenljivih tipa Vozilo, što su zapravo objekti klase Vozilo. Zapamtite da je deklaracija klase samo opis tipa i njome se ne pravi stvarni objekat. Zbog toga prethodna deklaracija ne kreira nikakve objekte tipa Vozilo. Da bi se kreirao objekat klase Vozilo, treba upotrebiti naredbu za deklaraciju sledećeg oblika:

```
Vozilo kombi; //kreira objekat klase Vozilo koji se zove kombi
```

Kreiranje objekata klase ponekad se zove i instanciranje klase. Kada se izvrši ova naredba, kombi će postati primerak (objekat) klase Vozilo. Kad god se kreira primerak klase, kreira se objekat koji sadrži sopstvenu kopiju podataka članova klase. To znači da će svaki objekat klase Vozilo imati svoje kopije promenljivih (podataka članova) putnici, rezervoar i potrosnja.

Operator tačka povezuje ime objekta sa njegovim podacima članovima. Opšti oblik operatora tačka je

*objekat.član*

Objekat se zadaje sa leve strane, a podatak član sa desne. Na primer, da bi se podatku članu gorivo objekta kombi dodelila vrednost 50, koristi se sledeća naredba:

```
kombi.gorivo = 50;
```

Operator tačka u opštem slučaju koristi se za pristup podacima članovima i funkcijama članicama klase. Evo primera programa koji koristi klasu Vozilo:

```
class Vozilo {
 public:
 int brojMesta;
 int kapacitetRezervoara;
 int potrosnja;
};
```

```

int main() {
 Vozilo kombi;
 Vozilo automobil;

 double kilometrazaKombi, kilometrazaAuto;

 //dodela vrednosti podacima clanovima objekta kombi
 kombi.brojMesta = 7;
 kombi.kapacitetRezervoara = 60;
 kombi.potrosnja = 7.;

 //dodela vrednosti podacima clanovima objekta automobil
 automobil.brojMesta = 5;
 automobil.kapacitetRezervoara = 50;
 automobil.potrosnja = 5.;

 autonomijaKombi = kombi.kapacitetRezervoara /
 kombi.potrosnja * 100;
 autonomijaAuto = automobil.kapacitetRezervoara /
 automobil.potrosnja * 100;

 cout << "U kombi staje " << kombi.brojMesta <<
 " putnika i sa punim rezervoarom prelazi "
 << autonomijaKombi << " kilometara." << endl;
 cout << "U auto staje " << automobil.brojMesta <<
 " putnika i sa punim rezervoarom prelazi "
 << autonomijaAuto << " kilometara." << endl;
 return 0;
}

```

Rezultat izvršavanja:

**U kombi staje 7 putnika i sa punim rezervoarom prelazi 800 kilometara.**  
**U auto staje 5 putnika i sa punim rezervoarom prelazi 1000 kilometara.**

Funkcija `main()` kreira dva objekta klase `Vozilo` koji se zovu `kombi` i `automobil`. Zatim pristupa podacima članovima tih objekata dodeljujući im vrednosti. Kôd u funkciji `main()` može da pristupi članovima klase `Vozilo` zato što su deklarirani kao javni (`public`). U prethodnom primeru važno je uočiti da svaki objekat ima sopstvene kopije svojih podataka članova definisanih u klasi. Između objekata `kombi` i `automobil` nema nikakve veze osim činjenice da su istog tipa: svaki od njih ima sopstvene vrednosti za podatke članove `brojMesta`, `kapacitetRezervoara` i `potrosnja`.

Većina klasa osim podataka članova ima i *funkcije članice*. U opštem slučaju, funkcije članice rade sa podacima članovima i komuniciraju sa drugim delovima programa. Funkcija `main()` u prethodnom primeru računala je autonomiju vozila tako što je kapacitet rezervoara delila potrošnjom. To je zadatak koji tipično treba da obavlja funkcija članica klase, koju ćemo sada dodati u klasu `Vozilo` zadavanjem njenog prototipa u deklaraciji:

```
class Vozilo {
public:
 int brojMesta;
 int kapacitetRezervoara;
 int potrosnja;
 int autonomija(); //funkcija clanica klase
};
```

Pošto se u deklaraciji klase navodi samo prototip funkcije članice, a ne i njena implementacija, na mestu gde se piše implementacija funkcije članice mora se navesti kojoj klasi ona pripada. Za tu svrhu se koristi operator razrešenja dosega (::):

```
//implementacija funkcije clanice autonomija
int Vozilo::autonomija() {
 return kapacitetRezervoara / potrosnja * 100;
}
```

Operator razrešenja dosega :: povezuje ime klase sa nazivom njene funkcije članice da bi kompajler znao na koju klasu se odnosi funkcija članica. Pošto više različitih klasa mogu da imaju funkcije članice istog imena, operator :: je neophodan prilikom implementacije funkcije članice. U ovom slučaju, operator razrešenja dosega povezuje funkciju autonomija() sa klasom Vozilo, odnosno deklarira da je funkcija autonomija() u dosegu klase Vozilo. Telo funkcije autonomija ima samo jednu naredbu; pošto svaki objekat tipa Vozilo ima sopstvenu vrednost podataka članova kapacitetRezervoara i potrosnja, izračunavaće se autonomija za svaki objekat posebno.

Primitite da je sintaksa definicije funkcije članice takva da se prvo navodi njen povratni tip.

Unutar funkcije autonomija() podacima članovima kapacitetRezervoara i potrosnja pristupa se direktno, bez navođenja imena objekta ili tačke. Kada funkcija članica koristi podatak član svoje klase, ona to radi bez eksplicitnog pozivanja objekta. To je zato što se funkcija članica nikad ne poziva samostalno, već se uvek poziva za neki objekat svoje klase, pa nema potrebe da se objekat ponovo navodi.

Postoje dva načina pozivanja funkcije članice. Ako se ona poziva iz kôda koji se nalazi izvan klase, mora se pozivati preko nekog objekta i operatora tačka:

```
kombi.autonomija();
```

Kada se pozove na ovaj način, funkcija autonomija() radi sa vrednostima podataka članova objekta kombi. Drugi način pozivanja funkcije članice je iz druge funkcije članice iste klase. Kada jedna funkcija članice klase poziva drugu funkciju članicu klase, to može da uradi direktno, bez operatora tačka, jer kompajler već zna sa kojim objektom radi.

### 6.3. Kontrola pristupa članovima klase

Članovi (podaci ili metode) klase koji se nalaze ispred specifikacije `private`: zaštićeni su od pristupa spolja (oni su sakriveni, odnosno kapsulirani). To su privatni članovi klase i njima mogu pristupati samo funkcije članice iste klase. Da bi se deo klase učinio javnim, tj. da bi bio vidljiv u kôdu izvan klase, navodi se specifikacija `public`: Članovi iza specifikacije `public`: dostupni su izvan klase i nazivaju se javnim članovima klase.

Ključne reči `public` i `private` zovu se *specifikatori pristupa* (access specifiers). Iza njih se u deklaraciji klase uvek navodi dvotačka.

Privatnim članovima klase obično se pristupa preko javnih funkcija članica. Kada funkcija članica koristi privatan podatak član svoje klase, pristupa mu direktno, bez korišćenja operatora tačka. Funkcija članica se uvek poziva za neki objekat svoje klase.

Ako se ne navede specifikator pristupa, svi članovi klase su podrazumevano privatni. Ipak, dobar stil programiranja nalaže da se i pored toga specifikator `private` navede za eksplicitno deklarisanje privatnih promenljivih. Takođe, po pravilu se u deklaraciji klase prvo navode privatni, a tek onda javni članovi, mada ne postoji pravilo koje to zahteva.

Javnom podatku članu klase može se pristupati iz drugih delova programa. Sintaksa za pristup javnom podatku članu je ista kao sintaksa za pozivanje funkcije članice: ime objekta, tačka, ime podatka člana. Sledeći jednostavan program ilustruje korišćenje javne promenljive:

```
#include <iostream>

using namespace std;

class mojaKlasa {
 public: int i, j, k; //dostupni celom programu
};

int main() {
 mojaKlasa a, b;
 a.i = 100; //pristup i, j, k je u redu
 a.j = 4;
 a.k = a.i * a.j;
 b.k = 12; // a.k i b.k su razliciti
 cout << a.k << " " << b.k; //ispisuje 400 12
 return 0;
}
```

## 6.4. Inline funkcije članice

Kada je telo funkcije članice ne samo deklarirano, već i definisano unutar deklaracije klase, kaže se da je funkcija ugrađena (inline).

```
class Brojac {
 private:
 int i;
 public:
 int broj { return ++i; } //inline funkcija članica
};
```

Inline funkcije od prevodioca zahtevaju da kôd tela funkcije ugradi neposredno u kôd programa na mestima poziva te funkcije, umesto uobičajenog postupka poziva funkcije. Inline je zapravo zahtev za optimizaciju kôda koji kompajler može, ali ne mora da ispuni. Funkcija članica se može deklarirati kao inline i van deklaracije klase. To se radi navođenjem specifikatora inline ispred deklaracije funkcije, kao u sledećem primeru:

```
inline int f() {
 // ...
}
```

Mehanizam ugrađivanja funkcija treba pažljivo primenjivati, samo za vrlo jednostavne funkcije. Pri svakom pozivanju funkcije i povratku iz nje mora da se izvrši niz naredbi (npr. stavljanje argumenata na stek i skidanje vrednosti argumenata sa steka prilikom povratka). Ako se funkcija ugradi program se ubrzava, ali ako je suviše složena, program postaje duži pa ugrađivanje ne poboljšava efikasnost.

## 6.5. Konstruktori i destruktori

U prethodnim primerima podacima članovima objekata klase Vozilo ručno smo dodeljivali vrednosti pomoću niza naredbi:

```
kombi.brojMesta = 7;
kombi.potrosnja = 7;
```

Ovakav pristup ručnog dodeljivanja vrednosti očigledno nije pogodan jer su mogućnosti grešaka velike. C++ nudi bolji način za inicijalizaciju podataka članova, a to je konstruktor. *Konstruktor* je funkcija članica klase koja se automatski poziva kada se kreira objekat. Ima isto ime kao klasa, ali nema nikakav povratni tip (čak ni void). Sintaksa konstruktora izgleda ovako:

```
ime_klase() {
 //telo konstruktora
}
```

Uobičajeno se konstruktor koristi za dodelu početnih vrednosti podacima članovima klase, ili za izvođenje drugih operacija potrebnih za kreiranje objekta.

Često se dešava da je i prilikom uništavanja potrebno obaviti određene operacije, na primer dealocirati memoriju koja je zauzeta ili zatvoriti neku otvorenu datoteku. U jeziku C++ postoji posebna funkcija članica za tu namenu koja se zove *destruktor*. Destruktor ima isto ime kao klasa, ispred koga se dodaje znak ~:

```
~ime_klase() {
 //telo destruktora
}
```

Poput konstruktora, ni destruktor nema povratni tip, a ne može imati ni argumente. Postoji samo jedan destruktor za klasu; ako se posebno ne napiše, kompajler će ga automatski kreirati. Evo jednostavnog primera klase sa konstruktorom i destruktorom:

---

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

class Primer {
public:
 Primer(); //konstruktor
 ~Primer(); //destruktor
};

Primer::Primer() {
 cout << "u konstruktoru"<<endl;
}

Primer::~~Primer() {
 cout << "u destrukturu" << endl;
}

int main() {
 Primer obj;
 cout << "program koji demonstrira objekat" << endl;
 return 0;
}
```

Rezultat izvršavanja:

```
u konstruktoru
program koji demonstrira objekat
u destrukturu
```

---

U ovom slučaju destruktor samo prikazuje poruku, ali u korisnim programima on se koristi za oslobađanje resursa (npr. memorije ili datoteka) koje koristi objekat klase.

### 6.5.1. Kada se izvršavaju konstruktori i destruktori?

Konstruktor lokalnog objekta izvršava se kada kompajler naiđe na naredbu deklaracije objekta. Destruktori lokalnih objekata izvršavaju se u obrnutom redosledu od konstruktora. Konstruktori globalnih objekata pozivaju se pre nego što počne izvršavanje funkcije `main()`. Globalni konstruktori izvršavaju se u redosledu deklarisanja unutar iste datoteke. Globalni destruktori izvršavaju se u obrnutom redosledu nakon završetka funkcije `main()`. Sledeći program ilustruje redosled pozivanja konstruktora i destruktora:

---

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

class mojaKlasa {
public:
 int ko;
 mojaKlasa(int id);
 ~mojaKlasa();
} glob_ob1(1), glob_ob2(2);

mojaKlasa::mojaKlasa(int id) {
 cout << "Inicijalizuje se " << id << endl;
 ko = id;
}

mojaKlasa::~mojaKlasa() {
 cout << "Unistava se " << ko << endl;
}

int main() {
 mojaKlasa lokalni_ob1(3);
 cout << "Ovo nece biti prvi prikazan red." << endl;
 mojaKlasa lokalni_ob2(4);
 return 0;
}
```

Rezultat izvršavanja:

```
Inicijalizuje se 1
Inicijalizuje se 2
Inicijalizuje se 3
Ovo nece biti prvi prikazan red.
Inicijalizuje se 4
Unistava se 4
Unistava se 3
Unistava se 2
Unistava se 1
```

---

### 6.5.2. Konstruktori sa parametrima

Konstruktorima je moguće proslediti argumente, koji se obično koriste za inicijalizaciju objekta tokom njegovog kreiranja. Konstruktor sa parametrima piše se kao i sve druge slične funkcije, a u telu konstruktora parametri se koriste za inicijalizaciju objekta. Sledeći primer ilustruje jednostavnu klasu koja ima konstruktor sa parametrima:

```
class mojaKlasa {
 int a, b;
 public:
 //inline konstruktor sa parametrima
 mojaKlasa(int i, int j) {
 a=i;
 b=j;
 }
 void prikazi() {
 cout << a << " " << b << endl;
 }
};

int main() {
 mojaKlasa ob(3, 5);
 ob.prikazi();
 return 0;
}
```

Primitite da se u definiciji konstruktora `mojaKlasa()` parametri `i` i `j` koriste za dodeljivanje početnih vrednosti promenljivama `a` i `b`. Ovaj program takođe ilustruje najčešće korišćen način za zadavanje argumenata konstruktora prilikom deklaracije objekta:

```
mojaKlasa ob(3, 4);
```

Ova naredba kreira objekat `ob`, a konstruktoru `mojaKlasa()` prosleđuje argumente 3 i 4. Parametri se mogu proslediti i sledećom deklaracijom:

```
mojaKlasa ob = mojaKlasa(3, 4);
```

Prvi metod se češće koristi i mi ćemo ga nadalje koristiti u ovoj knjizi.

Ako konstruktor ima samo jedan parametar, postoji i treći način za prosleđivanje vrednosti tom konstruktoru, koji ilustruje sledeći kôd:

```
class X {
 int a;
 public:
 X(int j) { a = j; }
 int get_a() { return a; }
};
```

```
int main() {
 X ob = 99; // prosledjuje 99 u j
 cout << ob.get_a(); // prikazuje 99
 return 0;
}
```

Konstruktor klase X prihvata jedan parametar; obratite pažnju na to kako je ob deklarisan u funkciji main(). Vrednost 99 se automatski prosleđuje parametru j u konstruktoru X(), odnosno kompajler tumači naredbu deklaracije objekta ob kao da je napisana ovako:

```
X ob = X(99);
```

Funkcijama članicama klase, pa i konstruktoru, mogu se dodeliti podrazumevane vrednosti argumenata. One se navode u deklaraciji konstruktora unutar klase, kao u sledećem primeru:

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

class Osoba {
 char *imePrezime;
 int godine;
public:
 Osoba(char *ime="Nikola Vukovic", int godine=40);
 void koSi();
};

Osoba::Osoba(char *i, int g) {
 imePrezime = i; godine = g;
}

void Osoba::koSi() {
 cout << imePrezime << endl;
}

int main() {
 Osoba o; //poziva konstruktor sa default argumentima
 o.koSi(); //ispisuje Nikola Vukovic
 return 0;
}
```

### 6.5.3. Preklapanje konstruktora

Konstruktori mogu da budu preklapljeni, i to je vrlo čest slučaj u klasama. Često su potrebne klase koje omogućuju više načina za kreiranje objekata. Ako se za svaki od načina kreiranja objekata obezbedi odgovarajući konstruktor, klasa postaje prilagodljivija jer korisnik može da izabere najbolji način za kreiranje objekta u skladu sa okolnostima. Proučimo primer klase datum sa dva konstruktora:

```

#include <iostream>
#include <cstdio>

using namespace std;

class datum {
 int dan, mesec, godina;
public:
 datum(char *d);
 datum(int d, int m, int g);
 void prikazi_datum();
};

//inicijalizacija pomocu stringa
datum::datum(char *d) {
 sscanf(d, "%d%c%d%c%d", &dan, &mesec, &godina);
}

//Inicijalizacija pomocu celih brojeva
datum::datum(int d, int m, int g) {
 dan = d;
 mesec = m;
 godina = g;
}

void datum::prikazi_datum() {
 cout << dan << "." << mesec;
 cout << "." << godina << endl;
}

int main() {
 datum ob1(12, 4, 2001), ob2("22.10.2010");
 ob1.prikazi_datum();
 ob2.prikazi_datum();
 return 0;
}

```

U ovom programu objekat tipa `datum` može da se inicijalizuje bilo tako što će se zadati tri cela broja koji predstavljaju dan, mesec i godinu, ili tako što će se navesti string sa datumom u obliku `dd.mm.gggg`. Pošto se za predstavljanje datuma koriste oba načina, ima smisla da klasa `datum` podržava oba načina kreiranja objekata. U sledećem programu `main()` od korisnika se traži da unese datum u obliku stringa, koji se zatim koristi direktno za kreiranje objekta klase `datum`. Kada klasa `datum` ne bi imala konstruktor koji prihvata string, uneti string bi morao ručno da se konvertuje u tri cela broja. Prilagodljivost klase koja se dobija pisanjem preklapljenih konstruktora posebno je bitna ako pišete biblioteke klase koje će koristiti drugi programeri.

```
int main() {
 char s[80];
 cout << "Unesite datum: ";
 cin >> s;
 datum d(s);
 d.prikazi_datum();
 return 0;
}
```

#### 6.5.4. Podrazumevani (default) konstruktori

Ako nijedan konstruktor nije eksplicitno deklarisan u definiciji klase, C++ kompajler će sam generisati tzv. podrazumevani (default) konstruktor. Podrazumevani konstruktor nema argumente i javan je. Ovakav konstruktor samo kreira objekat klase, ali ne inicijalizuje njegove podatke članove. Ako se za klasu deklarise neki konstruktor sa argumentima, tada podrazumevani konstruktor više neće biti dostupan, pa svaki deklarisan objekat klase mora da bude i inicijalizovan. Kada bi se u prethodnom primeru za klasu datum napisala deklaracija:

```
datum d;
```

ona bi prouzrokovala grešku jer za klasu datum kompajler više ne generiše podrazumevani konstruktor automatski. Oba preklapljeni konstruktora imaju argumente, pa ako želimo da omogućimo samo kreiranje objekata bez inicijalizacije, morali bismo eksplicitno da deklariramo još jedan konstruktor bez argumenata:

```
class datum {
 int dan, mesec, godina;
public:
 datum(char *d);
 datum(int d, int m, int g);
 datum(); //default konstruktor
 void prikazi_datum();
};
```

## 6.6. Zajednički članovi klase

Kada se ispred podatka člana klase navede ključna reč `static`, to je znak za kompajlera da treba da postoji samo jedna kopija tog podatka za sve objekte te klase koji će biti kreirani, odnosno da svi objekti te klase treba da dele taj podatak član. Sve statičke promenljive klase inicijalizuju se na 0 pre kreiranja prvog objekta. Kada se statički podatak član deklarise u klasi, on se istovremeno i ne definiše, tj. u memoriji se ne odvaja prostor za njega, već se definicija mora napisati izvan klase. Tek tada se alokira memorija za statički podatak član. Korišćenje statičkog podatka člana ilustrovano je u sledećem primeru:

```
#include <iostream>

using namespace std;

class deljena {
 static int a;
 int b;
 public:
 void set(int i, int j) {a=i; b=j;}
 void prikazi();
} ;

int deljena::a;

void deljena::prikazi() {
 cout << "staticko a: " << a << endl;
 cout << "nestaticko b: " << b << endl;
}

int main() {
 deljena x, y;
 x.set(1, 1); //a postaje 1
 x.prikazi();
 y.set(2, 2); //a postaje 2
 //a je promenjeno i za x i za y
 y.prikazi();
 x.prikazi();
 return 0;
}
```

Rezultat izvršavanja:

```
staticko a: 1
nestaticko b: 1
staticko a: 2
nestaticko b: 2
staticko a: 2
nestaticko b: 1
```

U prethodnom programu podatak član `a` je javni i statički, pa mu se može direktno pristupiti u funkciji `main()`. Takođe, pošto `a` postoji pre nego što se kreira bilo koji objekat klase `deljena`, promenljivoj `a` može se dodeliti vrednost bilo kad.

Sledeći program ilustruje činjenicu da vrednost podatka člana `a` ostaje neizmjenjena kada se kreira objekat `x`. Zbog toga obe naredbe `cout` prikazuju vrednost 99.

```
#include <iostream>

using namespace std;

class deljena {
public:
 static int a;
} ;

int deljena::a; // definise a

int main() {
 //inicijalizacija pre kreiranja bilo kog objekta
 deljena::a = 99;
 cout << "Pocetna vrednost a: " << deljena::a << endl;
 deljena x;
 cout << "x.a: " << x.a << endl;
 return 0;
}
```

Uočite kako se promenljivoj `a` pristupa preko imena klase i operatora razrešenja do-sega (`::`). U opštem slučaju, da bi se statičkom podatku članu pristupalo nezavisno od objekta, ispred njegovog imena mora se navesti ime klase čiji je član.

Statički podaci članovi često se koriste kada je potrebno osigurati pristup nekom resursu koji dele svi objekti klase, npr. istoj datoteci. U slučaju datoteke mora se osigurati da u određenom trenutku samo jedan objekat upisuje u nju; tada se deklariše statička promenljiva koja prati kada se datoteka koristi, a kada ne. Pre upisa u datoteku svaki objekat proverava stanje te statičke promenljive. Drugi primer je praćenje ukupnog broja objekata određene klase, što ilustruje sledeći primer:

---

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

class Brojac {
public:
 static int broj;
 Brojac() { broj++; } //konstruktor
 ~Brojac() { broj--; } //destruktor
};

int Brojac::broj;

void f();

int main() {
 Brojac o1;
 cout << "Ukupno objekata: ";
 cout << Brojac::broj << endl;
```

```

 Brojac o2;
 cout << "Ukupno objekata: ";
 cout << Brojac::broj << endl;
 f();
 cout << "Ukupno objekata: ";
 cout << Brojac::broj << endl;
 return 0;
}

void f() {
 Brojac temp;
 cout << "Ukupno objekata: ";
 cout << Brojac::broj << endl;
 //temp se unistava nakon izlaska iz f()
}

```

Rezultat izvršavanja:

```

Ukupno objekata: 1
Ukupno objekata: 2
Ukupno objekata: 3
Ukupno objekata: 2

```

Statički podatak član broj u prethodnom primeru se inkrementira kad god se objekat kreira, a dekrementira se kada se objekat uništava. Na taj način prati se koliko objekata klase Brojac trenutno postoji. Statički podaci članovi praktično uklanjaju potrebu za globalnim promenljivama, koje narušavaju objektno orijentisan pristup skrivanja (kapsuliranja) podataka.

I funkcije članice klase mogu da budu deklarisanе kao statičke. Statičke funkcije članice mogu direktno da pristupaju samo statičkim podacima članovima klase. U istoj klasi ne mogu da postoje dve verzije iste funkcije od kojih je jedna statička, a druga nestatička.

Statičke funkcije članice često se koriste za inicijalizaciju privatnih statičkih podataka članova pre kreiranja objekata klase, kao što ilustruje sledeći primer:

```

#include <iostream>

using namespace std;

class staticka {
 static int i;
public:
 static void init(int x) {i = x;}
 void prikazi() {cout << i << endl;}
};

int staticka::i; //definise i

```

```
int main() {
 //inicijalizacija statickih podataka
 //pre kreiranja objekata
 staticka::init(100);
 staticka x;
 x.prikazi(); //prikazuje 100
 return 0;
}
```

## 6.7. Prosleđivanje objekata funkciji

Objekti se mogu prosleđivati funkcijama na isti način kao što im se prosleđuju i druge promenljive, standardnim mehanizmom prenosa po vrednosti. To znači da se prilikom prosleđivanja argumenta funkciji pravi kopija objekta, odnosno kreira se drugi objekat. Postavlja se pitanje da li se prilikom kreiranja kopije objekta poziva konstruktor, odnosno da li se poziva destruktor kada se kopija objekta uništava. Da bismo odgovorili na to pitanje, proučimo sledeći primer:

---

```
#include <iostream>

using namespace std;

class mojaKlasa {
 int i;
public:
 mojaKlasa(int n);
 ~mojaKlasa();
 void set_i(int n) { i=n; }
 int get_i() { return i; }
};

mojaKlasa::mojaKlasa(int n) {
 i = n;
 cout << "kreira se " << i << endl;
}

mojaKlasa::~mojaKlasa() {
 cout << "unistava se " << i << endl;
}

void f(mojaKlasa ob);

int main() {
 mojaKlasa o(1);
 f(o);
 cout << "i u mainu: ";
```

```

 cout << o.get_i() << endl;
 return 0;
 }

 void f(mojaKlasa ob) {
 ob.set_i(2);
 cout << "i u funkciji: " << ob.get_i() << endl;
 }

```

Rezultat izvršavanja:

```

kreira se 1
i u funkciji: 2
unistava se 2
i u mainu: 1
unistava se 1

```

---

Primetite da se destruktork izvršava dva puta, iako se konstruktor poziva samo jednom. Kao što se vidi iz rezultata izvršavanja, konstruktor se ne poziva kada se kopija objekta `o` u funkciji `main()` prosleđuje u `ob` (u funkciji `f()`). Kada se objekat prosleđuje funkciji, ona treba da zna trenutno stanje tog objekta; kada bi se pozivao konstruktor, desila bi se inicijalizacija, tj. ne bi se prosledilo tekuće stanje objekta. Zbog toga se konstruktor ne poziva kada se u pozivu funkcije kreira kopija objekta, ali se zato poziva destruktork kada se ta kopija uništava. U opštem slučaju, kada se pravi kopija objekta, on se kopira bit po bit, da bi dobijeni objekat bio precizna kopija originalnog. To ponekad može da prouzrokuje probleme, na primer u slučaju kada objekat koji se koristi kao argument funkcije alocira i dealocira memoriju. Tada se dešava da tu istu memoriju oslobađa i kopija objekta prilikom poziva destruktorka, što oštećuje originalni objekat i čini ga neupotrebljivim. Da bi sprečio taj problem, za klasu se definiše specijalan konstruktor koji se zove konstruktor kopije; njime ćemo se pozabaviti u odeljku 6.11.

## 6.8. Dodela objekata jedan drugome

Pod pretpostavkom da su oba objekta istog tipa, oni se mogu dodeljivati jedan drugome. Time se podaci članovi objekta sa desne strane kopiraju u podatke članove objekta sa leve strane. Na primer, sledeći program prikazuje 99:

```

#include <iostream>

using namespace std;

```

```

class mojaKlasa {
 int i;
public:
 void set_i(int n) { i=n; }
 int get_i() { return i; }
};

int main() {
 mojaKlasa ob1, ob2;
 ob1.set_i(99);
 ob2 = ob1; //dodela ob1 objektu ob2
 cout << "ob2.i: " << ob2.get_i() << endl;
 return 0;
}

```

## 6.9. Pokazivači na objekte

Kao što mogu da postoje pokazivači na promenljive osnovnih tipova, postoje i pokazivači na objekte. Ako se koriste pokazivači na objekte, za pristup pojedinačnim članovima koristi se operator `->`. Tako je sintaksa `p -> funkcija()` ekvivalentna sa `(*p).funkcija()`. Zagrade oko `*p` su neophodne zato što operator `.` ima veći prioritet od operatora dereferenciranja `*`. Adresa objekta se dobija primenom operatora `&`, kao da se radi o običnoj promenljivoj prostog tipa. Takođe, reference na objekte deklariraju se na isti način kao reference na obične promenljive.

Sledeći primer ilustruje kako se objektu pristupa preko pokazivača:

```

class cl {
 int i;
public:
 cl(int j) { i=j; }
 int get_i() { return i; }
};

int main() {
 cl ob(88), *p;
 p = &ob; //p pokazuje na ob
 cout << p->get_i() << endl; //prikazuje 88
 return 0;
}

```

## 6.10. Pokazivač this

Već smo naučili da se funkcija članica uvek poziva za neki objekat svoje klase (osim ako nije statička). Kada se funkcija članica pozove, prosleđuje joj se jedan implicitni argument: pokazivač na objekat za koji je pozvana. Pokazivač na objekat za

koji je pozvana funkcija članica zove se `this`. Da bismo razumeli pokazivač `this`, proučimo program sa klasom `pwr` za stepenovanje brojeva:

```
#include <iostream>

using namespace std;

class pwr {
 double b;
 int e;
 double val;
public:
 pwr(double base, int exp);
 double get_pwr() { return val; }
};

pwr::pwr(double base, int exp) {
 b = base;
 e = exp;
 val = 1;
 if(exp==0)
 return;
 for(; exp>0; exp--)
 val = val * b;
}

int main() {
 pwr x(4.0, 2), y(2.5, 1), z(5.7, 0);
 cout << x.get_pwr() << " ";
 cout << y.get_pwr() << " ";
 cout << z.get_pwr() << endl;
 return 0;
}
```

Kada u funkciji članici koristimo direktno neki podatak član pristupom preko imena, koristimo sintaksu:

```
b = base;
```

kompajler automatski razume da koristimo `(*this).ime_člana`, što se u ovom primeru kraće piše kao

```
this->b = base;
```

Pokazivač `this` pokazuje na objekat koji je pozvao funkciju članicu `pwr()`. Funkcija `pwr()` bi mogla da se napiše i ovako:

```
pwr::pwr(double base, int exp) {
 this->b = base;
 this->e = exp;
 this->val = 1;
 if(exp==0)
```

```

 return;
 for(; exp>0; exp--)
 this->val = this->val * this->b;
}

```

Nijedan C++ programer ne bi funkciju `pwr()` napisao ovako, jer je standardni oblik bez pokazivača `this` čitljiviji. Međutim, ovaj pokazivač postaje veoma važan kad god funkcije članice moraju da koriste pokazivač na objekat koji ih je pozvao. Treba primetiti takođe da statičke funkcije članice nemaju pokazivač `this` jer se ne pozivaju za objekte klase.

## 6.11. Konstruktor kopije

Jedan od najvažnijih konstruktora je tzv. *konstruktor kopije* (copy constructor). Da bismo razumeli njegovu namenu, treba razumeti problem koji on rešava. Već smo pominjali da u situacijama kada se jedan objekat koristi za inicijalizaciju drugog C++ obavlja kopiranje bit po bit. To je u nekim slučajevima sasvim prihvatljivo, i upravo ono što i želimo da se desi. Međutim, postoje situacije kada to nije prikladno, npr. kada objekat dinamički alokira memoriju ili ima podatke članove koji su pokazivači.

Na primer, zamislimo klasu `mojaKlasa` koja alokira memoriju za objekat kada se on kreira, i neki objekat `A` te klase. Ako je objekat kreiran, to znači da je već alocirao memoriju. Sada zamislimo situaciju u kojoj se objekat `A` koristi za inicijalizaciju objekta `B` iste klase, kao u sledećoj naredbi:

```
mojaKlasa B = A; //objekat B se kreira kao identična kopija objekta A
```

Ako se obavlja kopija bit po bit, onda je objekat `B` identična kopija objekta `A`, što znači da će i objekat `B` koristiti istu memoriju koju je alocirao objekat `A`, umesto da alokira memoriju za sebe. Očigledno je da to nije ono što želimo, jer ako `mojaKlasa` ima destruktork koji oslobađa memoriju, onda će ista memorija biti oslobođena dva puta: jednom kada se uništava `A`, i drugi put kada se uništava `B`!

Isti problem se javlja i ako klasa ima podatak član koji je pokazivačkog tipa. Pošto se pravi kopija bit po bit, onda će i kopija objekta pokazivati na istu promenljivu u memoriji, umesto da pokazuje na "svoju" kopiju promenljive.

Da bi se rešio opisani problem, C++ omogućuje pisanje konstruktora kopije, koji kompajler poziva kad god se jedan objekat inicijalizuje drugim. Konstruktor kopije se deklarise na sledeći način:

```

ime_klase (const ime_klase &o) {
 // telo konstruktora
}

```

Argument konstruktora kopije je konstantna referenca na objekat klase. Modifikator `const` se navodi zato što nema razloga da konstruktor kopije menja objekat koji mu se prosleđuje kao argument, tj. objekat koji se koristi za inicijalizaciju. Konstruktor

kopije može da ima i druge parametre pod uslovom da su za njih definisane podrazumevane vrednosti (ali u svakom slučaju prvi argument mora da bude referenca na objekat koji se koristi za inicijalizaciju).

Važno je razumeti da C++ razlikuje dve situacije u kojima se vrednost jednog objekta dodeljuje drugom. Prvi je dodela, a drugi je inicijalizacija, koja se dešava u tri slučaja:

- kada jedan objekat eksplicitno inicijalizuje drugi
- kada se pravi privremeni objekat
- kada se pravi kopija objekta koja se prosleđuje funkciji

Konstruktor kopije poziva se samo u inicijalizacijama, ne i u dodelama. Na primer, konstruktor kopije bi se pozivao u sledećim naredbama:

```
mojaKlasa x = y; // y eksplicitno inicijalizuje x
funkc(y); // y se prosleđuje kao parametar
a ne bi se pozvao u sledećoj naredbi dodele:
mojaKlasa x, y;
x = y;
```

Argument konstruktora kopije ne može da bude sam objekat, već uvek referenca na objekat. Kada bi se objekat prosleđivao funkciji po vrednosti, pozivao bi se konstruktor kopije za njega jer se pravi kopija objekta. Zbog toga se u konstruktoru kopije ne može koristiti prenos po vrednosti, već po adresi (tj. preko reference) u suprotnom bi se pojavila rekurzija jer bi se konstruktor kopije pozivao unedogled.

U nastavku je dat primer kada je potrebno eksplicitno napisati konstruktor kopije jer klasa ima podatak član koji je pokazivač. Unutar konstruktora dinamički se alokira memorija za podatak član ime korišćenjem operatora `new`. Memorija se oslobađa pozivom operatora `delete` unutar destruktora.

```
class Osoba {
 char *ime;
 int godine;
public:
 Osoba(char *i, int g) {
 ime = new char[strlen(i)+1];
 strcpy(ime, i);
 godine = g;
 }
 ~Osoba() { delete[] ime; }
};
```

Konstruktor kopije za klasu `Osoba` dinamički alokira prostor u memoriji za niz znakove dužine podatka člana `ime` koji se prosleđuje kao argument:

```
Osoba(const Osoba &osoba) {
 ime = new char[strlen(osoba.ime)+1];
 strcpy(ime, osoba.ime);
 godine = osoba.godine;
}
```

## 6.12. Klasa string

U zaključku poglavlja o klasama, proučićemo veoma korisnu klasu `string` koja je ugrađena u jezik C++. Klasa `string` nudi brojne korisne funkcije, a poziva se navođenjem zaglavlja `<string>`. Da bi se koristili `string` objekti, nije potrebno poznavati način kako je ta klasa implementirana, već samo njene metode.

Rad sa `string` objektima sličan je radu sa drugim promenljivama osnovnih tipova. `String` objekat se deklarise na uobičajeni način:

```
string primer;
```

Postoji nekoliko načina za inicijalizovanje `string` objekata, prikazanih u sledećoj tabeli:

| Tabela 6.1: Inicijalizacija objekata klase <code>string</code> . |                                                                                                                                                                                                                  |
|------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>string adresa;</code>                                      | Deklariše prazan <code>string</code> <code>adresa</code>                                                                                                                                                         |
| <code>string ime("Nikola Vukovic");</code>                       | Deklariše <code>string</code> objekat <code>ime</code> i inicijalizuje ga sa "Nikola Vukovic"                                                                                                                    |
| <code>string osoba2(osoba1)</code>                               | Deklariše <code>string</code> objekat <code>osoba1</code> , koji je kopija <code>string</code> objekta <code>osoba2</code> , pri čemu <code>osoba2</code> može da bude drugi <code>string</code> ili niz znakova |
| <code>string set1(set2, 5);</code>                               | Deklariše <code>string</code> objekat <code>set1</code> koji se inicijalizuje sa prvih pet znakova niza znakova <code>set2</code>                                                                                |
| <code>string punRed('z', 10);</code>                             | Deklariše <code>string</code> objekat <code>punRed</code> koji se inicijalizuje sa 10 znakova 'z'                                                                                                                |
| <code>string ime(punoIme, 0, 7);</code>                          | Deklariše <code>string</code> objekat <code>ime</code> koji se inicijalizuje podstringom <code>stringa</code> <code>punoIme</code> . Podstring ima 7 znakova i počinje od pozicije 0.                            |

Objekti tokova `cin` i `cout` automatski podržavaju `string` objekte:

```
string ime;
cout << "Kako se zovete?";
cin >> ime;
cout << "Dobar dan, " << ime << endl;
```

Za učitavanje celog reda sa konzole (zajedno sa razmacima) koristi se funkcije `getline()`. Prvi argument funkcije `getline` je objekat ulaznog toka, a drugi je objekat klase `string` u koji se smešta ulaz sa konzole:

```
string ime;
cout << "Kako se zovete?";
getline (cin, ime);
```

Za poredenje stringova mogu se koristiti operatori `<`, `<=`, `>`, `>=` i `==`. Relacioni operatori porede objekte klase `string` na sličan način kao što funkcija `strcmp` poredi C-stringove (znakovne nizove koji se završavaju nulom).

Takođe, operatori `+` i `+=` mogu se koristiti za nadovezivanje (konkatenaciju) stringova, kao u sledećem primeru:

```
string str1, str2, str3;
str1='ABC';
str2='DEF';
str3 = str1 + str2;
cout << str3; //ispisuje ABCDEF
str3+='GHI'
cout<<str3; //ispisuje ABCDEFGHI
```

Klasa `string` takođe nudi brojne korisne funkcije članice, delimično navedene u sledećoj tabeli:

| <b>Tabela 6.2: Funkcije članice klase <code>string</code>.</b> |                                                                                                                                                                                                                       |
|----------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>str1.append(str2)</code>                                 | Dodaje <code>str2</code> na <code>str1</code> . <code>str2</code> može biti string objekat ili niz znakova.                                                                                                           |
| <code>str1.append(str2, x, n)</code>                           | <code>n</code> znakova objekta <code>str2</code> počev od pozicije <code>x</code> dodaju se objektu <code>str1</code> . Ako <code>str1</code> nema dovoljnu dužinu, kopiraće se onoliko znakova koliko može da stane. |
| <code>str1.append(str2, n);</code>                             | Prvih <code>n</code> znakova niza <code>str2</code> dodeljuju se <code>str1</code>                                                                                                                                    |
| <code>str.append(n, 'z')</code>                                | <code>n</code> kopija znaka <code>'z'</code> dodeljuje se objektu <code>str</code>                                                                                                                                    |
| <code>str1.assign(str2)</code>                                 | Dodeljuje <code>str2</code> objektu <code>str1</code> . <code>str2</code> može da bude string objekat ili niz znakova.                                                                                                |
| <code>str1.assign(str2, x, n)</code>                           | <code>n</code> znakova objekta <code>str2</code> počev od pozicije <code>x</code> dodeljuje se <code>str1</code> . Ako <code>str1</code> nema dovoljnu dužinu, kopiraće se onoliko znakova koliko može da stane.      |
| <code>str1.assign(str2, n)</code>                              | Prvih <code>n</code> znakova <code>str2</code> dodeljuje se objektu <code>str1</code> .                                                                                                                               |
| <code>str.assign(n, 'z')</code>                                | Dodeljuje <code>n</code> kopija znaka <code>'z'</code> objektu <code>str</code> .                                                                                                                                     |
| <code>str.at(x)</code>                                         | Vraća znak na poziciji <code>x</code> u objektu <code>str</code> .                                                                                                                                                    |

| <b>Tabela 6.2:</b> Funkcije članice klase string. |                                                                                                                                                                                                                                                        |
|---------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>str.capacity()</code>                       | Vraća veličinu memorije koja je alocirana za string.                                                                                                                                                                                                   |
| <code>str.clear()</code>                          | Briše sve znakove u stringu <code>str</code> .                                                                                                                                                                                                         |
| <code>str1.compare(str2)</code>                   | Poredi stringove kao funkcija <code>strcmp</code> za C stringove, sa istim povratnim vrednostima. <code>str2</code> može da bude niz znakova ili string objekat.                                                                                       |
| <code>str1.compare(x, n, str2)</code>             | Poredi stringove <code>str1</code> i <code>str2</code> počev od pozicije <code>x</code> narednih <code>n</code> znakova. Povratna vrednost je ista kao u funkciji <code>strcmp</code> . <code>str2</code> može da bude string objekat ili niz znakova. |
| <code>str1.copy(str2, x, n)</code>                | Kopira <code>n</code> znakova niza znakova <code>str2</code> u <code>str1</code> počev od pozicije <code>x</code> . Ako <code>str2</code> nije dovoljno dugačak, funkcija kopira onoliko znakova koliko može da stane.                                 |
| <code>str.data()</code>                           | Vraća niz znakova sa nulom na kraju, kao u <code>str</code>                                                                                                                                                                                            |
| <code>str.empty()</code>                          | Vraća <code>true</code> ako je <code>str</code> prazan.                                                                                                                                                                                                |
| <code>str.erase(x, n)</code>                      | Briše <code>n</code> znakova iz objekta <code>str</code> počev od pozicije <code>x</code> .                                                                                                                                                            |
| <code>str1.find(str2, x)</code>                   | Vraća prvu poziciju iza pozicije <code>x</code> gde se string <code>str2</code> nalazi u <code>str1</code> . <code>str2</code> može da bude string objekat ili niz znakova.                                                                            |
| <code>str.find('z', x)</code>                     | Vraća prvu poziciju iza pozicije <code>x</code> na kojoj se znak <code>'z'</code> nalazi u <code>str1</code>                                                                                                                                           |
| <code>str1.insert(x, str2)</code>                 | Umeće kopiju <code>str2</code> u <code>str1</code> počev pod pozicije <code>x</code> . <code>str2</code> može da bude string objekat ili niz znakova.                                                                                                  |
| <code>str.insert(x, n, 'z')</code>                | Umeće <code>'z'</code> <code>n</code> puta u <code>str</code> počev od pozicije <code>x</code>                                                                                                                                                         |
| <code>str.length()</code>                         | Vraća dužinu stringa <code>str</code>                                                                                                                                                                                                                  |
| <code>str1.replace(x, n, str2)</code>             | Zamenjuje <code>n</code> znakova u <code>str1</code> počev od pozicije <code>x</code> znakovima iz string objekta <code>str2</code>                                                                                                                    |

| <b>Tabela 6.2:</b> Funkcije članice klase string. |                                                                                                                          |
|---------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------|
| <code>str.size()</code>                           | Vraća dužinu stringa <code>str</code>                                                                                    |
| <code>str.substr(x, n)</code>                     | Vraća kopiju podstringa dugačkog <code>n</code> znakova koji počinje na poziciji <code>x</code> objekta <code>str</code> |
| <code>str1.swap(str2)</code>                      | Zamenjuje sadržaj <code>str1</code> sa <code>str2</code>                                                                 |

Sledi primer koji ilustruje korišćenje funkcija članica klase string:

```
string str1 = "ovo je probni string";
int pozicija = str1.find("probni"); //daje 7
pozicija = str1.find("bla"); //vraca -1
string str2 = str1.substr(7, 2); //pr
str1.replace(4, 2, "vise nije"); //ovo vise nije probni string
str1.erase(4, 7); //ovo je probni string
```

## Pitanja

1. Kapsuliranje je \_\_\_\_\_ podataka.

- a) skrivanje
- b) apstrakcija
- c) nasleđivanje

2. Uočavanje zajedničkih svojstava objekata i njihovo grupisanje u klasu naziva se:

- a) apstrakcija
- b) kapsuliranje
- c) polimorfizam

3. Objekat je primerak (instanca) neke:

- a) klase
- b) promenljive
- c) metode

4. Za definisanje klase koristi se ključna reč:

- a) method
- b) class
- c) main
- d) object

5. Privatni članovi klase moraju se deklarirati pre javnih.

- a) tačno
- b) pogrešno

6. Javnim podacima članovima objekta izvan objekta pristupa se pomoću:

- a) bilo koje funkcije članice
- b) operatora .
- c) operatora razrešenja dosega (::)

7. Sakrivanje detalja realizacije klase zove se:

- a) apstrakcija
- b) kapsuliranje
- c) polimorfizam

8. Javni članovi klase deklariraju se kao:

- a) public
- b) protected
- c) private

9. Objekti klase:

- a) imaju sopstvene kopije vrednosti podataka članova
- b) dele zajedničke vrednosti podataka članova

10. Unutar deklaracije klase navode se:

- a) definicije funkcija članica
- b) prototipovi (deklaracije) funkcija članica

11. Specijalna funkcija članica koja se poziva prilikom kreiranja objekta klase zove se:

- a) glavni metod
- b) metod bez argumenata
- c) konstruktor
- d) rekurzivni metod

12. U jeziku C++ postoji funkcija bez povratnog tipa i argumenata.

- a) tačno
- b) pogrešno

13. Podaci članovi ili funkcije članice deklarirani nakon ključne reči `private` dostupni su funkcijama članicama klase u kojoj su deklarirani.

- a) tačno
- b) pogrešno

14. Promenljive deklarirane u telu neke funkcije članice zovu se podaci članovi i mogu se koristiti u svim funkcijama članicama klase.

- a) tačno
- b) pogrešno

15. Članovi klase deklarirani kao \_\_\_\_\_ dostupni su svuda gde je objekat klase dostupan.

- a) `public`
- b) `protected`
- c) `private`

16. Operator \_\_\_\_\_ može se koristiti za dodelu objekta neke klase drugom objektu iste klase.

- a) `=`
- b) `==`
- c) konstruktor

17. Ključna reč \_\_\_\_\_ označava da se objekat ili promenljiva ne mogu menjati nakon inicijalizacije.

- a) `const`
- b) `static`
- c) `volatile`

18. Pod pretpostavkom da je `sapun` objekat klase `Kozmetika`, koji od sledećih je važeći poziv funkcije članice `operi()`?

- a) `operi();`
- b) `sapun::operi();`
- c) `sapun.operi();`
- d) `sapun:operi();`

19. Ključna reč `inline`:

- a) ugrađuje telo funkcije neposredno u kôd na mestu pozivanja
- b) umeće ceo objekat u jedan red
- c) izvršava kod jednog reda

20. Operator :: služi za:

- a) pristup objektu
- b) razrešenje dosega
- c) poziv metode

21. Operator strelica -> :

- a) pokazuje na objekat
- b) pokazuje na operator množenja
- c) pristupa članu objekta preko pokazivača

22. Podrazumevani (default) konstruktor:

- a) mora da ima argumente
- b) ne sme da ima argumente
- c) može, ali ne mora da ima argumente

23. Klasa može da ima samo jedan konstruktor.

- a) tačno
- b) pogrešno

24. Destruktor klase može da ima argumente.

- a) tačno
- b) pogrešno

25. Klasa može da ima više preklopljenih destruktora.

- a) tačno
- b) pogrešno

26. Svaka klasa ima konstruktor i destruktor.

- a) tačno
- b) pogrešno

27. Klasa može imati više preklopljenih konstruktora.

- a) tačno
- b) pogrešno

28. Ako je funkcija i definisana, a ne samo deklarirana unutar definicije klase, ona je:

- a) preklopljena
- b) inline
- c) statička

29. Skriveni pokazivač na objekat za koji je pozvana funkcija članica zove se `this`.

- a) tačno
- b) pogrešno

30. Moguće je dodeljivati objekte iste klase jedan drugome.

- a) tačno
- b) pogrešno

31. Destruktor klase `Primer` je:

- a) `Primer::Primer();`
- b) `Primer::~~Primer();`

32. Potrebno je eksplicitno pozivati destruktora za lokalnu promenljivu.

- a) tačno
- b) pogrešno

33. Za svaku klasu mora se napisati konstruktor kopije.

- a) tačno
- b) pogrešno

34. Ako se za klasu napiše konstruktor sa argumentima, podrazumevani (default) konstruktor više nije dostupan.

- a) tačno
- b) pogrešno

35. Konstruktor kopije se poziva u slučaju kada se objekat klase inicijalizuje drugim objektom.

- a) tačno
- b) pogrešno

37. Za klasu `string` mora se uključiti zaglavlje:

- a) `cstring`
- b) `string`
- c) `iostream`

38. Funkcija \_\_\_\_\_ briše znakove iz stringa.

- a) erase
- b) delete
- c) at

39. Funkcija \_\_\_\_\_ pronalazi prvo pojavljivanje bilo kog znaka u stringu.

- a) at
- b) find
- c) open

40. Konkatenacija (nadovezivanje) string objekata može se postići i pomoću operatora +=.

- a) tačno
- b) pogrešno





## Glava 7

# Nasleđivanje, virtuelne funkcije i polimorfizam

U ovom poglavlju bavićemo se osobinama jezika C++ koje ga čine objektno orijentisanim: nasleđivanjem, virtuelnim funkcijama i polimorfizmom. Nasleđivanje omogućuje izvođenje klasa, odnosno omogućuje jednoj klasi da nasledi osobine druge. Uz pomoć nasleđivanja možemo da uvedemo hijerarhiju u projektovanje: definišemo opštu klasu koja ima zajedničke osobine nekog pojma, iz koje izvodimo specifične klase sa posebnim svojstvima. Na nasleđivanje se oslanjaju virtuelne funkcije pomoću kojih je implementiran polimorfizam, odnosno objektno orijentisana filozofija "jedan interfejs, više metoda".

### 7.1. Nasleđivanje

Klasa iz koje se izvode druge klase u jeziku C++ zove se *osnovna klasa* (base class). Klase koje nasleđuju osnovnu klasu zovu se *izvedene klase* (derived class). Izvedena klasa je specijalan slučaj osnovne klase. Izvedena klasa nasleđuje sve podatke članove osnovne klase, ali joj dodaje i neke svoje, posebne. U jeziku C++ nasleđivanje je implementirano tako što se u deklaraciju izvedene klase ugrađuje osnovna klasa. Počecemo sa jednostavnim primerom koji ilustruje najvažnije osobine nasleđivanja. Sledeći primer kreira osnovnu klasu DvaDOblik u kojoj se čuvaju visina i širina dvodimenzionalnog objekta, i iz nje izvodi klasu Trougao:

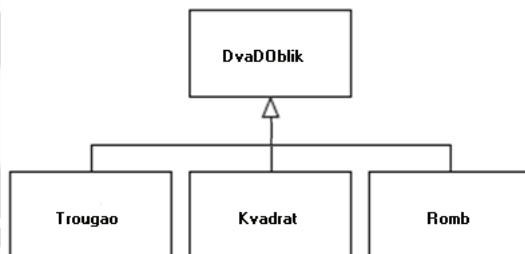
```
class DvaDOblik {
public:
 double visina;
 double sirina;
 void prikaziDimenzije() { cout << "sirina: " << sirina << ", "
 << "visina: " << visina << endl; }
};

class Trougao: public DvaDOblik {
public:
 string vrsta;
 double površina() { return sirina * visina / 2; }
};
```

Klasa `DvaDOblik` definiše svojstva opšteg dvodimenzionalnog oblika, a to može biti kvadrat, pravougaonik, trougao, romb itd. Klasa `Trougao` je specifična vrsta dvodimenzionalnog oblika koja ima sve njegove osobine (širinu i visinu), ali i još nešto što je specifično za trougao. To su podatak član vrsta (koji opisuje da li je trougao jednakostranični, pravougli, jednakokraki) i funkcija članica `povrsina()` koja racuna površinu trougla. Kao što ovaj primer pokazuje, sintaksa izvođenja klase je prilično jednostavna:

```
class izvedena_klasa : specifikator_izvodjenja osnovna_klasa {
 // telo izvedene klase
}
```

Specifikator izvođenja nije obavezan, ali ako se navede, mora biti jedna od vrednosti `public`, `private` ili `protected`. Detalje o tome saznaćemo u nastavku poglavlja, a za sada treba znati da kada se koristi specifikator `public` (što je najčešće slučaj), svi javni članovi osnovne klase biti javni i u izvedenim klasama.



Slika 7.1: Hijerarhija izvođenja klase.

Pošto klasa `Trougao` sadrži sve članove svoje osnovne klase `DvaDOblik`, može da im pristupi unutar funkcije članice `povrsina()`. Takođe, iako je `DvaDOblik` osnovna klasa za `Trougao`, to je potpuno nezavisna klasa koja može samostalno da se koristi.

Na sličan način kao klasa `Trougao`, iz klase `DvaDOblik` mogla bi se izvesti klasa `Pravougaonik` sa metodama `kvadrat()`, koji određuje da li je pravougaonik kvadrat i `povrsina()`, koji računa površinu pravougaonika:

```
class Pravougaonik: public DvaDOblik {
public:
 bool kvadrat() {
 if(duzina == sirina)
 return true;
 else
 return false;
 }
 double povrsina() { return sirina * visina; }
};
```

Najveća prednost nasleđivanja je u tome što kada se jednom kreira osnovna klasa koja definiše svojstva zajednička za skup objekata, ona se dalje može koristiti da bi se iz nje izveo proizvoljan broj specifičnih klasa. Svaka izvedena klasa može precizno da prilagodi svoja svojstva, kao što smo videli u slučaju metode `povrsina()` koja je različita za klase `Trougao` i `Pravougaonik`. Time se pojednostavljuje održavanje programa, a istovremeno omogućuje ponovno korišćenje kôda. Ako je potrebno promeniti neku osobinu klase, neće se menjati osnovna klasa, već će iz nje biti izvedena klasa koja opisuje ta specijalna svojstva. Time će postojeći kôd ostati nepromenjen.

## 7.2. Kontrola pristupa članovima klase

Već smo naučili da se podaci članovi klase po pravilu deklariraju kao privatni da bi se sprečilo njihovo neovlašćeno korišćenje ili menjanje. Nasleđivanjem ne mogu da se naruše ograničenja pristupa članovima osnovne klase koja su već definisana. Tako, iako izvedena klasa nasleđuje sve članove iz osnovne, ona ne može da pristupi članovima osnovne klase koji su deklarirani kao privatni. U prethodnom primeru, da su podaci članovi `visina` i `sirina` u klasi `DvaDOblik` bili deklarirani kao privatni, metod `povrsina()` klasa `Trougao` i `Pravougaonik` ne bi mogao da im pristupi.

Rešenje ovog ograničenja nije u tome da se podaci članovi učine javnim kao u prethodnom primeru, jer tako osim izvedenoj klasi postaju dostupni i drugom kôdu. Postoje dva rešenja za ovaj problem: prvo je da se koristi specifikator pristupa `protected`, a drugo da se koriste javne funkcije koje će omogućiti pristup privatnim podacima članovima. Proučicemo oba pristupa.

### 7.2.1. Specifikator pristupa `protected`

Zaštićeni član osnovne klase je isti kao privatni, uz jedan važan izuzetak: mogu da mu pristupe izvedene klase. Zaštićeni član se deklarira pomoću ključne reči `protected`. Tako pomoću modifikatora `protected` mogu da se kreiraju podaci članovi klase koji su privatni, ali im mogu pristupiti izvedene klase. Proučimo sledeći primer:

```
class osnovna {
private:
 int privatan;
protected:
 int zasticen;
public:
 int javan;
};
```

```

class izvedena: public osnovna {
public:
 void prikazi() {
 cout << zasticen << endl;
 cout << javan << endl;
 cout << privatan; //greska!
 }
};

```

Da su u prethodnom primeru podaci članovi `privatan` i `zasticen` bili deklarirani kao `private`, izvedena klasa ne bi mogla da im pristupi i kôd se ne bi kompajlirao. Specifikator pristupa `protected` može da se navede bilo gde u deklaraciji klase, a najčešće se navodi posle (podrazumevanih) privatnih članova, a pre javnih. Zbog toga je najčešći oblik deklaracije klase sledeći:

```

class ime_klase{
 //privatni clanovi
protected:
 //zasticeni clanovi
public:
 //javni clanovi
};

```

### 7.2.2. Javno i privatno izvođenje

Kada se klasa izvodi iz osnovne klase sa specifikatorom pristupa `public`, sledećom sintaksom:

```

class izvedena: public osnovna {
 //telo izvedene klase
};

```

zaštićeni članovi osnovne klase ostaju zaštićeni i u izvedenoj klasi. Kada se klasa izvodi sa specifikatorom pristupa `private`, zaštićeni članovi osnovne klase postaju privatni u izvedenoj klasi. U opštem slučaju, nasleđivanjem dostupnost podataka članova može eventualno ostati na istom nivou, a nikako se ne može povećati.

Javno izvođenje realizuje pojam nasleđivanja u opštem smislu: javni članovi osnovne klase jesu javni članovi i izvedene klase, pa korisnici izvedene klase mogu da “vide” sve osobine osnovne klase i u izvedenoj klasi. U tom slučaju, izvedena klasa je vrsta (a kind of) osnovne klase.

Privatno izvođenje realizuje pojam sadržavanja (kompozicije): javni članovi osnovne klase su sakriveni u izvedenoj klasi, tj. objekti izvedene klase sadrže objekat osnovne klase u sebi. Osnovna klasa je deo (a part of) izvedene klase. Ovakva vrsta izvođenja može se realizovati i članstvom objekata, tj. kompozicijom. Tako je deklaracija:

```
class izvedena: private osnovna {
 //telo izvedene klase
};
```

funkcionalno identična sa

```
class izvedena {
private:
 osnovna o;
 //telo klase
};
```

Specifikator izvođenja nije obavezno navesti u deklaraciji klase, a ako se ne navede, podrazumeva se privatno izvođenje.

Postoji i zaštićeno izvođenje, kada se kao specifikator izvođenja navodi ključna reč `protected`. Kada se klasa izvede kao zaštićena, svi javni i zaštićeni članovi osnovne klase postaju zaštićeni članovi u izvedenoj klasi. Zaštićeno izvođenje se skoro nikad ne koristi u praksi jer nema uporišta u stvarnim problemima objektno orijentisanog projektovanja.

### 7.2.3. Inspektori i mutatori

C++ programeri obično pristupaju privatnim podacima članovima klase preko javnih funkcija članica. Funkcije koje samo čitaju vrednosti podataka članova, a ne menjaju ih, zovu se *inspektorske funkcije članice* (accessor functions). Slično, funkcije članice koje menjaju vrednost svojih podataka članova zovu se *mutatorske funkcije* (mutator functions). Ovako bi izgledala klasa `DvaDOblik` sa dodatim inspektorima i mutatorima funkcijama za širinu i visinu:

```
class DvaDOblik {
private:
 double visina;
 double sirina;
public:
 void prikaziDimenzije() {
 cout << "sirina: " << sirina << ", "
 << "visina: " << visina << endl;
 }
 //inspektori
 double getVisina() const { return visina;}
 double getSirina() const { return sirina;}
 //mutatori
 void setSirina(double s) { sirina = s;}
 void setVisina(double v) { visina = v;}
};
```

Pošto su `setVisina()`, `setSirina()` i `prikaziDimenzije()` javne funkcije članice klase `DvaDOblik`, mogu se pozvati za objekat tog tipa iz funkcije `main()`. U isto vreme, pošto su podaci članovi `sirina` i `visina` privatni, oni ostaju privatni i u izvedenoj klasi i ona ne može da im pristupa.

Da bismo rekli kompajleru da je metod inspektorski i da treba da mu bude dozvoljeno samo da radi sa objektima, ali da ih ne menja, taj metod treba da bude deklarisan kao konstantan. Konstantan metod (inspektor) deklarise se dodavanjem ključne reči `const` iza zagrada i u deklaraciji i u definiciji funkcije članice, kao u prethodnom primeru funkcija `getVisina()` i `getSirina()`. Deklarisanje funkcija članica kao konstantnih nije obavezno, ali je dobra programerska praksa, jer kompajler proverava da li je konstantnost ispunjena. To je zapravo “obećanje” projektanta klase korisnicima te klase da neka funkcija neće menjati stanje objekta.

### 7.3. Konstruktori i destruktori izvedenih klasa

U hijerarhiji klasa, moguće je (i najčešće potrebno) da i osnovna klasa i izvedene klase imaju svoje konstruktore. Postavlja se pitanje koji konstruktor je odgovoran za kreiranje objekta izvedene klase. Odgovor je da konstruktor osnovne klase kreira deo objekta koji pripada osnovnoj klasi, a konstruktor izvedene klase kreira samo deo objekta iz izvedene klase. To ima smisla zato što osnovna klasa ne zna ništa o svojstvima objekta u izvedenoj klasi. U ovom odeljku objasnićemo kako C++ upravlja situacijama u kojima postoje konstruktori osnovne i izvedene klase.

Kada samo izvedena klasa ima konstruktor, rešenje je jednostavno: poziva se konstruktor za izvedenu klasu, a deo objekta osnovne klase kreira se automatski, pozivom podrazumevanog (default) konstruktora osnovne klase. Na primer, evo prepravljene verzije klase `Trougao` u kojoj je definisan konstruktor:

```
#include <iostream>
#include <string>

using namespace std;

class DvaDOblik {
private:
 double visina;
 double sirina;
public:
 void prikaziDimenzije() {
 cout << "sirina: " << sirina << ", " <<
 "visina: " << visina << endl;
 }
 double getVisina() const { return visina;}
 double getSirina() const { return sirina;}
```

```

 void setSirina(double s) { sirina = s;}
 void setVisina(double v) { visina = v;}
};

class Trougao: public DvaDOblik {
public:
 string vrsta;
 double površina() { return getSirina() * getVisina() / 2; }
 Trougao(string tip, double v, double s) {
 //inicijalizacija osnovnog objekta
 setSirina(s);
 setVisina(v);
 //inicijalizacija izvedenog dela
 vrsta = tip;
 }
 void prikaziVrstu() {
 cout << "Trougao je: " << vrsta << endl;
 }
};

```

Kada su i u osnovnoj i u izvedenoj klasi definisani konstruktori, postupak je složeniji. Kada osnovna klasa ima konstruktor, izvedena klasa mora eksplicitno da ga pozove da bi inicijalizovao deo objekta iz osnovne klase. Izvedena klasa poziva konstruktor svoje osnovne klase specijalnim oblikom deklaracije u konstruktoru izvedene klase:

```

ime_izvedene_klase(lista_argumenata) :
 konstruktor_osnovne_klase(lista_argumenata); {
 //telo konstruktora izvedene klase
}

```

Primitite dvotačku koja razdvaja deklaraciju konstruktora izvedene klase od poziva konstruktora osnovne klase. Sledeći program ilustruje kako se prosleđuju argumenti konstruktoru osnovne klase:

---

```

#include <iostream>
#include <string>

using namespace std;

class DvaDOblik {
private:
 double visina;
 double sirina;
public:
 void prikaziDimenzije() {
 cout << "sirina: " << sirina << ", " <<
 "visina: " << visina << endl;
 }
};

```

```

 }
 double getVisina() const { return visina;}
 double getSirina() const { return sirina;}
 void setSirina(double s) { sirina = s;}
 void setVisina(double v) { visina = v;}

 //konstruktor osnovne klase
 DvaDoblik(double s, double v) { sirina = s; visina = v; }
};

class Trougao: public DvaDoblik {
public:
 string vrsta;
 double povrsina() { return getSirina() * getVisina() / 2; }

 Trougao(string tip, double s, double v) : DvaDoblik(s, v) {
 //inicijalizacija izvedenog dela
 vrsta = tip;
 }
 void prikaziVrstu() {
 cout << "Trougao je: " << vrsta << endl;
 }
};

int main() {
 Trougao t("jednakostranicni", 4.0, 4.0);
 t.prikaziDimenzije();
 t.prikaziVrstu();
 cout << "Povrsina trougla je: " << t.povrsina() << endl;
 return 0;
}

```

Rezultat izvršavanja:

**sirina: 4, visina: 4**

**Trougao je: jednakostranicni**

**Povrsina trougla je: 8**

---

Ovde konstruktor `Trougao()` poziva konstruktor osnovne klase `DvaDoblik` sa paramerima `v` i `s`, kojima se inicijalizuju podaci članovi `visina` i `sirina` iz osnovne klase. Konstruktor klase `Trougao` treba da inicijalizuje samo član koji je jedinstven za `trougao`, a to je `vrsta`. Na taj način se klasi `DvaDoblik` ostavlja potpuna sloboda da konstruiše podobjekat osnovne klase. Takođe, u klasu `DvaDoblik` mogu se dodati svojstva za koja izvedene klase ne moraju ni da znaju, bez narušavanja postojećeg kôda. Izvedena klasa može da pozove sve oblike konstruktora osnovne klase; izvršava se onaj konstruktor koji odgovara navedenim argumentima. Na primer, u klasama `DvaDoblik` i `Trougao` može da bude definisano nekoliko konstruktora:

```

DvaDOblik::DvaDOblik(double s, double v) { sirina = s; visina = v; }
DvaDOblik::DvaDOblik(double x) { sirina = visina = x; }
//default konstruktor
DvaDOblik::DvaDOblik() { sirina = visina = 0.0; }

//default konstruktor
Trougao::Trougao() { vrsta = "nepoznat"; }
Trougao::Trougao(string tip, double v, double s):
 DvaDOblik(s, v) { vrsta = tip; }
//konstruktor jednakokraničnog trougla
Trougao::Trougao(string tip, double x):
 DvaDOblik(x) { vrsta = "jednakokranični; }

```

U hijerarhiji klasa, ako konstruktor osnovne klase zahteva parametre, onda sve izvedene klase moraju da prosleđuju taj parametar "naviše". To pravilo važi bez obzira da li konstruktor izvedene klase ima parametre ili ne.

Postavlja se pitanje kojim redom se pozivaju konstruktori kada se kreira objekat izvedene klase. Odgovor na to pitanje daće nam sledeći primer:

---

```

#include <iostream>

using namespace std;

class X {
public:
 X() { cout << "konstruktor klase X" << endl; }
 ~X() { cout << "destruktor klase X" << endl; }
};

class osnovna {
public:
 osnovna() { cout << "konstruktor osnovne klase" << endl; }
 ~osnovna() { cout << "destruktor osnovne klase" << endl; }
};

class izvedena {
 X x;
public:
 izvedena() { cout << "konstruktor izvedene klase" << endl; }
 ~izvedena() { cout << "destruktor izvedene klase" << endl; }
};

int main() {
 izvedena i;
 return 0;
}

```

Rezultat izvršavanja:

```

konstruktor osnovne klase
konstruktor klase X
konstruktor izvedene klase
destruktor izvedene klase
destruktor klase X
destruktor osnovne klase

```

---

Ovaj program samo kreira i uništava objekat ob klase izvedene iz osnovne klase, koji u sebi sadrži i jedan podatak član tipa klase X. Kao što rezultat izvršavanja ilustruje, prvo se izvršava konstruktor osnovne klase, zatim konstruktori svih eventualnih klasa podataka članova, i na kraju konstruktor izvedene klase. Pošto se objekat ob uništava čim se završi `main()`, redosled pozivanja destruktora je obrnut: prvo se poziva destruktor izvedene klase, zatim destruktori svih klasa podatak članova, i na kraju destruktor izvedene klase. Drugim rečima, konstruktori se pozivaju u redosledu izvođenja, a destruktori u obrnutom smeru. To pravilo je intuitivno jasno, jer kada bi se destruktori pozivali u redosledu izvođenja, prvo bi bio uništen deo objekta koji pripada osnovnoj klasi, pa bi u memoriji ostao nepotpun objekat sa delom koji pripada samo izvedenoj klasi kome ne bi moglo da se pristupi.

#### 7.4. Višestruko nasleđivanje

Klasa može da nasledi nekoliko osnovnih klasa; tada svaki objekat izvedene klase nasleđuje sve članove svih svojih osnovnih klasa. Na primer, motocikl sa prikolicom je jedna vrsta motocikla, ali i jedna vrsta vozila sa tri točka. Pri tom, motocikl nije vrsta vozila sa tri točka, niti je vozilo sa tri točka vrsta motocikla, već su to dve različite klase. Klasa se višestruko izvodi tako što se u zaglavlju njene deklaracije navode imena osnovnih klasa razdvojena zarezima, kao u sledećem primeru:

```

class Motocikl {
 //
};
class VoziloSaTriTocka {
 //
};

class MotociklSaPrikolicom: public VoziloSaTriTocka, public Motocikl {
 //
};

```

## 7.5. Pokazivači na objekte izvedenih klasa

Da bismo razumeli virtuelne funkcije i polimorfizam, potrebno je da savladamo još jednu važnu osobinu pokazivača u jeziku C++. Već smo naučili da se pokazivači mogu dodeljivati jedan drugome samo ako su istog osnovnog tipa (odnosno, pokazivač na `int` ne može se dodeliti pokazivaču na `double`, osim uz eksplicitnu konverziju koja gotovo nikad nije dobra ideja). Međutim, postoji jedan važan izuzetak od tog pravila: u jeziku C++, pokazivač na objekat osnovne klase može da pokazuje i na objekat bilo koje klase koja je izvedena iz te osnovne klase. To ćemo ilustrovati sledećim primerom:

```
osnovna osnovni_ob;
izvedena izvedeni_ob;
osnovna *po;
//obe sledece naredbe su u redu
po = &osnovni_ob;
po = &izvedeni_ob;
```

Pokazivač na objekat osnovne klase može se koristiti za pristup samo onim delovima objekta izvedene klase koji su nasleđeni iz osnovne klase. Tako se u prethodnom primeru pokazivač `po` može koristiti za pristup svim elementima objekta `izvedeni_ob` koji su nasleđeni iz klase `osnovna`. Treba razumeti i da iako se pokazivač na osnovnu klasu može koristiti za pokazivanje na izvedeni objekat, obrnuto ne važi, odnosno pokazivač na objekat izvedene klase ne može se koristiti kao pokazivač na objekat osnovne klase. Takođe, prisetimo se da se pokazivač inkrementira i dekrementira u skladu sa svojim osnovnim tipom, odnosno sa tipom objekta na koji pokazuje. Kada pokazivač na osnovnu klasu pokazuje na objekat izvedene klase, inkrementiranje ga neće pomeriti na sledeći objekat osnovne klase, već na ono što on "misli" da je sledeći objekat osnovne klase. Zbog toga treba biti pažljiv. Činjenica da se pokazivač na osnovni tip može koristiti i kao pokazivač na izvedene tipove je izuzetno važna osobina jezika C++ i način na koji se u njemu implementira polimorfizam.

## 7.6. Nadjačavanje funkcija u izvedenim klasama

Ponekad je korisno da izvedena klasa promeni "ponašanje" osnovne klase, odnosno da definiše sopstvenu verziju nekog metoda iz osnovne klase. To se zove *nadjačavanje funkcija članica* (overriding), a znači da svaka izvedena klasa može da ima svoju verziju funkcije iz osnovne klase. Primer nadjačavanja smo već videli u klasama `DvaDoblik` i `Trougao` na primeru funkcije `povrsina()`, koja je različito definisana za ove dve klase:

```
double DvaDoblik::povrsina() { return getSirina() * getVisina(); }
double Trougao::povrsina() { return getSirina() * getVisina() / 2; }
```

Treba razlikovati preklapanje i nadjačavanje funkcija. Preklopljene funkcije imaju isto ime, ali različit broj i (li) tipove argumenata. Kompajler na osnovu tipa i broja stvarnih argumenata preklopljene funkcije određuje koju će verziju funkcije pozvati. Preklapanje se može koristiti i za "obične" funkcije, tj. i za funkcije koje nisu članice neke klase, a kada se koristi u klasama, tada imamo nekoliko funkcija članica istog imena u istoj klasi.

Nadjačane funkcije postoje samo u različitim klasama. One imaju isto ime, povratni tip, broj i tip parametara. Razlikuju im se samo tela u osnovnoj i izvedenim klasama. Važno je primetiti da iako izvedene klase imaju svoju verziju nadjačane funkcije, objekti tipa osnovne klase pozivaće svoju, "osnovnu" verziju nadjačane funkcije, kao što ilustruje sledeći primer:

---

```
#include <iostream>

using namespace std;

class osnovna {
public:
 void prikaziPoruku() {
 cout << "ovo je osnovna klasa" << endl;
 }
};

class izvedena: public osnovna {
public:
 void prikaziPoruku() {
 cout << "ovo je izvedena klasa" << endl;
 }
};

int main() {
 osnovna o;
 izvedena i;
 o.prikaziPoruku();
 i.prikaziPoruku();
 return 0;
}
```

Rezultat izvršavanja:

**ovo je osnovna klasa**  
**ovo je izvedena klasa**

---

## 7.7. Virtuelne funkcije i polimorfizam

*Polimorfizam* je veoma važna osobina objektno orijentisanog jezika jer omogućuje da osnovna klasa definiše funkcije koje će biti zajedničke za sve izvedene klase, ali da izvedenim klasama ostavi slobodu da same implementiraju sve te funkcije (ili samo neke od njih). Ponekad se ideja polimorfizma objašnjava i ovako: osnovna klasa diktira opšti interfejs koji će imati sve klase izvedene iz nje, ali ostavlja izvedenim klasama da definišu kako će zaista implementirati taj interfejs. Zbog toga se polimorfizam često opisuje frazom “jedan interfejs, više metoda”. Da bi se razumeo polimorfizam, treba imati na umu da osnovna klasa i izvedene klase formiraju hijerarhiju koja se kreće od uže generalizacije ka široj (tj. od osnovne ka izvedenim klasama). Kada je pravilno projektovana, osnovna klasa ima sve elemente koje izvedene klase mogu direktno da koriste, kao i funkcije koje bi izvedene klase trebalo samostalno da implementiraju. Pošto interfejs diktira osnovna klasa, deliće ga sve izvedene klase, ali će implementacija interfejsa biti specifična za svaku od njih. Postavlja se pitanje zašto je toliko važno imati dosledan interfejs sa različitim implementacijama. Da bismo odgovorili na to pitanje, podsetimo se ideje vodilje objektno orijentisanog pristupa, a to je da treba da olakša održavanje veoma složenih programa. Ako je program pravilno projektovan, zna se da se svim objektima izvedenim iz osnovne klase pristupa na isti način, iako će se njihovo ponašanje pomalo razlikovati u zavisnosti od toga kojoj izvedenoj klasi pripadaju. To znači da programer treba da zapamti samo jedan interfejs, umesto nekoliko njih. Takođe, izvedena klasa može da koristi sve funkcije osnovne klase, ako joj tako odgovara; nema potrebe da “izmišlja toplu vodu”. Razdvajanje interfejsa i implementacije omogućuje projektovanje biblioteka klasa, koje se zatim mogu koristiti za izvođenje sopstvenih klasa koje zadovoljavaju postavljene zahteve. To je veliko olakšanje u programiranju složenih sistema.

Da bi se u jeziku C++ ostvario polimorfizam, koriste se virtuelne funkcije. *Virtuelna funkcija* je deklarirana ključnom reči `virtual` u osnovnoj klasi i za nju se pretpostavlja da će biti nadjačana u izvedenim klasama. Ključna reč `virtual` na početku deklaracije virtuelne funkcije u osnovnoj klasi ne ponavlja se u izvedenim klasama (mada nije greška ako se to uradi); takođe, kada je funkcija jednom deklarirana kao virtuelna u osnovnoj klasi, ona ostaje takva bez obzira na dubinu hijerarhije izvođenja. Klasa koja sadrži virtuelnu funkciju zove se *polimorfna klasa*.

Kada se virtuelna funkcija pozove preko pokazivača na osnovnu klasu, C++ određuje koja će se nadjačana verzija te funkcije pozvati na osnovu tipa objekta na koji pokazuje pokazivač, a ne na osnovu njegovog osnovnog tipa. Određivanje koja će verzija funkcije biti pozvana dešava se tokom izvršavanja programa. Dakle, polimorfizam je svojstvo da svaki objekat izvedene klase izvršava metod tačno onako kako je to definisano u njegovoj izvedenoj klasi, čak i kada mu se pristupa preko pokazivača na objekat osnovne klase.

Da bismo bolje razumeli virtuelne funkcije, vratićemo se na primer klase `DvaDoblik`

u kojoj ćemo definisati funkciju `povrsina()` kao virtuelnu. Ova funkcija biće nadjačana u klasama `Trougao` i `Pravougaonik`.

---

```
#include <iostream>

using namespace std;

class DvaDOblik {
 double sirina;
 double visina;
public:
 DvaDOblik(double s, double v) {
 visina = v;
 sirina = s;
 }
 double getVisina() const { return visina;}
 double getSirina() const { return sirina;}
 virtual double povrsina() {

 cout << "greska! funkcija povrsina mora biti nadjacana! ";
 return 0.0 ;
 }
};

class Pravougaonik: public DvaDOblik {
public:
 Pravougaonik(double s, double v): DvaDOblik(s, v) { }
 double povrsina() {
 return getVisina() * getSirina();
 }
};

class Trougao: public DvaDOblik {
public:
 Trougao(double s, double v): DvaDOblik(s, v) { }
 double povrsina() {
 return getVisina() * getSirina() / 2;
 }
};

int main() {
 DvaDOblik dvaDOblik(3, 4);
 Trougao trougao(3, 4);
 Pravougaonik pravougaonik(3,4);
 DvaDOblik *poblik;
 poblik = &dvaDOblik;
 cout << poblik->povrsina() << endl;
```

```

 poblik = &trougao;
 cout << poblik->povrsina() << endl;
 poblik = &pravougaonik;
 cout << poblik->povrsina() << endl;
 return 0;
}

```

Rezultat izvršavanja:

**greska! funkcija povrsina mora biti nadjacana. 0**

**6**

**12**

Iako je jednostavan, ovaj primer ilustruje snagu virtuelnih funkcija i polimorfizma. U funkciji `main()` deklarisan je pokazivač `poblik` tipa `DvaD0blik`. Ovom pokazivaču su dodeljene adrese objekata tipa `Trougao` i `Pravougaonik`; to je moguće jer pokazivač na osnovni tip može da pokazuje i na objekte izvedenih tipova. Međutim, stvarni tip objekta na koji pokazuje pokazivač na osnovnu klasu određuje se tokom izvršavanja i u skladu sa tim poziva se odgovarajuća verzija funkcije `povrsina()`. Interfejs za računanje površine za sve 2D objekte tako postaje isti bez obzira o kom tipu oblika se radi.

Drugim rečima, verziju virtuelne funkcije koja će biti pozvana određuje stvarni tip objekta na koji pokazuje pokazivač tokom izvršavanja programa, a ne osnovni tip tog pokazivača sa kojim je deklarisan. Zbog toga se u prethodnom primeru pozivaju tri različite verzije funkcije `povrsina()`.

Virtuelna funkcija se može pozvati i bez pokazivača ili reference na objekat osnovne klase, standardnom sintaksom tačke. To znači da bi u prethodnom primeru bilo sintakso ispravno napisati:

```
trougao.povrsina();
```

Međutim, polimorfizam se ostvaruje samo kada se virtuelne funkcije pozivaju preko pokazivača na osnovnu klasu. Efekat poziva virtuelne funkcije direktno, bez pokazivača, isti je kao kada se izostavi reč `virtual` u njenoj deklaraciji i svodi se na prosto nadjačavanje funkcija.

### 7.7.1. Statičko i dinamičko povezivanje

Često se u diskusijama o objektno orijentisanom programiranju mogu čuti dva termina koji su u direktnoj vezi sa polimorfizmom: statičko ili rano povezivanje, i dinamičko ili kasno povezivanje, pa ćemo se na njih ukratko osvrnuti.

*Statičko ili rano povezivanje* (early binding) odnosi se na događaje koji se dešavaju tokom kompajliranja. U suštini, rano povezivanje dešava se kada su sve informacije potrebne za pozivanje funkcije poznate tokom kompajliranja (odnosno objekat se može povezati sa funkcijom tokom kompajliranja). Primeri ranog poveziva-

nja su standardni pozivi bibliotečkih funkcija i pozivi preklapljenih funkcija. Velika prednost ranog povezivanja je efikasnost: pošto su sve informacije potrebne za poziv funkcije poznate tokom kompajliranja, pozivi funkcija su veoma brzi.

*Kasno ili dinamičko povezivanje* u jeziku C++ odnosi se na pozive funkcija koji se rešavaju tek tokom izvršavanja programa. Za kasno povezivanje u jeziku C++ koriste se virtualne funkcije. Kao što znamo, kada se virtualnim funkcijama pristupa preko pokazivača ili reference na osnovnu klasu, verziju virtualne funkcije koja će biti pozvana određuje stvarni tip objekta na koji pokazuje pokazivač. Pošto u većini slučajeva stvarni tip objekta na koji pokazuje pokazivač na osnovnu klasu nije poznat tokom kompajliranja, objekat se povezuje sa funkcijom tek tokom izvršavanja. Najvažnija prednost kasnog povezivanja je prilagodljivost; za razliku od ranog povezivanja, dinamičko povezivanje omogućuje pisanje programa koji reaguju na događaje koji se dešavaju tokom izvršavanja programa bez pisanja kôda koji bi se posebno bavio svim mogućim varijantama. Ipak, pošto se poziv funkcije razrešava tek tokom izvršavanja, dinamičko povezivanje je sporije od statičkog.

### 7.7.2. Čiste virtualne funkcije i apstraktne klase

Ponekad je potrebno napraviti osnovnu klasu koja definiše samo opšti oblik koji će imati sve izvedene klase, tako da one same definišu detalje. Takva klasa određuje prirodu funkcija koje izvedene klase moraju da implementiraju, ali ne nudi i implementaciju tih funkcija. Jedan tipičan slučaj za to je kada osnovna klasa ne može da obezbedi smislenu implementaciju neke funkcije, kao u slučaju funkcije `povrsina()` u klasi `DvaDOblik`. Definicija funkcije `povrsina()` u osnovnoj klasi samo "čuva mesto" za njene stvarne implementacije u izvedenim klasama. Takva situacija se može rešiti tako što verzija funkcije u osnovnoj klasi samo prikaže upozoravajuću poruku, što smo i iskoristili u prethodnom primeru. Iako je ovakav pristup koristan u nekim situacijama kao što je debugovanje, obično nije preporučljiv.

U jeziku C++ moguće je deklarirati funkcije koje moraju da budu nadjačane u izvedenim klasama da bi imale smisla. Takve funkcije zovu se *čiste virtualne funkcije* (pure virtual functions). Funkcija koja je u osnovnoj klasi definisana kao čista virtualna funkcija nema implementaciju u osnovnoj klasi; zbog toga svaka izvedena klasa mora da je definiše. Čista virtualna funkcija deklarise se sledećom sintaksom:

```
virtual tip ime_funkcije(lista_parametara) = 0;
```

Uz pomoć čiste virtualne funkcije možemo da poboljšamo klasu `DvaDOblik` tako što ćemo funkciju `povrsina()` prepraviti u čistu virtualnu funkciju. Time smo istovremeno primorali sve izvedene klase da nadjačaju ovu funkciju.

```
class DvaDOblik {
 double sirina;
 double visina;
```

```
public:
 DvaDOblik(double s, double v) {
 visina = v;
 sirina = s;
 }
 double getVisina() const { return visina;}
 double getSirina() const { return sirina;}
 virtual double površina() {} = 0; //čista virtuelna funkcija
};
```

Ako klasa ima barem jednu čistu virtuelnu funkciju, ona se zove *apstraktna klasa*. Objekti apstraktnih klasa ne mogu da postoje, jer se ne bi znalo šta treba da se desi kada se za objekat apstraktna klase pozove čista virtuelna funkcija:

```
DvaDOblik oblik(3,4); //greska!
oblik.površina(); //greska!
```

Međutim, mogu da postoje pokazivači na objekte apstraktnih klasa, koji se koriste za ostvarivanje polimorfizma, odnosno koriste se kao pokazivači na objekte izvedenih klasa.

### 7.7.3. Virtuelni destruktori

Postavlja se pitanje da li konstruktori i destruktori klase mogu da budu virtuelni. Konstruktor je funkcija koja od “obične gomile u memoriji” pravi “živi” objekat. Poziva se pre nego što je objekat kreiran, pa nema smisla da bude virtuelan, što C++ ni ne dozvoljava. Kada se objekat kreira, njegov tip je uvek poznat, pa je određen i konstruktor koji se poziva.

Destruktor sa druge strane pretvara postojeći objekat u “hrpu bitova” u memoriji. Destruktori, za razliku od konstruktora, mogu da budu virtuelni. Virtuelni mehanizam, tj. dinamičko vezivanje tačno određuje koji destruktor (osnovne ili izvedene klase) će biti prvo pozvan kada se objektu pristupa posredno, preko pokazivača.

Odatle i praktično pravilo: ako osnovna klasa ima neku virtuelnu funkciju, onda i njen destruktor treba da bude virtuelan.

---

```
#include <iostream>

using namespace std;

class osnovna {
public:
 virtual ~osnovna() {
 cout<< "virtuelni destruktor osnovne" << endl;
 }
};
```

```
class izvedena: public osnovna {
public:
 virtual ~izvedena() {
 cout <<"virtuelni destruktor izvedene" << endl;
 }
};

void oslobodi(osnovna *po) {
 delete po; //poziv virtuelnog destruktora
}

int main() {
 osnovna *po = new osnovna;
 izvedena *pi = new izvedena;
 //pozivi virtuelnih destruktora
 oslobodi(po);
 oslobodi(pi);
 return 0;
}
```

Rezultat izvršavanja:

**virtuelni destruktor osnovne  
virtuelni destruktor izvedene  
virtuelni destruktor osnovne**

---

## Pitanja

1. Članovi klase deklarirani kao \_\_\_\_\_ dostupni su samo funkcijama članicama klase.

- a) public
- b) protected
- c) private

2. Nasleđivanje je vrsta ponovnog korišćenja softvera u kome nove klase obuhvataju svojstva i ponašanje postojećih klasa i obogaćuju ih novim mogućnostima.

- a) tačno
- b) pogrešno

3. Nasleđivanjem se olakšava ponovno korišćenje softvera koji se dokazao kao kvalitetan.

- a) tačno
- b) pogrešno

4. Članovi deklarirani kao \_\_\_\_\_ u osnovnoj klasi dostupni su u osnovnoj i izvedenim klasama.

- a) public
- b) protected
- c) private

5. Klasa Ptica u deklaraciji `class Ljubimac : Ptica` je izvedena:

- a) javno
- b) privatno

6. Kada se kreira objekat izvedene klase, poziva se \_\_\_\_\_ osnovne klase koji obavlja sve neophodne inicijalizacije podataka članova osnovne klase u objektu izvedene klase.

- a) konstruktor
- b) konstruktor kopije
- c) destruktor

7. Izvedena klasa ne nasleđuje konstruktore osnovne klase.

- a) tačno
- b) pogrešno

8. Konstruktor izvedene klase je dužan da prosledi sve potrebne parametre konstruktoru osnovne klase.

- a) tačno
- b) pogrešno

9. Kada se uništava objekat izvedene klase, destruktori se pozivaju u obrnutom redosledu od konstruktora.

- a) tačno
- b) pogrešno

10. Ako klasa sadrži barem jednu čistu virtuelnu metodu, onda je takva klasa apstraktna.

- a) tačno
- b) pogrešno

11. Ne mogu da se kreiraju objekti apstraktnih klasa.

- a) tačno
- b) pogrešno

12. Sve virtuelne funkcije u apstraktnoj osnovnoj klasi moraju se deklarirati kao čiste virtuelne funkcije.

- a) tačno
- b) pogrešno

13. Ako je u osnovnoj klasi deklarirana čista virtuelna funkcija, onda izvedena klasa mora da implementira tu funkciju da bi bila konkretna klasa.

- a) tačno
- b) pogrešno

14. Pokazivačima na objekat osnovne klase može se dodeljivati adresa objekta izvedene klase.

- a) tačno
- b) pogrešno

15. Ako osnovna klasa ima neku virtuelnu funkciju, onda i njen destruktor treba da bude virtuelan.

- a) tačno
- b) pogrešno

## Glava 8

# Ulazno izlazni (U/I) sistem

Od početka knjige koristili smo U/I sistem jezika C++, bez mnogo formalnih objašnjenja. Pošto je U/I sistem zasnovan na hijerarhiji klasa, nije bilo moguće da se objasne njegovi detalji bez prethodnog savladavanja klasa i nasleđivanja. Sada je trenutak da se detaljno pozabavimo ulazno-izlaznim sistemom. Pošto je on ogroman, nećemo moći da pomenemo sve klase i funkcije, ali ćemo se upoznati sa najvažnijim i najčešće korišćenim delovima.

Trenutno postoje dve verzije objektno orijentisane C++ biblioteke za ulaz/izlaz: starija se zasniva na originalnim specifikacijama za C++, a noviju definiše ANSI/ISO standard jezika. Stariju U/I biblioteku podržava zaglavlje `<iostream.h>`, a novu zaglavlje `<iostream>`. U najvećem broju slučajeva ove dve datoteke programeru izgledaju isto, zato što je nova U/I biblioteka zapravo ažurirana i poboljšana verzija stare. U stvari, većina izmena je "ispod haube", u načinu na koji su biblioteke implementirane, a ne kako se koriste. Sa tačke gledišta programera, dve su najvažnije razlike između starih i novih U/I biblioteka: prvo, nova sadrži dodatne funkcije i nove tipove podataka, što znači da je ona proširenje stare. Skoro svi programi pisani za stari U/I kompajliraće se bez izmena i kada se koristi nova biblioteka. Drugo, stara U/I biblioteka nalazila se u globalnom imeniku, dok je nova u imeniku `std`. Pošto se stara biblioteka više ne koristi, opisaćemo samo novu U/I biblioteku (ali treba znati da većina informacija važi i za staru).

## 8.1. Pojam toka

Kada je reč o U/I sistemu jezika C++, najvažnije je znati da on radi sa tokovima. *Tok* (stream) je apstrakcija koja proizvodi ili prihvata podatke. Tok se sa fizičkim uređajima povezuje preko U/I sistema. Svi tokovi se ponašaju na isti način, čak i kada su povezani sa različitim fizičkim uređajima. Na primer, isti metod koji se koristi za ispis na ekranu može da se koristi za ispis datoteke na štampaču. U svom najopštijem obliku, tok je logički interfejs za datoteku. Pojam datoteke u jeziku C++ je veoma širok, pa to može biti datoteka na disku, ekran, tastatura, priključak na računaru itd. Iako se datoteke razlikuju po mnogim svojstvima, svi tokovi su isti. Prednost takvog pristupa je u tome što programeru svi hardverski uređaji izgledaju isto.

### 8.1.1. Binarni i tekstualni tokovi

Kada se govori o ulazu i izlazu programa, treba razlikovati dve široke kategorije podataka: binarne i tekstualne. Binarni podaci se predstavljaju u binarnom obliku na isti način na koji se zapisuju unutar računara. Binarni podaci su dakle obični nizovi nula i jedinica čija interpretacija nije odmah prepoznatljiva. Tekstualni podaci se predstavljaju nizovima znakova koji se mogu pročitati, odnosno razumljivi su za programera. Ali pojedinačni znakovi u tekstualnom podatku se i sami zapisuju u binarnom obliku prema raznim šemama kodiranja (npr. ASCII ili Unicode). Prema tome, svi podaci u računarima su binarni, ali su tekstualni podaci samo viši stepen apstrakcije u odnosu na bezličan niz nula i jedinica. Za ove dve vrste podataka se primenjuju različiti postupci prilikom pisanja i čitanja podataka.

Da bismo bolje razumeli razliku između binarnih i tekstualnih tokova, razmotrimo jedan konkretan primer izlaza programa: pretpostavimo da se u programu ceo broj 167 tipa `int` upisuje u datoteku na disku. Broj 167 je u programu zapisan unutar jednog bajta u binarnom obliku 10100111 (ili ekvivalentno 0xA7 u heksadecimalnom zapisu). Ukoliko se za pisanje broja 167 koristi binarni tok, jedan bajt odgovarajuće binarne vrednosti 10100111 (0xA7 heksadecimalno) biće upisan neizmenjen u datoteku. Međutim, ako se za njegovo pisanje koristi tekstualni tok, onda se broj 167 upisuje kao niz znakova '1', '6' i '7'. Pri tom se svaki znak u nizu konvertuje specifičnom kodnom šemom u odgovarajući bajt koji se zapravo upisuje u datoteku na disku. Ako se, recimo, koristi ASCII šema, onda znaku '1' odgovara jedan bajt čija ASCII vrednost iznosi 0x31 u heksadecimalnom zapisu (ili 00110001 u binarnom obliku), znaku '6' odgovara bajt 0x36 i znaku '7' odgovara bajt 0x37. Prema tome, za broj 167 tipa `int` u datoteku se redom upisuju tri bajta 0x31, 0x36 i 0x37.

Ukoliko se koristi binarni tok, bajtovi se neizmenjeni prenose iz programa ili u njega. Binarni tokovi su mnogo efikasniji za ulaz ili izlaz programa, jer se kod njih ne gubi dodatno vreme potrebno za konverziju bajtova u znakove prilikom prenosa. To znači i da su podaci upisani korišćenjem binarnog toka prenosivi, jer se mogu čitati neizmenjeni na računaru koji je različit od onog na kome su upisani. Ipak, binarni tokovi se mnogo ređe koriste, praktično samo kada se radi o velikim količinama prenosa podataka u specijalnim primenama (na primer, kod baza podataka). Osnovni nedostatak binarnih tokova je to što su dobijeni binarni podaci nečitljivi za ljude, pa je potreban dodatni napor da bi se oni mogli interpretirati. Zbog toga ćemo se u ovoj knjizi ograničiti na tehnike upisa i čitanja iz tekstualnih tokova.

### 8.1.2. Predefinisani tokovi

C++ sadrži nekoliko predefinisanih objekata tokova koji su automatski dostupni kada C++ program počne da se izvršava. To su `cin`, `cout`, `cerr` i `clog`.

| Tabela 8.1: Predefinisani objekti tokova. |                   |                   |
|-------------------------------------------|-------------------|-------------------|
| Tok                                       | Značenje          | Standardni uređaj |
| cin                                       | standardni ulaz   | tastatura         |
| cout                                      | standardni izlaz  | ekran             |
| cerr                                      | standardna greška | ekran             |
| clog                                      | baferovani cerr   | ekran             |

cin je objekat toka povezan sa standardnim ulazom, a objekat cout je pridružen standardnom ulazu. Tok cerr je povezan sa standardnim izlazom, kao i clog. Razlika između ova dva objekta je u tome što je clog baferisan, a cerr nije. To znači da se sve što se šalje na cerr odmah prikazuje, dok se ono što je poslato u clog prikazuje tek kada je bafer pun. Obično su cerr i clog tokovi u koje se prikazuju rezultati debagovanja ili informacije o greškama. C++ tokovi su standardno vezani za konzolu, ali ih program ili operativni sistem mogu preusmeriti u druge uređaje ili u datoteke.

### 8.1.3. Klase tokova

Već smo pominjali da C++ podržava U/I tokove u zaglavlju <iostream> u kome se nalazi prilično složena hijerarhija klasa za U/I operacije. U/I sistem zasniva se na dve povezane, ali ipak različite, hijerarhije klasa. Prva se zasniva na U/I klasi niskog nivoa basic\_streambuf. Ona obezbeđuje osnovne operacije za ulaz i izlaz niskog nivoa, kao i podršku za ceo C++ U/I sistem. Osim ako se ne budete bavili naprednijim U/I programiranjem, klasu basic\_streambuf nećete morati da koristite direktno. Hijerarhija klasa sa kojom se najčešće radi izvedena je iz klase basic\_ios. To je U/I klasa visokog nivoa koja obezbeđuje formatiranje, proveru grešaka i statusne informacije vezane za U/I tokove. Klasa basic\_ios se koristi kao osnovna klasa za nekoliko izvedenih klasa, uključujući basic\_istream, basic\_ostream i basic\_iostream. Te klase se koriste za pravljenje ulaznih, izlaznih i ulazno/izlaznih tokova.

U nastavku je dat spisak U/I klasa:

| Tabela 8.2: U/I klase tokova |                                                                          |
|------------------------------|--------------------------------------------------------------------------|
| Klasa                        | Opis                                                                     |
| streambuf                    | Klasa niskog nivoa koja se retko koristi direktno                        |
| ios                          | Klasa sa korisnim metodama koje upravljaju ili nadgledaju operacije toka |
| istream                      | Ulazni tok                                                               |
| ostream                      | Izlazni tok                                                              |
| iostream                     | Ulazno-izlazni tok                                                       |
| fstream                      | Tok za datoteke                                                          |
| ifstream                     | Ulazni tok za datoteke                                                   |
| ofstream                     | Izlazni tok za datoteke                                                  |

Klase u prethodnoj tabeli najčešće ćete koristiti u programima, a njihovi nazivi su isti kao u staroj U/I biblioteci. Klasa `ios` sadrži brojne funkcije članice i promenljive koje kontrolišu osnovne operacije sa tokovima. Prilikom korišćenja pomenutih klasa obavezno je uključiti zaglavlje `<iostream>` u program.

## 8.2. Formatiranje ulaza i izlaza

Sve dosad formatiranje ulaznih i izlaznih podataka prepuštali smo standardnim podešavanjima jezika C++. Međutim, moguće je precizno kontrolisati format podataka i to na dva načina. Prvi način koristi funkcije članice klase `ios`, a drugi specijalne manipulatorske funkcije. U nastavku ćemo proučiti oba metoda.

### 8.2.1. Formatiranje pomoću funkcija članica klase `ios`

Sa svakim tokom povezan je skup flegova za formatiranje koji upravljaju načinom na koji tok formatira podatke. Klasa `ios` ima tzv. bit-masku `fmtflags` koja sadrži flegove za formatiranje. Kada je postavljen fleg `skipws`, beline na početku (razmaci, tabulatori i znakovi za nov red) odbacuju se kada se unose podaci u tok. Kada je `skipws` resetovan (tj. kada mu je vrednost 0), beline se ne preskaču. Kada je postavljen fleg `left`, izlaz se poravnava ulevo, a kada je postavljen fleg `right`, izlaz se poravnava udesno. Kada je postavljen fleg `internal`, numerička vrednost se odseca tako da popuni polje umetanjem razmaka između bilo kog znaka ili osnove. Ako nijedan od pomenutih flegova nije postavljen, izlaz se poravnava udesno. Numeričke vrednosti se standardno prikazuju u decimalnoj notaciji, ali je i to moguće promeniti. Postavljanje flega `oct` prouzrokuje prikaz vrednosti u oktalnom režimu, a postavljanje flega `hex` ih prikazuje u heksadecimalnoj notaciji. Da bi se prikaz vratio u decimalan format, treba postaviti fleg `dec`. Postavljanje flega `showbase` prouzrokuje prikaz osnove brojnih vrednosti. Na primer, ako je osnova za konverziju heksadecimalna, vrednost 1F biće prikazana kao 0x1F. Standardno, kada se koristi eksponencijalna notacija za brojeve, slovo `e` je prikazano kao malo, isto kao i slovo `x` u heksadecimalnoj vrednosti. Ako je postavljen fleg `uppercase`, te vrednosti prikazuju se velikim slovima. Postavljanje flega `showpos` prouzrokuje prikaz znaka `+` ispred pozitivnih vrednosti. Postavljanjem flega `showpoint` prikazuju se decimalna tačka i vodeće nule za sve vrednosti u pokretnom zarezu, bez obzira da li su neophodne ili ne. Postavljanjem flega `scientific`, brojne vrednosti u pokretnom zarezu prikazuju se u eksponencijalnoj notaciji. Kada je postavljen fleg `fixed`, vrednosti u pokretnom zarezu prikazuju se u standardnoj notaciji. Kada nije postavljen nijedan fleg, kompajler bira odgovarajući metod. Kada je postavljen fleg `unitbuf`, bafer se ne prazni nakon svake operacije umetanja. Kada

je postavljen fleg `boolalpha`, logičke vrednosti mogu da se unose i ispisuju pomoću ključnih reči `true` i `false`.

### Postavljanje i brisanje flegova za formatiranje

Da bi se fleg postavio, koristi se funkcija `setf()` koja je članica klase `ios`. Na primer, da bi se postavio fleg `showbase`, koristi se naredba:

```
tok.setf(ios::showbase);
```

Primitite korišćenje kvalifikatora dosega `ios::` za `showbase`. Pošto je `showbase` nabrojana konstanta u klasi `ios`, ona se mora navesti na taj način, a to važi i za sve ostale flegove za formatiranje. Sledeći program koristi funkciju `setf()` za postavljanje flegova `showpos` i `scientific`:

```
#include <iostream>
using namespace std;
int main() {
 cout.setf(ios::showpoint);
 cout.setf(ios::showpos);
 cout << 100.0; //prikazuje +100.0
 return 0;
}
```

Proizvoljan broj flegova se može kombinovati u jednoj naredbi. Na primer, u prethodnom programu umesto dve naredbe u kojoj se postavljaju flegovi, mogla se koristiti samo jedna:

```
cout.setf(ios::scientific | ios::showpos);
```

Za isključivanje flega koristi se funkcija `unsetf()`, čiji je prototip:

```
void unsetf(fmtflags flags);
```

Ova funkcija briše flegove zadate u argumentu `flags`, a ostale ne menja. Ponekad je korisno znati trenutne vrednosti flegova. To se može postići pomoću funkcije `flags()`:

```
fmtflags flags();
```

Ova funkcija vraća trenutnu vrednost flegova u toku za koji je pozvana. Sledeći program ilustruje korišćenje funkcija `flags()` i `unsetf()`:

```
#include <iostream>
using namespace std;
int main() {
 cout.setf(ios::uppercase | ios::scientific);
 cout << 100.12; //prikazuje 1.0012E+02
 cout.unsetf(ios::uppercase); //briše uppercase
 cout << endl << 100.12; //prikazuje 1.0012e+02
 return 0;
}
```

Osim flegova za formatiranje, klasa `ios` definiše tri funkcije članice kojima se mogu dodatno podesiti širina polja za prikaz, preciznost i znak za popunu. Funkcije kojima se te vrednosti podešavaju su `width()`, `precision()` i `fill()`.

Kada se prikazuje neka vrednost, ona zauzima samo onoliko mesta koliko je potrebno za broj znakova potrebnih za prikaz. Međutim, može se zadati i minimalna širina polja pomoću funkcije `width()`. Prototip te funkcije je:

```
streamsize width(streamsize w);
```

Ovde je `w` širina polja; funkcija kao rezultat vraća prethodnu širinu polja. U nekim implementacijama, širina polja mora da bude postavljena pre bilo kakvog ispisa, a ako nije postavljena, koristi se standardna širina. Tip `streamsize` definisan je kao vrsta celog broja. Kada se postavi minimalna vrednost za širinu polja, a vrednost koja treba da se prikaže koristi manju širinu, ona se dopunjava tekućim znakom za popunu (što je standardno razmak) do zadate širine polja. Ako širina vrednosti za prikaz prevazilazi minimalnu širinu polja, onda će vrednost biti prikazana u potrebnoj širini, bez odsecanja. Kada se prikazuju vrednosti u pokretnom zarezu u eksponencijalnoj notaciji, moguće je zadati broj cifara koje će biti prikazane iza decimalne tačke pomoću funkcije `precision()` čiji je prototip:

```
streamsize precision(streamsize p);
```

Preciznost se postavlja na `p`, a funkcija vraća staru vrednost. Podrazumevana vrednost za preciznost je 6. U nekim implementacijama preciznost mora da bude postavljena pre bilo kakvog ispisa vrednosti u pokretnom zarezu; u suprotnom se koristi podrazumevana vrednost za preciznost. Ako treba da se dopuni, polje se standardno dopunjava razmacima. Postoji i specijalna funkcija `fill()` za dopunu znakovima čiji je prototip:

```
char fill(char ch);
```

Nakon poziva funkcije `fill()`, `ch` postaje nov znak za popunu, a funkcija vraća stari znak. Evo programa koji demonstrira te tri funkcije:

```
#include <iostream>

using namespace std;

int main() {
 cout.precision(4) ;
 cout.width(10);
 cout << 10.12345 << endl //prikazuje 10.12
 cout.fill('*');
 cout.width(10);
 cout << 10.12345 << endl; //prikazuje *****10.12
 //polje width primenjuje se i na stringove
 cout.width(10);
 cout << "Hi!" << endl; //prikazuje *****Hi!
 cout.width(10);
 cout.setf(ios::left); //levo poravnavanje
```

```

 cout << 10.12345; //prikazuje 10.12*****
 return 0;
}

```

Kao što je pomenuto, u nekim implementacijama neophodno je resetovati polje za širinu pre bilo koje operacije izlaza. Zbog toga se `width()` u prethodnom programu više puta poziva (postoje i preklopljene verzije funkcija `width()`, `precision()` i `ifill()` koje čitaju trenutna podešavanja, ali ih ne menjaju.)

### 8.3. Korišćenje U/I manipulatora

I/O sistem jezika C++ omogućuje formatiranje parametara toka i pomoću specijalnih funkcija koje se zovu *manipulatori*. Da bi se manipulatori sa argumentima koristili u programu, potrebno je uključiti zaglavlje `<iomanip>`.

**Tabela 8.3:** U/I manipulatori

| Manipulator                                               | Namena                                       | U/I            |
|-----------------------------------------------------------|----------------------------------------------|----------------|
| <code>boolalpha</code>                                    | Postavlja fleg <code>boolalpha</code>        | ulazno/izlazni |
| <code>dec</code>                                          | Postavlja fleg <code>dec</code>              | ulazno/izlazni |
| <code>endl</code>                                         | Prelazi u novi red i prazni tok              | izlazni        |
| <code>ends</code>                                         | Prikazuje <code>null</code>                  | izlazni        |
| <code>fixed</code>                                        | Uključuje fleg <code>fixed</code>            | izlazni        |
| <code>flush</code>                                        | Prazni tok                                   | izlazni        |
| <code>hex</code>                                          | Postavlja fleg za heksadecimalni prikaz      | izlazni        |
| <code>internal</code>                                     | Postavlja fleg <code>internal</code>         | izlazni        |
| <code>left</code>                                         | Postavlja fleg <code>left</code>             | izlazni        |
| <code>noboolalpha</code>                                  | Isključuje fleg <code>noboolalpha</code>     | ulazno/izlazni |
| <code>noshowbase</code>                                   | Isključuje fleg <code>noshowbase</code>      | izlazni        |
| <code>noshowpoint</code>                                  | Isključuje fleg <code>noshowpoint</code>     | izlazni        |
| <code>noshowpos</code>                                    | Isključuje fleg <code>noshowpos</code>       | izlazni        |
| <code>noskipws</code>                                     | Isključuje fleg <code>noskipws</code>        | ulazni         |
| <code>nounitbuf</code>                                    | Isključuje fleg <code>nounitbuf</code>       | izlazni        |
| <code>nouppercase</code>                                  | Isključuje fleg <code>nouppercase</code>     | izlazni        |
| <code>oct</code>                                          | Postavlja fleg za oktalni prikaz             | ulazno/izlazni |
| <code>resetiosflags</code><br>( <code>fmtflags f</code> ) | Resetuje flegove navedene u <code>f</code>   | ulazno/izlazni |
| <code>right</code>                                        | Postavlja fleg <code>right</code>            | izlazni        |
| <code>scientific</code>                                   | Uključuje eksponencijalni prikaz             | izlazni        |
| <code>set base(int base)</code>                           | Postavlja brojnu osnovu na <code>base</code> | ulazno/izlazni |
| <code>setfill(int ch)</code>                              | Postavlja znak za popunu na <code>ch</code>  | izlazni        |

**Tabela 8.3: U/I manipulatori**

| Manipulator             | Namena                               | U/I            |
|-------------------------|--------------------------------------|----------------|
| setiosflags(fmtflags f) | Postavlja flegove navedene u f       | ulazno/izlazni |
| setprecision(int p)     | Postavlja broj znakova za preciznost | izlazni        |
| setw(int w)             | Postavlja širinu polja na w          | izlazni        |
| showbase                | Postavlja fleg showbase              | izlazni        |
| showpoint               | Postavlja fleg showpoint             | izlazni        |
| showpos                 | Postavlja fleg showpos               | izlazni        |
| skipws                  | Postavlja fleg skipws                | ulazni         |
| unitbuf                 | Postavlja fleg unitbuf               | izlazni        |
| uppercase               | Uključuje fleg uppercase             | izlazni        |
| ws                      | Preskače vodeće razmake              | ulazni         |

Evo primera koji koristi manipulatore:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
 cout << hex << 100 << endl;
 cout << setfill('?') << setw(10) << 2343.0;
 return 0;
}
```

Rezultat izvršavanja:

**64 ??????2343**

Primetite kako se manipulatori pojavljuju unutar većih U/I izraza. Takođe uočite da kada manipulator nema argumenata, kao što je `endl()` u primeru, iza njega se ne navode zagrade. Radi poređenja, evo funkcionalno ekvivalentne verzije prethodnog programa koji koristi funkcije članice klase `ios` za postizanje istih rezultata:

```
#include <iostream>
#include <iomanip>

using namespace std;

int main() {
 cout.setf(ios::hex, ios::basefield);
 cout << 100 << endl; //100 hex
 cout.fill('?');
```

```

 cout.width(10);
 cout << 2343.0;
 return 0;
}

```

Kao što se vidi iz primera, najvažnija prednost korišćenja manipulatora nad korišćenjem funkcija članica klase `ios` jeste to što po pravilu generišu kompaktniji kôd.

Jedan od zanimljivijih manipulatora je `boolalpha` koji omogućuje ulaz i izlaz logičkih podataka pomoću reči "true" i "false" umesto brojeva. Na primer,

```

#include <iostream>
using namespace std;
int main() {
 bool b;
 b = true;
 cout << b << " " << boolalpha << b << endl;
 cout << "Unesite logičku vrednost: ";
 cin >> boolalpha >> b;
 cout << "Uneli ste: " << b;
 return 0;
}

```

Za razliku od manipulatorskih funkcija koje imaju argumente, manipulatorske funkcije bez argumenata se prilično lako programiraju i vrlo su korisne. Prototip manipulatorske funkcije je:

```

ostream &ime_manipulatora(ostream &stream) {
 //telo manipulatora;
 return stream;
}

```

Iako je argument manipulatora referenca na tok sa kojim radi, argument se ne navodi kada se manipulator poziva za tok (u ovom slučaju, za `ostream`, tj. za objekat `cout`). U nastavku je dat primer manipulatorske funkcije nazvane `setup` koja uključuje poravnavanje ulevo, podešava širinu prikaza na deset mesta i zadaje `$` kao znak za popunu:

```

#include <iostream>
#include <iomanip>
using namespace std;
ostream &setup(ostream &stream) {
 stream.setf(ios::left);
 stream << setw(10) << setfill('$');
 return stream;
}

```

```
int main() {
 cout << setup << 10 << endl;
 return 0;
}
```

Rezultat izvršavanja:

**10\$\$\$\$\$\$\$\$**

---

## 8.4. Rad sa datotekama

U/I sistem jezika C++ može se koristiti i za rad sa datotekama. Da bi se uključio datotečki U/I, u program se mora uključiti zaglavlje `<fstream>` u kome se nalazi nekoliko važnih klasa i konstanti.

U jeziku C++ datoteka se otvara tako što se povezuje sa tokom. Već znamo da postoje tri vrste tokova: ulazni, izlazni i ulazno/izlazni. Da bi se otvorio ulazni tok, on se mora deklarirati tako da bude tipa `ifstream`. Da bi se otvorio izlazni tok, on se mora deklarirati tako da pripada klasi `ofstream`. Slično, tok koji će obavljati i ulazne i izlazne operacije mora se deklarirati tako da bude tipa `fstream`. Na primer, sledeće naredbe kreiraju jedan ulazni tok, jedan izlazni tok i jedan ulazno-izlazni tok:

```
ifstream in; //ulazni tok
ofstream out; //izlazni tok
fstream io; //ulazno izlazni tok
```

Kada se kreira tok, on se sa datotekom može povezati pomoću funkcije `open()` koja postoji u sve tri klase tokova. Prototipovi funkcija `open()` su:

```
void ifstream::open(const char *filename, ios::openmode mode = ios::in);
void ofstream::open(const char *filename, ios::openmode mode = ios::out
| ios::trunc);
void fstream::open(const char *filename, ios::openmode mode = ios::in |
ios::out);
```

Ovde je `filename` ime datoteke. Ono može sadržati i relativni ili apsolutnu putanju do datoteke, ali to nije obavezno. Korišćenje relativne putanje se preporučuje jer se program može koristiti pod različitim platformama:

```
ofstream outfile;
//otvara datoteku u istom direktorijumu gde i program
outfile.open("studenti.dat");
//apsolutna putanja
outfile.open("C:\\fakultet\\casovi\\studenti.dat");
```

Vrednost parametra `mode` određuje način na koji će datoteka biti otvorena. Parametar režima (`mode`) može imati jednu od sledećih vrednosti, definisanih u klasi `ios`:

**Tabela 8.4:** Režimi rada sa datotekama.

| Parametar mode           | Značenje                                                                                                                                                          |
|--------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ios::app</code>    | Sadržaj se dodaje na kraj postojeće datoteke.                                                                                                                     |
| <code>ios::ate</code>    | Ako datoteka već postoji, program se pomera direktno na njen kraj.<br>Može se upisivati bilo gde u datoteku (ovaj režim se obično koristi sa binarnim datotekama) |
| <code>ios::binary</code> | Sadržaj se upisuje u datoteku u binarnom obliku, a ne u tekstualnom (koji je podrazumevani)                                                                       |
| <code>ios::in</code>     | Sadržaj se čita iz datoteke. Ako datoteka ne postoji, neće biti napravljena.                                                                                      |
| <code>ios::out</code>    | Sadržaj se upisuje u datoteku, a ako ona već ima sadržaj, prepisuje se.                                                                                           |
| <code>ios::trunc</code>  | Ako datoteka već postoji, njen sadržaj će biti prepisan (podrazumevani režim za <code>ios::out</code> )                                                           |

Nekoliko vrednosti za parametar `mode` mogu se kombinovati korišćenjem operatora `|`. Parametar `ios::app` prozrokuje dodavanje sadržaja na kraj datoteke i može se koristiti samo za izlazne datoteke (tj. tokove tipa `ofstream`). Parametar `ios::ate` prouzrokuje traženje kraja datoteke prilikom njenog otvaranja (iako se U/I operacije mogu dešavati bilo gde u datoteci). Vrednost `ios::in` zadaje da se u datoteku može upisivati, a `ios::out` da se iz nje može čitati. Vrednost `ios::binary` prouzrokuje otvaranje datoteke u binarnom režimu. Standardno se sve datoteke otvaraju u tekstualnom režimu. Svaka datoteka, bez obzira da li sadrži formatiran tekst ili binarne podatke, može da se otvori u binarnom ili u tekstualnom režimu; jedina razlika je da li će se dešavati konverzija znakova.

Vrednost `ios::trunc` prouzrokuje uništavanje sadržaja postojeće datoteke istog imena, tj. svođenje datoteke na nultu dužinu. Kada se izlazni tok kreira kao objekat klase `ofstream`, sve postojeće datoteke istog imena automatski se odsecaju. Sledeći deo kôda otvara tekstualnu datoteku za čitanje:

```
ofstream tok;
tok.open("test");
```

Pošto je standardna vrednost parametra `mode` funkcije `open()` ona koja je odgovarajuća za tip toka koji se otvara, često nije ni potrebno zadavati njegovu vrednost, kao u prethodnom primeru. (Neki kompajleri ne dodeljuju standardne vrednosti parametra `mode` za funkciju `fstream::open()`, tj. `in` | `out`, pa one moraju eksplicitno da se navedu.) Ako otvaranje datoteke ne uspe, vrednost toka biće `false` kada se koristi u logičkim izrazima, što se može iskoristiti za proveru da li je datoteka uspešno otvorena u naredbi kao što je ova:

```

if(!tok) {

 cout << "Datoteka ne moze da se otvori" << endl; //obrada greske
}

```

Rezultat funkcije `open()` trebalo bi proveriti pre bilo kog pokušaja menjanja datoteke. Za proveru da li je datoteka uspešno otvorena može se upotrebiti i funkcija `is_open()`, koja je članica klasa `fstream`, `ifstream` i `ofstream`. Prototip te funkcije je:

```
bool is_open();
```

Funkcija `is_open()` vraća `true` ako je tok povezan sa otvorenom datotekom, a `false` u suprotnom. Na primer, sledeće naredbe proveravaju da li je datoteka `tok` trenutno otvorena:

```

if(!tok.is_open())
 cout << "Datoteka je otvorena" << endl;

```

Iako je sasvim u redu da se funkcija `open()` koristi za otvaranje datoteke, to se uglavnom ne radi zato što klase `ifstream`, `ofstream` i `fstream` imaju konstruktore koji automatski otvaraju datoteke. Konstruktori imaju iste parametre i standardne vrednosti kao funkcija `open()`. Zbog toga se datoteka najčešće otvara kao u sledećem primeru:

```
ifstream tok("mojaDatoteka"); //otvaranje datoteke za upis
```

Ako iz nekog razloga datoteka ne može da se otvori, promenljiva `toka` imaće vrednost `false`.

Za zatvaranje datoteke koristi se funkcija članica `close()`. Na primer, da bi se zatvorila datoteka povezana sa tokom `mojTok`, koristi se sledeća naredba:

```
tok.close();
```

Funkcija `close()` nema parametara i ne vraća nikakvu vrednost.

#### 8.4.1. Čitanje i upis u tekstualne datoteke

Čitanje i upis u tekstualne datoteke je vrlo jednostavno: treba samo upotrebiti operatore `<<` i `>>` na isti način kao što smo ih dosada koristili za ispis i učitavanje podataka sa konzole. Jedina razlika je u tome što umesto objekata `cin` i `cout` treba upotrebiti tok koji je povezan sa datotekom. Na primer, sledeća datoteka pravi kratak inventar sa nazivom i cenom svake stavke:

```

#include <iostream>
#include <fstream>

using namespace std;

int main() {
 ofstream out("INVENTAR"); //izlazna datoteka

```

```

 if(!out) {

 cout << "Datoteka za inventar ne moza da se otvori" << endl;
 return 1;
 }
 out << "Plejeri " << 39.95 << endl;
 out << "Tosteri " << 19.95 << endl;
 out << "Mikseri " << 24.80 << endl;
 out << "Pegle " << 44.90 << endl;
 out.close();
 return 0;
}

```

Kada čitamo podatke iz datoteke, ona mora već da postoji. Prilikom čitanja treba proveravati da li se stiglo do kraja datoteke (EOF, end of file). Funkcija članica `eof()` klase `istream` vraća `true` kada se stigne do kraja datoteke. Za čitanje podataka možemo koristiti operator `>>` (nasleđen iz klase `istream`) koji prekida čitanje kada naiđe na belinu (razmak, tabulator i sl.), ili funkciju `getline()` koja učitava sadržaj iz datoteke red po red.

Funkcija `getline()` je članica svih klasa ulaznih tokova, a njeni prototipovi su:

```

istream &getline(char *buf, streamsize num);
istream &getline(char *buf, streamsize num, char delim);

```

Prva verzija učitava znake u niz na koji pokazuje `buf` sve dok se ne učitava `num-1` znakova, dok ne naiđe znak za novi red ili kraj datoteke (EOF, end of file). Funkcija `getline()` će automatski dodati nulu na kraj učitano g niza koji se smešta u `buf`. Ako se u ulaznom toku naiđe na znak za nov red, on se ne stavlja u `buf`. Druga verzija funkcije `getline()` učitava znakove u niz `buf` dok ne učitava `num-1` znakova, ne pronađe znak koji je zadat kao `delim` ili naiđe na kraj datoteke. Funkcija `getline()` će automatski dodati nulu na kraj učitano g niza koji se smešta u `buf`. Ako se znak za razdvajanje (`delim`) pronađe u ulaznom toku, on se ne stavlja u `buf`.

Kada se upisuje u datoteku, podaci se ne upisuju odmah u fizički uređaj povezan sa tokom, već se informacije čuvaju u internom baferu dok se on ne napuni. Tek tada se sadržaj bafera upisuje na disk. Međutim, fizičko upisivanje podataka na disk se može aktivirati i pre nego što je bafer pun pozivanjem funkcije `flush()` čiji je prototip:

```
ostream &flush();
```

Funkcija `flush()` posebno je korisna kada se program koristi u "nesigurnim" oruženjima (npr. u sistemima gde često nestaje struje i sl.)

U nastavku je dat primer programa koji otvara istu datoteku za upis i čitanje:

```

#include <iostream>
#include <fstream>

using namespace std;

```

```
int main() {
 char bafer[200];
 cout << "unesite ime i adresu: ";
 cin.getline(bafer, 200);
 cout << endl;
 ofstream outfile("ime.txt", ios::app);
 if(!outfile) {
 cerr << endl;
 cerr << "greska! datoteka ne moze da se otvori " <<
 "u rezimu dodavanja!";
 cerr << endl;
 return 1;
 };
 outfile << bafer << endl;
 outfile.close();
 ifstream infile("ime.txt");
 while(!infile.eof()) {
 infile.getline(bafer, 200);
 cout << bafer << endl;
 }
 infile.close();
 return 0;
}
```

## Pitanja

1. Većini C++ datoteka koji obavljaju U/I operacije potrebno je zaglavlje \_\_\_\_\_ koje sadrži deklaracije potrebne za sve operacije sa U/I tokovima.

- a) iostream
- b) fstream
- c) string

2. Zaglavlje \_\_\_\_\_ sadrži deklaracije potrebne za rad sa datotekama.

- a) fstream
- b) iostream
- c) string

3. Simbol za umetanje u tok je

- a) <<
- b) >>

4. Funkcija članica tokova `fstream`, `ifstream` i `ofstream` koja zatvara datoteku je:
- a) `fclose`
  - b) `close`
  - c) `fclose`
5. Programer mora eksplicitno da kreira objekte `cin`, `cout`, `cerr` i `clog`.
- a) tačno
  - b) pogrešno
6. Program mora eksplicitno da pozove funkciju `close` da bi zatvorio datoteku povezanu sa objektima toka `ifstream`, `ofstream` ili `fstream`.
- a) tačno
  - b) pogrešno
7. Objekti toka `ifstream` podrazumevano prave datoteku ako ona prilikom otvaranja ne postoji.
- a) tačno
  - b) pogrešno
8. Iste tehnike formatiranja izlaza koje se koriste sa objektom `cout` mogu se koristiti i sa objektima tokova za datoteke.
- a) tačno
  - b) pogrešno
9. Operator `>>` se može koristiti za upis u datoteku.
- a) tačno
  - b) pogrešno
10. Kada se stigne do kraja datoteke, funkcija članica `eof()` vraća:
- a) `true`
  - b) `false`
  - c) `0`
11. Moguće je otvoriti datoteku i za upis i za čitanje.
- a) tačno
  - b) pogrešno



Dodatak A: ASCII kôdovi znakova

| hex | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9  | A   | B   | C  | D  | E  | F   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|----|-----|-----|----|----|----|-----|
| 0   | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS  | HT | LF  | VT  | FF | CR | SO | SI  |
| 1   | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US  |
| 2   |     | !   | “   | #   | \$  | %   | &   | ,   | (   | )  | *   | +   | ,  | -  | .  | /   |
| 3   | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9  | :   | ;   | <  | =  | >  | ?   |
| 4   | @   | A   | B   | C   | D   | E   | F   | G   | H   | I  | J   | K   | L  | M  | N  | O   |
| 5   | P   | Q   | R   | S   | T   | U   | V   | W   | X   | Y  | Z   | [   | \  | ]  | ^  | _   |
| 6   | ,   | a   | b   | c   | d   | e   | f   | g   | h   | i  | j   | k   | l  | m  | n  | o   |
| 7   | p   | q   | r   | s   | t   | u   | v   | w   | x   | y  | z   | {   |    | }  | ~  | DEL |



## Dodatak B: Preprocesorske direktive

Preprocesor je deo prevodioca (kompajlera) koji radi sa izvornim kôdom (tekstom) pre nego što se on prevede u objektni kôd. Preprocesoru se navode komande za rad sa tekstom koje se zovu *preprocesorske direktive*. Iako one tehnički nisu deo jezika C++, proširuju njegovu sintaksu. Preprocesor je nasleđen iz jezika C i u jeziku C++ nije toliko važan kao u jeziku C; takođe, neke funkcije preprocesora u jeziku C++ zamenjene su boljim elementima jezika. Ipak, pošto se često sreću u C++ kôdu, najvažnije direktive biće ukratko opisane u nastavku.

Sve preprocesorske direktive počinju znakom `#`. C++ preprocesor sadrži sledeće direktive:

|                      |                     |                       |
|----------------------|---------------------|-----------------------|
| <code>#define</code> | <code>#error</code> | <code>#include</code> |
| <code>#if</code>     | <code>#else</code>  | <code>#elif</code>    |
| <code>#endif</code>  | <code>#ifdef</code> | <code>#ifndef</code>  |
| <code>#undef</code>  | <code>#line</code>  | <code>#pragma</code>  |

### Direktiva `#define`

Direktiva `#define` koristi se za definisanje identifikatora i niza znakova koji će biti zamenjeni tim identifikatorom kad god se on pronađe u datoteci sa izvornim kôdom. Opšti oblik direktive je

```
#define ime_makroa niz_znakova
```

Između identifikatora i početka niza znakova može da bude proizvoljan broj praznina, ali kada niz znakova počne, završava se tek prelaskom u sledeći red. Na primer, ako želimo da koristimo reč "GORE" za vrednost 1 i reč "DOLE" za vrednost 0, to bi deklarirali ovako:

```
#define GORE 1
#define DOLE 0
```

Ove naredbe naterale bi kompajler da reči GORE i DOLE zameni sa 1 ili 0 kad god ih pronađe u datoteci.

Ako je niz znakova duži od jednog reda, može se nastaviti tako što će se na kraj reda staviti obrnuta kosa crta, kao u sledećem primeru:

```
#define DUGACAK_STRING "ovo je veoma dugacak \
string koji služi kao primer"
```

C++ programeri često koriste sva velika slova za nazive makroa, jer to olakšava prepoznavanje mesta na kojima će nastupiti zamena. Takođe, preporučljivo je sve direktive `#define` staviti na početak datoteke, ili u posebno zaglavlje, radi bolje preglednosti. Treba znati da C++ umesto korišćenja ove direktive nudi bolji način za definisanje konstanti, a to je ključna reč `const`. Međutim, pošto su mnogi C++ programeri prešli sa jezika C, u kome za tu namenu služi direktiva `#define`, ona se često sreće i u C++ kôdu.

## Makroi funkcija

Ime makroa u direktivi `#define` može i da ima argumente. U tom slučaju, kad god preprocesor naiđe na ime makroa, zamenjuje njegove argumente stvarnim argumentima iz programa. Tipičan primer je makro za traženje manjeg od dva argumenta:

```
#define MIN(a,b) (a < b ? a : b)
```

Makroi funkcija koriste se za definisanje funkcija čiji se kôd razvija na licu mesta, umesto da se pozivaju. Naizgled nepotrebne zagrade oko makroa `MIN` neophodne su da bi se obezbedilo pravilno tumačenje izraza koji se zamenjuje zbog prioriteta operatora. Iako se još uvek često sreću u C++ kdu, makroi funkcija su potpuno nepotrebni jer za istu namenu služi specifikator `inline`, koji je bolji i bezbedniji za korišćenje jer ne zahteva dodatne zagrade.

## Direktiva `#error`

Kada naiđe na direktivu `#error`, kompajler prekida prevođenje. Ova direktiva se koristi prvenstveno za debugovanje, a opšti oblik joj je

```
#error error_message
```

Uočite da se poruka o grešci (`error-message`) ne piše između navodnika. Kada kompajler naiđe na ovu direktivu, prikazuje poruku `error_message` i prekida prevođenje.

## Direktiva `#include`

Preprocesorska direktiva `#include` kompajleru nalaže da uključi standardno zaglavlje ili neku drugu datoteku sa izvornim kôdom u datoteku u kojoj je navedena direktiva `#include`. Naziv standardnog zaglavlja treba da se nalazi između znakova `<` `>` (npr. `#include <fstream>` uključuje standardno zaglavlje za rad sa datotekama). Ako se uključuje neka druga datoteka, njeno ime navodi se između navodnika (npr. `#include "sample.h"`). Kada se uključuje datoteka, način na koji se navodi (između znakova `<` `>` ili pod navodnicima) određuje mesto gde će se tražiti datoteka. Ako je naziv datoteke naveden između znakova `<` `>`, kompajler će je tražiti u nekoliko predefinisanih direktorijuma. Ako je naziv naveden pod navodnicima, ona se obično traži

u tekućem radnom direktorijumu (a ako se tamo ne pronađe, onda se traži kao da je navedena pod znacima < >).

## Direktive za uslovno prevođenje

Postoji nekoliko direktiva koje omogućuju selektivno kompajliranje delova izvornog kôda. Ovakav pristup, koji se zove uslovno prevođenje (conditional compilation), često se koristi u softverskim kućama koje održavaju veliki broj prilagođenih verzija istog programa.

### Direktive `#if`, `#else`, `#elif`, and `#endif`

Ako konstantan izraz iza direktive `#if` ima vrednost `true`, kôd koji se nalazi između ove i direktive `#endif` će se kompajlirati; u suprotnom, kompajler će ga preskočiti. Direktiva `#endif` koristi se za označavanje kraja `#if` bloka. Opšti oblik direktive `#if` je

```
#if konstantan_izraz niz naredbi #endif
```

### Direktive `#ifdef` and `#ifndef`

Drugi metod za uslovno prevođenje su direktive `#ifdef` i `#ifndef`, koje znače “ako je definisano” i “ako nije definisano,” redom, a odnose se na imena makroa. Opšti oblik direktive `#ifdef` je

```
#ifdef ime_makroa niz naredbi #endif
```

Ako je ime makroa prethodno definisano pomoću direktive `#define`, niz naredbi između `#ifdef` i `#endif` će se kompajlirati. Opšti oblik direktive `#ifndef` je

```
#ifndef ime_makroa niz naredbi #endif
```

Ako ime makroa trenutno nije definisano pomoću direktive `#define`, blok kôda će se kompajlirati.

Za utvrđivanje da li je ime makroa definisano osim direktive `#ifdef` postoji još jedan način. Direktiva `#if` može se koristiti zajedno sa operatorom. Na primer, da bi se odredilo da li je definisan makro `MOJADATOTEKA`, može se upotrebiti bilo koja od sledeće dve preprocesorske direktive:

```
#if defined MOJADATOTEKA
#ifdef MOJADATOTEKA
```

Direktiva za uslovno prevođenje `#ifndef` često se koristi za izbegavanje višestrukog uključivanja zaglavlja u program (tzv. include guard). Ta tehnika će biti objašnjena na sledećem primeru:

```
#ifndef PRIMER_H
#define PRIMER_H
 //sadrzaj datoteke zaglavlja primer.h
#endif
```

Kada se prvi put nađe na direktivu `#include primer.h`, definisaće se konstanta `PRIMER`. Kada se sledeći put nađe na direktivu `#include primer.h`, preskočiće se ceo sadržaj zaglavlja jer je konstanta `PRIMER_H` već definisana. Tako se obezbeđuje da se zaglavlje u program uključi samo jednom.

### **#undef**

Direktiva `#undef` koristi se za uklanjanje definicije imena makroa koji je prethodno definisan u direktivi `#define`. Opšti oblik ove direktive je

```
#undef ime_makroa
```

Na primer:

```
#define TIMEOUT 100
```

```
#define WAIT 0 // ...
```

```
#undef TIMEOUT
```

```
#undef WAIT
```

U ovom slučaju i `TIMEOUT` i `WAIT` su definisani dok se ne nađe na direktive `#undef`. Najvažnija primena direktive `#undef` jeste za lokalizovanje makroa samo na delove kôda kojima su oni potrebni.

## Dodatak C: Prevođenje programa sa više datoteka

Mnogi programi koje smo dosad pisali nalazili su se u jednoj datoteci. Međutim, kako programi postaju veći, postaje neophodno da se oni podele u više datoteka. U opštem slučaju, program sa više datoteka sastoji se od dva tipa datoteka: jednih koje sadrže definicije funkcija, i drugih koje sadrže prototipove (deklaracije) funkcija. Obično se prilikom deljenja programa u datoteke primenjuje sledeća strategija:

- Sve specijalizovane funkcije koje izvršavaju slične zadatke grupišu se u istu datoteku. Na primer, pravi se datoteka koja sadrži funkcije za matematičke operacije, datoteka koja sadrži funkcije za unos i ispis podataka i sl.
- Funkcija `main()` i sve druge funkcije koje imaju veoma važnu ulogu u programu grupišu se u jednu datoteku.
- Za svaku datoteku sa definicijama funkcija (.cpp) pravi se posebna datoteka zaglavlje (.h) u kojoj se nalaze prototipovi svih funkcija.

Svaka datoteka čije se ime završava sa .cpp sadrži definicije funkcija. Svaka .cpp datoteka ima odgovarajuću datoteku zaglavlje koja se završava sa .h. Zaglavlje sadrži prototipove svih funkcija koje se nalaze u odgovarajućoj .cpp datoteci. Svaka .cpp datoteka ima direktivu `#include` kojom čita sopstveno zaglavlje (odgovarajuću .h datoteku). Ako .cpp datoteka poziva funkcije iz neke druge .cpp datoteke, onda će imati direktivu `#include` za čitanje zaglavlja tih funkcija.

Sve .cpp datoteke kompajliraju se u zasebne objektne datoteke, koje se zatim povezuju u jednu izvršnu datoteku. Taj postupak se u IDE okruženjima obavlja automatski unutar projekta (može se postići i pomoću uslužnog programa `make` i odgovarajućih datoteka `Makefile`).

Doseg (scope) globalnih promenljivih je isključivo datoteka u kojoj su definisane. Da bi globalna promenljiva definisana u datoteci A bila dostupna u datoteci B, datoteka B mora da sadrži deklaraciju te promenljive ispred koje se nalazi ključna reč `extern`. Deklaracija `extern` ne definiše drugu promenljivu, već proširuje doseg postojeće globalne promenljive.

Ako je globalna promenljiva definisana kao statička, ona nikako ne može da bude vidljiva izvan datoteke u kojoj je definisana. Ključna reč `static` u ovom slučaju se koristi da bi globalna promenljiva bila privatna za datoteku u kojoj je definisana.

Deklaracije klasa i definicije funkcija članica obično se čuvaju u zasebnim datotekama. Obično se to radi na sledeći način:

- Deklaracija klase smešta se u posebno zaglavlje, koje se zove specifikacija klase. Ime tog zaglavlja obično je isto kao ime klase, sa nastavkom `.h`.
- Definicije funkcija članica klase čuvaju se u posebnoj `.cpp` datoteci, koja se zove implementaciona datoteka. Ime te datoteke obično je isto kao ime klase, sa nastavkom `.cpp`.
- Svai program koji koristi klasu treba da ima direktivu `#include` za specifikaciju klase. Datoteka `.cpp` sa implementacijom klase trebalo bi da se kompajlira i povezuje sa programom `main`, što se u IDE okruženjima obavlja automatski.



## Dodatak D: Poređenje jezika C++ i Java

Ovaj dodatak namenjen je čitaocima koji već poznaju programski jezik Java, a prelaze na C++. Bez obzira da li je C++ prvi programski jezik koji savladavate ili ne, sasvim je izvesno da neće biti i poslednji. Jezici C++ i Java trenutno su dva objektno orijentisana jezika koja se najčešće koriste, a pošto je Java nastala po ugledu na C++, mnoge osobine su im slične. U nastavku će biti prikazane samo razlike između ova dva jezika; pošto su na primer, naredbe za kontrolu toka programa (`if`, `while`, `for`) u suštini identične u oba jezika, o njima neće biti reči.

### Java je potpuno objektno orijentisan jezik

To znači da se sve operacije odvijaju isključivo preko objekata, odnosno preko metoda koji se na njih primenjuju. Zbog toga je programer prisiljen da od početka razmišlja na objektno orijentisan način. S druge strane, jezik C++ omogućuje i pisanje proceduralnog kôda, zbog kompatibilnosti sa jezikom C. On je projektovan tako da ne bude potpuno nov jezik, već tako da podrži kôd u jeziku C koji je osamdesetih godina bio dominantan. Java je potpuno nov jezik, neopterećen nasleđem, pa je mnogo toga u Javi elegantnije rešeno.

### Javin kôd se izvodi na virtuelnoj mašini

Java kôd se ne prevodi u izvorni kôd mašine na kojoj se izvršava, kao C++ kôd, već u tzv. Java bajtkod koji Javina virtuelna mašina interpretira i izvršava. Prednost ovog pristupa je to što se bajtkod može izvršavati na bilo kom sistemu na kome je instalirana virtuelna mašina. Nedostatak je sporost u poređenju sa C++ kôdom koji se prevodi (kompajlira) i izvršava direktno u procesoru.

### Tipovi podataka i promenljivih

Tipovi podataka u jezicima C++ i Java su slični. Kao i Java, C++ ima tipove `int` i `double`. Međutim, opseg numeričkih tipova kao što je `int` zavisi od računara. Dok Java propisuje da integer zauzima tačno 32 bita, u zavisnosti od mašine na kojoj je implementiran, ceo broj u jeziku C++ može da zauzima 16, 20, 32, čak i 64 bita.

C++ ima i tipove `short` i `unsigned` koji omogućuju efikasnije čuvanje brojeva, ali ih je bolje izbegavati osim u slučajevima kada je prednost koju nude od kritične važnosti.

Logički tip (Boolean ili boolean u Javi), u jeziku C++ se zove `bool`.

Tip znakovnog niza (`string`) u jeziku C++ se zove `string` i dosta liči na Javin tip `String`. Ipak, treba imati u vidu sledeće razlike:

1. C++ stringovi čuvaju 8-bitne ASCII, a ne 16-bitne Unicode znakove (kao u Javi)
2. C++ stringovi se mogu menjati, dok su Javini objekti klase `String` nepromenljivi.
3. Mnoge metode Javine klase `String` postoje i u klasi `string` jezika C++, ali su im nazivi različiti.
4. Stringovi u jeziku C++ mogu se nadovezivati isključivo na druge stringove, a ne na objekte proizvoljnog tipa kao u Javi.
5. Za poređenje stringova u jeziku C++ koriste se relacioni operatori `==` `!=` `<` `<=` `>` `>=` koji obavljaju leksičko poređenje. To je u suštini prirodnije od korišćenja metoda `equals` i `compareTo` u Javi.

### Promenljive i konstante

C++ prevodilac ne proverava da li su sve lokalne promenljive inicijalizovane pre nego što se prvi put upotrebe, kao što je slučaj u Javi. Prilično je lako zaboraviti na inicijalizaciju promenljive u jeziku C++, a njena vrednost je u tom slučaju slučajan sadržaj memorijske lokacije koja joj je dodeljena. Očigledno je da je to pogodno tle za generisanje grešaka u programiranju.

Kao i u Javi, i u jeziku C++ klase mogu da sadrže podatke članove i statičke promenljive. Osim toga, promenljive mogu da budu deklarisanе i izvan metoda i klase. Takozvanim globalnim promenljivama može se pristupiti iz bilo koje funkcije (metoda) u programu, što otežava upravljanje programima i zbog toga bi ih valjalo izbegavati.

U jeziku C++ konstante se mogu deklarirati na bilo kom mestu, za razliku od Jave gde konstante moraju da budu statički podaci članovi klase. C++ koristi ključnu reč `const` umesto Javine ključne reči `final`.

### Klase

Definicija klase u jeziku C++ donekle se razlikuje od one u Javi.

1. U jeziku C++, razdvojeni su javni i privatni delovi klase, označeni ključnim rečima `public` i `private`. U Javi, svaka pojedinačna stavka mora da bude označena kao `public` ili `private`.
2. Definicija klase u jeziku C++ sadrži samo deklaracije metoda, a stvarne implementacije (tj. definicije metoda) pišu se izvan deklaracije klase.
3. Inspektorske metode (koje ne menjaju podatke članove, tj. argumente funkcija) označavaju se ključnom reči `const`
4. Na kraju deklaracije klase u jeziku C++ piše se tačka-zarez.
5. Implementacija metoda piše se nakon deklaracije klase. Pošto se metode defi-

nišu izvan klase, ispred naziva svakog metoda navodi se ime klase. Operator razrešavanja dosega `::` razdvaja ime klase i ime metoda.

## Objekti

Najvažnija razlika između jezika Java i C++ je u ponašanju objektnih promenljivih. U jeziku C++ objektne promenljive sadrže vrednosti, a ne reference na objekte. Operator `new` nikada se ne koristi za konstruisanje objekata u jeziku C++, već se argumenti konstruktora navode nakon imena promenljive. Ako se ne navedu parametri konstruktora, objekat se kreira pozivom podrazumevanog konstruktora.

To je veoma važna razlika u odnosu na Javu. Ako je Tacka korisnički definisana klasa, onda bi naredba:

```
Tacka t;
```

u Javi napravila samo neinicijalizovanu referencu, dok u jeziku C++ konstruiše stvarni objekat.

Kada se jedan objekat dodeljuje drugome, u jeziku C++ pravi se kopija njegovih stvarnih vrednosti, dok se u Javi kopiranjem objekta pravi samo još jedna referenca na isti objekat. Kopiranje u jeziku C++ zapravo je ono što se postiže pozivom metode `clone` u Javi: menjanje kopije ne utiče na original. U većini slučajeva, činjenica da se objekti ponašaju kao vrednosti je pogodna, ali postoje i situacije kada ovakvo ponašanje nije poželjno.

1. Kada se menja objekat u funkciji, programer mora da se seti da koristi poziv preko reference.

2. Dve objektne promenljive ne mogu zajednički da pristupaju istom objektu; ako je potrebno postići takav efekat u jeziku C++ , moraju da se koriste pokazivači.

3. Objektne promenljive može da sadrži samo objekte određenog tipa; ako treba da sadrži i objekte tipa izvedenih klasa, moraju da se upotrebe pokazivači

4. Ako neka promenljiva treba da pokazuje na `null` ili na stvarni objekat, u jeziku C++ moraju da se koriste pokazivači.

## Funkcije

U Javi, svaka funkcija mora da bude ili funkcija članica (metod) klase, ili statička funkcija klase. C++ podržava oba pomenuta tipa metoda, ali dozvoljava i funkcije koje nisu sastavni deo nijedne klase. Takve funkcije zovu se globalne funkcije.

Izvršavanje svake C++ funkcije počinje iz globalne funkcije `main()`. Povratni tip funkcije `main()` u jeziku C++ je `int` (0 ako je program uspešno izvršen, neki drugi ceo broj ako nije).

U Javi, funkcije mogu da promene vrednost objekata koji im se prenose kao stvarni argumenti prilikom poziva. U jeziku C++ postoje dve vrste prenosa argumenata u funkcije: po vrednosti i po referenci. Kod prenosa po vrednosti (`call-by-value`), pravi se kopija stvarnih argumenata koji su korišćeni kao stvarni argumenti prilikom poziva funkcije, što znači da funkcija nikako ne može da promeni stvarni argument. Drugi

mehanizam prenosa je poziv po referenci (call-by-reference); u tom slučaju funkcija može da promeni vrednost svog stvarnog argumenta. Poziv po referenci u jeziku C++ označava se navođenjem znaka & iza tipa argumenta.

```
void povecajPlatu(Zaposleni& e, double zaKoliko){
 ...
}
```

Java ne podržava preklapanje operatora.

### Pokazivači i upravljanje memorijom

Već smo napomenuli da u jeziku C++ objektne promenljive sadrže vrednosti objekata, za razliku od Jave u kojoj objektne promenljive sadrže referencu na objekat. Postoje situacije kada je tako nešto potrebno i u jeziku C++, i za to služe pokazivači. Promenljiva koja pokazuje na neki objekat (odnosno sadrži adresu tog objekta u memoriji) u jeziku C++ zove se pokazivač. Ako je T neki tip (prosti ili klasni), onda je T\* tip pokazivača na objekat tipa T.

Slično kao u Javi, pokazivačka promenljiva može se inicijalizovati vrednošću NULL, drugim pokazivačem ili pozivom operatora new.

```
Zaposleni* p = NULL;
Zaposleni* q = new Zaposleni("Petar, Petrovic", 35000);
Zaposleni* r = q;
```

Četvrta mogućnost jeste da se pokazivač inicijalizuje adresom drugog objekta, korišćenjem operatora &.

```
Zaposleni sef("Nikola, Vukovic", 83000);
Zaposleni* s = &sef;
```

To obično nije preporučljivo. Zlatno pravilo koga se treba držati jeste da C++ pokazivači treba da pokazuju isključivo na objekte alocirane pomoću operatora new.

Dosad, C++ pokazivači veoma liče na Javine objektne promenljive. Postoji međutim jedna velika razlika u sintaksi: da bi se pristupilo objektu na koji pokazuje pokazivač, u jeziku C++ mora se upotrebiti operator \*. Ako je p pokazivač na objekat Zaposleni, onda \*p pokazuje na taj objekat.

```
Zaposleni* p = . . .;
Zaposleni sef = *p;
```

Sintaksa \*p mora da se koristi i kada se izvršava metod ili se pristupa podatku članu.

```
(*p).zadajPlatu(91000);
```

Zagrade su neophodne zato što je operator . većeg prioriteta od operatora \*. Da je to ružno, primetili su i projektanti jezika C, pa su uveli alternativni operator -> koji služi za kombinovanje operatora \* i ., pa tako izraz:

```
p->zadajPlatu(91000);
```

poziva metod zadajPlatu za objekat \*p.

Ako se pokazivač u jeziku C++ ne inicijalizuje, ima vrednost NULL ili pokazuje na neki objekat koji više ne postoji, primena operatora \* ili -> prouzrokuje grešku. Tokom izvršavanja C++ programa takve greške se ne proveravaju, pa se dešava da program “umire u mukama” ili ne radi kako bi trebalo.

U Javi takve greške nisu moguće, jer ne može da postoji referenca koja nije inicijalizovana. Objekti se održavaju “u životu” dogod postoji referenca na njih, što znači da ne može da postoji referenca na obrisani objekat. Tokom izvršavanja Java programa proverava se da li postoje null reference i baca se izuzetak ako se to otkrije.

U Javu je ugrađen “sakupljač smeća” (engl. *garbage collector*) koji automatski iz memorije uklanja sve objekte koji više nisu potrebni. U jeziku C++, programer je odgovoran za upravljanje memorijom.

Objektne promenljive u jeziku C++ automatski nestaju kada prestane njihova oblast važnosti, ali objekti napravljeni pomoću operatora new moraju se ručno ukloniti pozivom operatora delete.

```
Zaposleni* p = new Zaposleni("Petar, Petrovic", 38000);
...
delete p; /* ovaj objekat vise nije potreban*/
```

Ako se objekat napravljen pomoću operatora new ne ukloni pomoću operatora delete, može se desiti da se “potroši” sva memorija koja je na raspolaganju programu. Takvi problemi zovu se “curenje memorije” (memory leak). Što je još važnije, ako se neki objekat ukloni, a zatim nastavi da se koristi, može se desiti da se promene podaci koji više ne pripadaju programu. Ako se prepisu memorijske lokacije koje su bile korišćene za dinamičko alociranje, mehanizmi za alokaciju mogu da se kompromituju i da prouzrokuju greške koje se veoma teško pronalaze i ispravljaju. Zbog toga treba biti vrlo oprezan pri radu sa pokazivačima u jeziku C++.

## Nasledivanje

U jeziku C++, za označavanje nasleđivanja koristi se sintaksa : public umesto ključne reči extends. (C++ podržava i koncept privatnog nasleđivanja, ali on nije preterano koristan jer se isti efekat može postići i kompozicijom.) Funkcije u jeziku C++ nisu podrazumevano sa dinamičkim vezivanjem; ako to želite, funkciju morate da označite kao virtuelnu (navođenjem ključne reči virtual).

```
class Direktor : public Zaposleni {
private:
 string odeljenje;
public:
 Direktor(string ime, double plata, string odeljenje);
 virtual void print() const;
};
```

Kao i u Javi, u jeziku C++ postoji posebna sintaksa za poziv konstruktora roditeljske klase. U Javi se koristi ključna reč `super`, a u jeziku C++ konstruktor roditeljske klase mora se pozvati izvan tela konstruktora klase-naslednice, kao u sledećem primeru.

```
Direktor::Direktor(string ime, double plata, string od):
 Zaposleni(ime, plata){
 odeljenje = od;
 }
```

Java ključnu reč `super` koristi i kada se iz metoda klase-naslednice poziva metod roditeljske klase. U jeziku C++, umesto toga se koriste ime roditeljske klase i operator `::`

```
void Direktor::print() const {
 Zaposleni::print();
 cout << odeljenje << "\n";
}
```

U jeziku C++ objektna promenljiva sadrži objekte određenog tipa (a ne reference na njih); da bi se iskoristio polimorfizam, u jeziku C++ su neophodni pokazivači. Pokazivač tipa `T*` može da pokazuje na objekte tipa `T`, ali i na objekte tipa bilo koje klase izvedene iz `T`.

U Javi ne postoji ključna reč `virtual`, već su sve metode klase implicitno virtuelne (mogu se nadjačavati).

U jeziku C++ ne postoji pojam interfejsa (`interface`) iz Jave; umesto njega, koristi se apstraktna klasa u kojoj su sve metode čiste virtuelne, a koja nema podataka članova.

I konačno, u jeziku C++ ne postoji korenska klasa (klasa `Object` u Javi), ali je omogućeno višestruko nasleđivanje.

## Dodatak E: Okruženje Microsoft Visual C++ 2010 Express

Ovaj dodatak služi kao kratko uputstvo za izvođenje sledećih operacija u integrisanom razvojnom okruženju (IDE) Microsoft Visual C++ 2010 Express koje se može besplatno preuzeti sa Interneta:

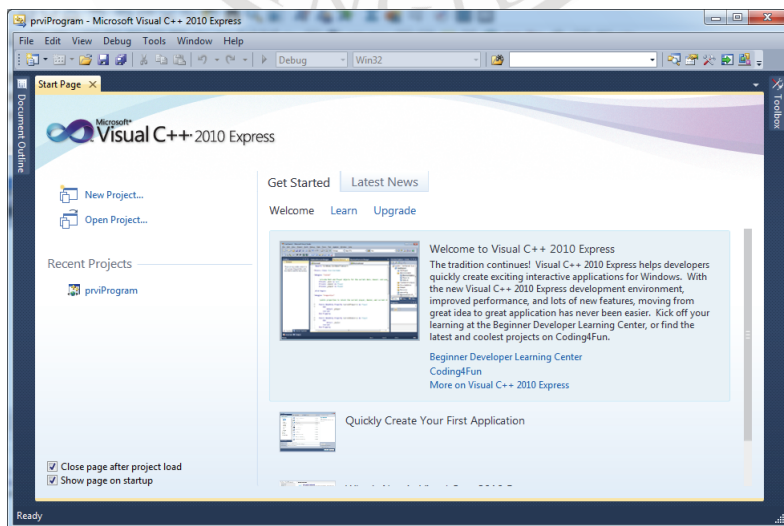
1. Pravljenje novog projekta i unošenje kôda
2. Snimanje projekta na disk
3. Kompajliranje i izvršavanje projekta
4. Otvaranje postojećeg projekta

Uputstvo se odnosi na pravljenje konzolnih programa koji su korišćeni za većinu primera u ovoj knjizi.

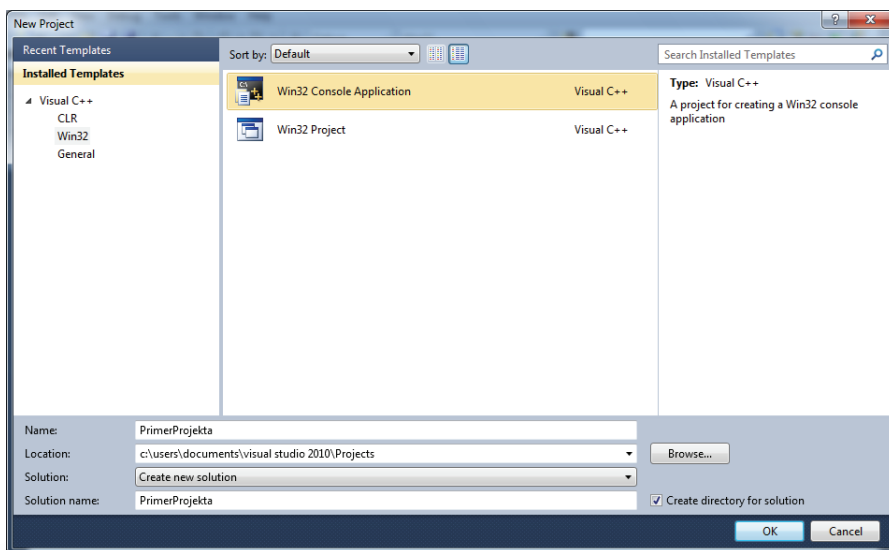
Prvi korak u pravljenju programa u okruženju Visual C++ jeste kreiranje projekta. Projekat (project) je skup datoteka koje čine program (aplikaciju). Čak i ako program ima samo jednu datoteku, ona mora da pripada projektu.

Da biste kreirali projekat:

1. Pokrenite Microsoft Visual C++ (iz menija *Start > All Programs*). Otvoriće se sledeći prozor:

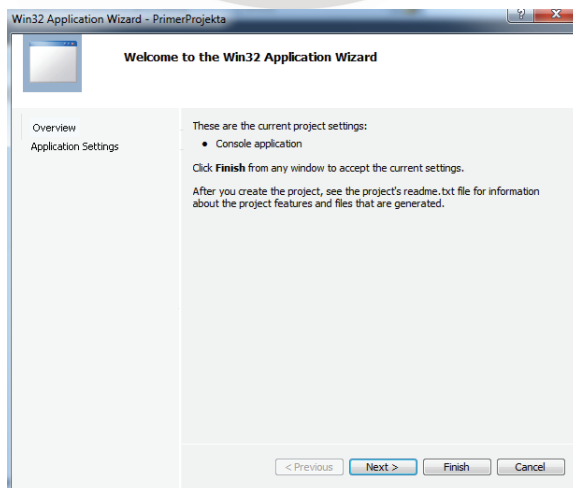


2. Izaberite *File* u liniji menija, a zatim *New > Project* (ili pritisnite opciju *New Project* na naslovnoj strani). Pojaviće se sledeći dijalog:



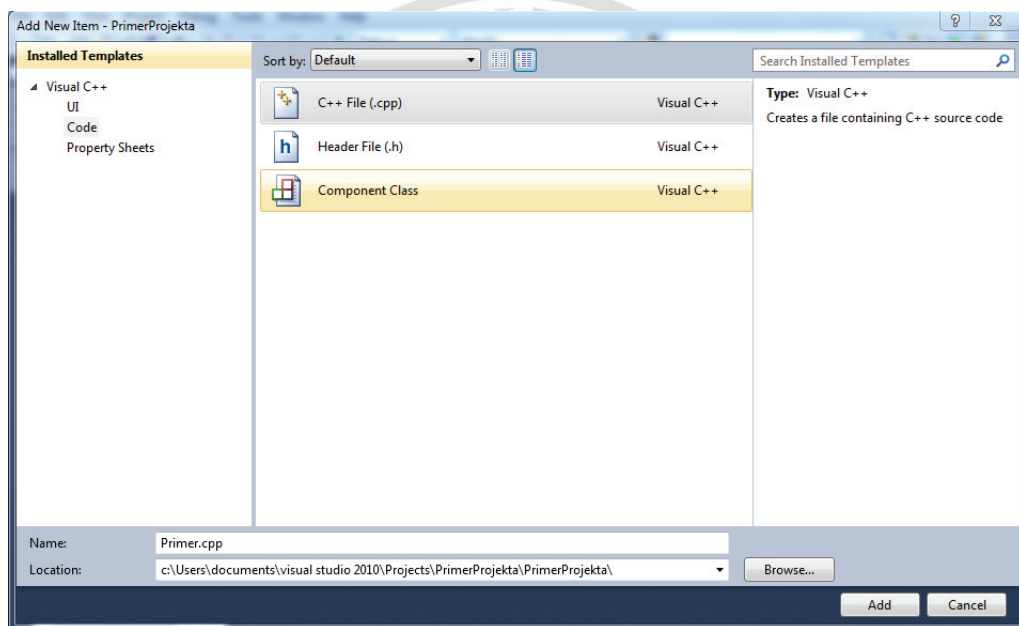
Pošto su svi programi u ovoj knjizi konzolni, u listi tipova projekata sa leve strane izaberite *Win32*, a zatim sa desne strane *Win32 Console Application*. Zatim unesite naziv projekta u polje *Name* (u ovom slučaju, *PrimerProjekta*). U polju *Location* pojaviće se naziv direktorijuma gde će projekat biti smešten. Standardna lokacija gde se smeštaju projekti može se promeniti pritiskom na dugme *Browse*.

3. Kada pritisnete dugme *OK*, pojaviće se dijalog za izbor podešavanja aplikacije; izaberite opciju *ApplicationSettings*.



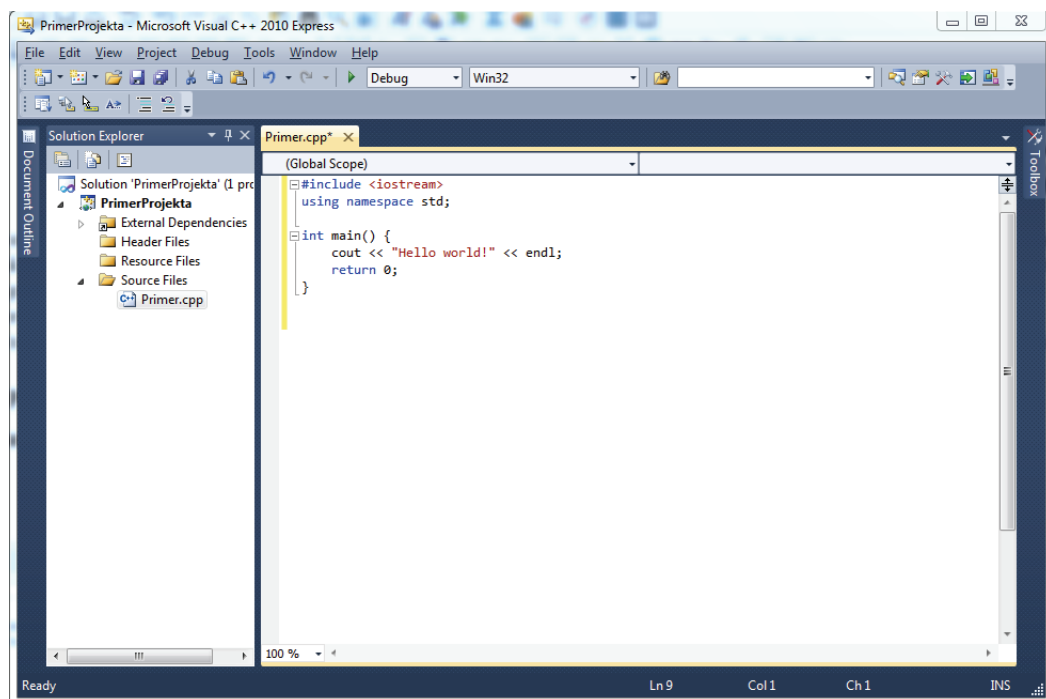
4. U dijalogu *Application Settings* izaberite opciju *Empty Project*, kao na slici; kada pritisnete *Finish*, vratićete se u glavni prozor okruženja koji će biti prazan. Ako nije prikazan prozor *Solution Explorer*, prikažite ga pritiskom na ikonu  u gornjem desnom uglu ekrana.

5. Sada treba napraviti novu datoteku sa izvornim kôdom u projektu. U prozoru *Solution Explorer* desnim tasterom miša pritisnite poslednju kategoriju (Source Files) i izaberite *Add > New Item ...* Pojaviće se sledeći dijalog:



U panou sa leve strane izaberite *Code*, a sa desne strane *C++ file (.cpp)*. U polje *Name* upišite ime .cpp datoteke (u ovom slučaju, *Primer*). Nije potrebno da kucate nastavak imena (.cpp), već samo naziv. Kada pritisnete dugme *Add*, vratićete se u glavni prozor okruženja. Ako želite da napravite datoteku zaglavlje (header), u ovom dijalogu na sličan način treba izabrati opciju *Header file (.h)*.

6. Sa desne strane sada je prikazan editor u kome možete da unosite izvorni kôd.



7. Datoteku sa kôdom koji ste uneli najlakše ćete snimiti na disk pomoću prečice Ctrl+S (ili izborom opcije *File > Save Primer.cpp* iz menija). Kada snimate datoteku, nestaće zvezdica pored njenog naziva u naslovnoj liniji.

8. Kada završite sa unosom kôda, možete da ga kompajlirate i izvršite sledećim metodama:

- pritiskanjem ikone *Debug* 
- pritiskanjem tastera Ctrl+F5

Prozor na dnu ekrana prikazivaće poruke o statusu tokom kompajliranja programa, kao i poruke o greškama.

9. Ako u projektu imate više datoteka, a ne želite sve uvek da ih kompajlirate, možete da kompajlirate pojedinačne datoteke koje su trenutno aktivne pomoću prečice Ctrl+F7.

10. U meniju *File* pronaći ćete i opcije za otvaranje postojećih projekata (*File > Open Project/Solution* ili *File > Recent Projects and Solutions*).

## Odgovori na pitanja

|    | Glava 1 | Glava 2 | Glava 3 | Glava 4 | Glava 5 | Glava 6 | Glava 7 | Glava 8 |
|----|---------|---------|---------|---------|---------|---------|---------|---------|
| 1  | a       | a, d    | c       | a       | c       | a       | c       | a       |
| 2  | a       | b, c, e | a       | b       | c       | a       | a       | a       |
| 3  | a       | a, b    | b       | a       | a, d    | a       | a       | a       |
| 4  | b       | b       | a       | a       | b       | b       | b       | b       |
| 5  | b       | a       | b       | c       | b       | b       | b       | b       |
| 6  | c       | a, c    | a       | b       | a       | b       | a       | a       |
| 7  | b       | f       | b       | a, d    | a       | b       | a       | b       |
| 8  | a       | b       | a       | b       | b       | a       | a       | a       |
| 9  | a       | a       | b       | b       | a       | a       | a       | a       |
| 10 | b       | a       | c       | a       | a       | b       | a       | a       |
| 11 | a       | b       | a       | b       | a       | c       | a       | a       |
| 12 | a       | a       | a       | c       | b       | a       | b       |         |
| 13 | a       | a       | a       | b       | b       | a       | a       |         |
| 14 | a       | a       | a       | b       | b       | b       | a       |         |
| 15 | b       | b       | b       | c       | a       | a       | a       |         |
| 16 | a       | a       | a       | a       | a       | a       |         |         |
| 17 | d       | a       | b       | c       | b       | a       |         |         |
| 18 | b       | b       | b       | b       | b       | c       |         |         |
| 19 | b       | c       | c       | b       | c       | a       |         |         |
| 20 |         | a, b, d | c       | a       | a       | b       |         |         |
| 21 |         | c       | c       | a       | b       | c       |         |         |
| 22 |         | b       | c       | a       | c       | b       |         |         |
| 23 |         | b       | b       | a       | a       | b       |         |         |
| 24 |         | a       |         | c       | b       | b       |         |         |
| 25 |         | b       |         | b       | a       | b       |         |         |
| 26 |         | b, d    |         | c       | b       | a       |         |         |
| 27 |         | a       |         | c       | b       | a       |         |         |
| 28 |         | a       |         | c       | b       | b       |         |         |

|    |  |         |  |      |   |   |  |  |
|----|--|---------|--|------|---|---|--|--|
| 29 |  | a       |  | a    | b | b |  |  |
| 30 |  | b       |  | b    | a | a |  |  |
| 31 |  | c       |  | b    |   | b |  |  |
| 32 |  | b       |  | a    |   | b |  |  |
| 33 |  | c       |  | b    |   | b |  |  |
| 34 |  | c       |  | a    |   | a |  |  |
| 35 |  | a, c    |  | a    |   | a |  |  |
| 36 |  | a       |  | a, c |   | b |  |  |
| 37 |  | a       |  | a    |   | a |  |  |
| 38 |  | b       |  | a    |   | b |  |  |
| 39 |  | c       |  | a    |   | a |  |  |
| 40 |  | b       |  | b    |   |   |  |  |
| 41 |  | a       |  |      |   |   |  |  |
| 42 |  | a, c, e |  |      |   |   |  |  |
| 43 |  | b       |  |      |   |   |  |  |
| 44 |  | b       |  |      |   |   |  |  |
| 45 |  | c       |  |      |   |   |  |  |
| 46 |  | c       |  |      |   |   |  |  |
| 47 |  | b       |  |      |   |   |  |  |

## Bibliografija

- [1] Cay Horstmann and Timothy Budd. *Big C++*. John Wiley and Sons, 2005.
- [2] Ivor Horton. *Beginning C++*. John Wiley and Sons, 2006.
- [3] Herbert Schildt. *C++ The Complete Reference*. McGraw Hill, 3rd Edition, 2003.
- [4] Herbert Schildt. *The art of C++*. McGraw-Hill, 2004.
- [5] John Moluzzo. *C++ for Business Programming*. 2nd Edition, Prentice Hall, 2005.
- [6] Deitel. *C++ How To Program*. 5<sup>th</sup> Edition, Prentice Hall, 2005.
- [7] Stanley Lipmann, *C++ Primer*. 4<sup>th</sup> Edition, Addison-Wesley, 2005.
- [8] Julijan Šribar i Boris Motik. *Demistificirani C++*. Element, 2010.
- [9] Bjarne Stroustrup. *The C++ programming language (Special Edition)*. Addison-Wesley, 2000.
- [10] Dragan Milićev. *Objektno orijentisano programiranje na jeziku C++*. Mikro-knjiga, 1995.
- [11] Ranko Popović i Zona Kostić. *Programski jezik C++ sa rešenim zadacima*. Univerzitet Singidunum, 2010.
- [12] Jeffrey Richter. *Programming Applications for Windows*. Microsoft Press, 1999.



# Indeks

ANSI/ISO standard, 11  
apstraktna klasa, 167  
ASCII kôd, 29, 30

beskonačna petlja, 58  
bezuslovni skok, 62  
biblioteke kôda, 13  
blokovi naredbi, 49  
bool, 30  
break, 55

C stringovi, 70

cerr, 173

char, 29

cin, 71, 173

clog, 173

const

u funkcijama članicama, 156

u parametrizaciji funkcije, 108

continue, 60

cout, 18, 173

datoteke, 171, 180

čitanje i upis, 182

režimi rada, 181

debager, 14

debugovanje, 14

default, 54

delete, 199

destruktori

definicija, 126

redosled pozivanja, 127, 160

virtuelni, 167

dinamička alokacija memorije, 83

operator delete, 85

operator new, 84

do-while, 59

doseg, 50, 96, 123

globalni, 98

double, 30

else, 50

exit(), 95

extern, 193

fill(), 176

flegovi za formatiranje, 175

float, 30

for, 56

fstream, 180

funkcije

argumenti, 92

close(), 182

deklaracija, 91

exit(), 95

flush(), 183

getline(), 183

gets(), 71

inline, 125

main(), 18, 110

nadjačavanje, 161

open(), 181

parametri, 91

podrazumevani argumenti, 111

povratak, 93

preklapanje, 110

prototip, 92, 109

rad sa stringovima, 72

rad sa znakovima, 72

rekurzivne, 112

setf(), 175

strcmp(), 76

unsetf(), 175

funkcije članice klase, 122

gets(), 71

- goto, 62
- grafički interfejs, 16
- greške
  - tokom izvršavanja, 14
  - tokom povezivanja, 14
  - tokom prevođenja, 13
- identifikatori, 23
- if, 50
- ifstream, 180
- imenici, 17
  - bezimeni, 101
  - deklaracija, 101
  - direktiva using, 101
- inspektorske funkcije članice, 155
- int, 28
- integrirana razvojna okruženja (IDE), 15
- interpretiranje, 15
- ios, 174
- islower(), 82
- isupper(), 82
- izlazni tok, 18
- izvedena klasa, 151
- izvorni kôd, 13
- izvršna datoteka, 14
- kapsuliranje, 11
- klase
  - apstraktne, 167
  - definicija, 120
  - deklaracija, 120
  - destruktori, 126
  - funkcije članice, 12, 120
  - inline funkcije, 125
  - ios, 174
  - izvođenje, 152
  - konstruktori, 125
  - kontrola pristupa članovima, 124, 153
  - nasleđivanje, 12
  - osnovna, 151
  - podaci članovi, 12, 120
  - specifikator private, 124
  - specifikator public, 124
  - string, 141
- ključne reči, 23, 24
- komentari, 17
- C stil, 17
- kompajler (prevodilac), 13
- kompajliranje programa, 13
- konstante, 28
- konstruktor kopije, 139
  - argumenti, 140
- konstruktori
  - definicija, 125
  - podrazumevani (default), 131
  - preklapanje, 129
  - preklopljeni, 158
  - redosled pozivanja, 127, 160
  - sa parametrima, 128
  - u izvedenim klasama, 156
- konverzija tipova, 39
  - pravila, 39
- konzolni programi, 16
- linker, 14
- literali, 28
- lokalne promenljive, 50
- long, 26, 28
- lvrednosti, 32
- manipulatori
  - programiranje, 179
- mutatorske funkcije članice, 155
- nadjačavanje funkcija, 161
- naredbe
  - break, 55, 60
  - continue, 60
  - do-while, 59
  - for, 56
  - goto, 62
  - if, 50
  - if-else-if lestvica, 52
  - return, 94
  - switch, 54
  - ugnježdeni if, 52
  - while, 58
- nasleđivanje, 151
  - javno izvođenje, 154
  - privatno izvođenje, 154
  - višestruko, 160
- new, 199
- niz stringova, 75

- nizovi, 67
  - deklaracija, 67
  - inicijalizacija, 73
  - inicijalizacija bez dimenzija, 74
  - jednodimenzionalni, 67
  - provera granica, 68
  - višedimenzionalni, 69
- nizovi znakova, 70
  - deklaracija, 70
- objekti, 120
  - dodela, 136
  - kopiranje, 136
  - prosleđivanje funkciji, 135
- objektni kôd, 13
- objektno orijentisano programiranje, 11, 119
- oblast važenja promenljivih, 96
- ofstream, 180
- operatori
  - ++, 32
  - , 32
  - , 137
  - ., 121
  - ::, 123
  - <<, 18, 37
  - >>, 37
  - ?:, 54
  - alternativne oznake, 24
  - aritmetički, 32
  - delete, 85
  - dodela, 31
  - logički, 34
  - modulo, 32
  - nabrajanje, 36
  - new, 84
  - prioritet, 37
  - relacioni, 33
  - sizeof, 35
  - skraćeni oblik, 33
  - na objekte izvedenih klasa, 161
  - NULL, 77
  - operator &, 78
  - operator \*, 78
  - osnovni tip, 79
  - this, 138
  - polimorfizam, 12, 110, 161, 163
  - polimorfna klasa, 163
  - povezivanje, 14
    - dinamičko, 166
    - statičko, 165
  - precision(), 176
  - preklapanje funkcija, 162
  - prenos po adresi, 105
  - prenos po vrednosti, 105
  - preprocesorske direktive, 17
    - define, 41, 189
    - error, 190
    - if, 191
    - ifdef, 191
    - ifndef, 191
    - include, 17, 190
    - undef, 192
  - proceduralno programiranje, 119
  - promenljive
    - definicija, 28
    - deklaracija, 27
    - inicijalizacija, 27
    - lokalne, 96
    - statičke, 100
    - statičke globalne, 100
  - protected
    - specifikator pristupa, 153
  - public, 120
  - reference
    - i konstruktor kopije, 139
    - i pokazivači, 107
    - kao parametri funkcija, 106
    - kao povratni tip funkcije, 107
    - samostalne, 107
  - rekurzija, 112
  - short, 26
  - signed, 26
  - simboličke konstante, 41
- pokazivači
  - aritmetika, 80
  - definicija, 76
  - i nizovi, 80
  - kao argumenti funkcija, 102
  - na objekte, 137

- sizeof, 35
- skrivanje podataka, 11
- specifikator izvođenja, 152
- specijalni znakovi, 25
- static
  - funkcije članice, 134
  - u podacima članovima, 131
- STL, 11
- strcat(), 72
- strcmp(), 72
- strcpy(), 72
- string, 196
  - klasa, 141
  - konstanta, 30
- stringovi
  - učitavanje preko tastature, 71
- strlen(), 72
- Stroustrup, Bjarne, 10
- switch, 54
- this, pokazivač, 138
- tipovi podataka, 26
  - brojevi u pokretnom zarezu, 30
  - celobrojni, 28
  - eksplicitna konverzija, 40
  - konverzija, 39
  - logički, 30
  - modifikatori, 26
  - opseg, 27
  - osnovni, 26
  - typedef, 35
  - znakovni, 29
- tokovi, 171
  - binarni, 172
  - klase, 173
  - predefinisani objekti, 172
  - tekstualni, 172
- tokovi podataka, 37
- tolower(), 72
- toupper(), 72
- typedef, 35
- U/I manipulatori, 177
- U/I sistem, 171
- ugnježđene petlje, 61
- unsigned, 26
- using, 101
- virtual, 163
- virtuelne funkcije, 163
  - čiste, 166
- void, povratni tip funkcije, 92
- while, 58
- width(), 176
- zaglavlja, 17, 109
  - cctype, 72
  - cfloating, 35
  - climits, 35
  - cmath, 57
  - cstdio, 71
  - cstdlib, 95
  - fstream, 180
  - iomanip, 177
  - iostream, 18, 171
  - znakovne konstante, 29

Na osnovu člana 23. stav 2. tačka 7. Zakona o porezu na dodatu vrednost („Službeni glasnik RS”, br. 84/04... i 61/07), Odlukom Senata Univerziteta Singidunum, Beograd, broj 260/07 od 8. juna 2007. godine, ova knjiga je odobrena kao osnovni udžbenik na Univerzitetu.

CIP - Каталогизација у публикацији  
Народна библиотека Србије, Београд

004.004.432.2C++(075.8)

004.42.045(075.8)

**БРАНОВИЋ, Ирина, 1975-**

Објектно оријентисано програмирање : C++  
/ Irina Branović. - 1. izd. - Beograd :  
#Univerzitet #Singidunum, 2011 (Loznica :  
Mladost Grup). - 212 str. : ilustr. ; 24 cm

Тираж 300. - Bibliografija: str. 207. -  
Registar.

ISBN 978-86-7912-373-2

а) Програмски језик “C++” б) Објектно  
оријентисано програмирање  
COBISS.SR-ID 186010380

© 2011.

Sva prava zadržana. Nijedan deo ove publikacije ne može biti reprodukovan u bilo kom vidu i putem bilo kog medija, u delovima ili celini bez prethodne pismene saglasnosti izdavača.