C++ PROGRAMIRANJE

Repetitorij s laboratorijskim vježbama

Drugo, urednički prerađeno izdanje

Tonći Kozina

Tonko Kovačević

Sveučilišni odjel za stručne studije Sveučilišta u Splitu

Split, 2026.

Sadržaj

Predgovor ovom izdanju **viii**

Terminološki pregled **ix**

1 Uvod u programiranje **1**
1.1 Brojevni sustavi . . . 2
1.1.1 Binarni brojevni sustav . . . 4
1.1.2 Oktalni brojevni sustav . . . 5
1.1.3 Heksadekadski brojevni sustav . . . 6
1.2 ASCII kod . . . 7
1.3 Programiranje u C++ . . . 8
1.4 Standardna zaglavlja . . . 10
1.4.1 Namespace `std` . . . 11
1.5 Struktura C++ programa . . . 12
1.5.1 Komentar . . . 12
1.5.2 Uključivanje zaglavlja . . . 12
1.5.3 Direktiva `using namespace std;` . . . 13
1.5.4 Funkcija `main` . . . 13
1.5.5 Ispis podataka . . . 13
1.5.6 Završetak programa . . . 13
1.5.7 Oblikovanje izvornoga koda . . . 14
1.5.8 Vrste komentara . . . 14

2 Tipovi podataka i operatori **17**
2.1 Identifikatori . . . 18
2.2 Tipovi podataka . . . 19
2.3 Deklaracije varijabli . . . 27
2.4 Inicijalizacija varijabli . . . 27
2.5 Područje važenja i trajanje objekata . . . 28
2.6 Literali i konstante . . . 30
2.6.1 Cjelobrojni literali . . . 30
2.6.2 Realni literali . . . 30
2.6.3 Znakovni i stringovni literali . . . 31
2.6.4 Imenovane konstante . . . 31
2.6.5 Makronaredbe . . . 32
2.7 Operatori . . . 32

2.7.1 Operator pridruživanja (=) . . . 32
2.7.2 Aritmetički operatori . . . 33
2.7.3 Skraćeni izrazi pridruživanja . . . 36
2.7.4 Operatori povećavanja i smanjenja . . . 36
2.7.5 Relacijski operatori . . . 36
2.7.6 Logički operatori . . . 37
2.7.7 Uvjetni operator . . . 38
2.7.8 Bit operatori . . . 39
2.7.9 Pretvorbe tipova . . . 41
2.7.10 Operator `sizeof` . . . 42
2.7.11 Manipulatori tokova . . . 42
2.7.12 Prioritet operatora . . . 42

3 Kontrola toka programa 44
3.1 Naredba `if-else` . . . 44
3.2 Naredba `switch` . . . 46
3.3 Programske petlje . . . 48
3.3.1 Petlja `while` . . . 48
3.3.2 Petlja `do-while` . . . 50
3.3.3 Petlja `for` . . . 51
3.4 Naredba `break` . . . 54
3.5 Naredba `continue` . . . 55
3.6 Naredba `goto` . . . 55

4 Funkcije 58
4.1 Definicija i poziv funkcije . . . 59
4.2 Funkcija `main` . . . 59
4.3 Prototipovi funkcija (*function prototypes*) . . . 60
4.4 Prijenos argumenta po vrijednosti . . . 62
4.4.1 Funkcije bez povratne vrijednosti . . . 64
4.5 Prijenos argumenta po referenci . . . 64
4.6 Zadane (*default*) vrijednosti parametara . . . 66
4.7 Preopterećivanje (*overloading*) funkcija . . . 67
4.8 Ostala svojstva funkcija . . . 68
4.8.1 Specifikator `inline` . . . 68
4.8.2 Rekurzivne funkcije . . . 68

5 Izvedeni oblici podataka 71
5.1 Polja podataka . . . 71
5.1.1 Inicijalizacija polja . . . 72
5.1.2 Pristupanje elementima polja . . . 73
5.1.3 Višedimenzijska polja . . . 74
5.1.4 Polja kao parametri funkcije . . . 78
5.1.5 Znakovni nizovi . . . 79
5.1.6 Inicijalizacija znakovnoga niza . . . 80

5.1.7 Pridruživanje vrijednosti C-znakovnom nizu . . . 80
5.1.8 Pretvorba C-znakovnoga niza u broj . . . 83

6 Klasa `string` 86
6.1 Deklaracija i inicijalizacija . . . 86
6.2 Unos i ispis . . . 87
6.3 Osnovne operacije . . . 87
6.3.1 Pridruživanje i spajanje . . . 87
6.3.2 Uspoređivanje . . . 88
6.4 Pristup pojedinim znakovima . . . 88
6.5 Najčešće korišteni članovi klase . . . 89
6.6 Pregled ostalih članova klase . . . 91
6.7 Prijenos stringa funkciji . . . 91

7 Pokazivači 94
7.1 Adresni operator (&) . . . 96
7.2 Operator dereferenciranja (*) . . . 97
7.3 Deklaracija pokazivača . . . 98
7.4 Pokazivači i polja . . . 100
7.5 Pokazivači i znakovna polja . . . 102
7.6 Aritmetika pokazivača . . . 102
7.7 Pokazivači na pokazivače . . . 103
7.8 Pokazivači tipa `void*` . . . 104

8 Reference 106
8.1 Deklaracija i inicijalizacija reference . . . 106
8.2 Reference kao parametri funkcija . . . 107
8.2.1 Konstantne reference . . . 108
8.2.2 Funkcije koje vraćaju reference . . . 108

9 Strukture 112
9.1 Strukture podataka . . . 112
9.2 Pokazivači na strukturu . . . 115
9.3 Ugniježđene strukture . . . 117
9.4 Korisnički definirana imena tipova . . . 118
9.5 Unije . . . 118
9.6 Anonimne unije . . . 119
9.7 Pobrojeni tipovi podataka . . . 120

10 Organizacija i dinamičko upravljanje memorijom 122
10.1 Objekt, memorija i životni vijek . . . 122
10.2 Četiri vrste trajanja pohrane . . . 123
10.3 Tipična područja memorije programa . . . 124
10.4 Što se gdje smješta? . . . 125
10.5 Stack i pozivi funkcija . . . 126

10.5.1 Rekurzija i stack overflow . . . 127
10.6 Free store, heap i izraz new . . . 128
10.6.1 Inicijalizacija dinamičkoga objekta . . . 129
10.6.2 Neuspjela dodjela . . . 129
10.7 Delete-izrazi `delete` i `delete[]` . . . 129
10.8 Ugrađena i dinamička polja . . . 130
10.9 Objekti klasa u memoriji . . . 132
10.9.1 Veličina, poravnanje i padding . . . 134
10.10 string i vector: objekt nije isto što i njegovi elementi . . . 134
10.11 Vlasništvo nad dinamičkim objektom . . . 136
10.12 Najčešće pogreške pri radu s memorijom . . . 136
10.12.1 Curenje memorije . . . 136
10.12.2 Pristup nakon oslobađanja . . . 136
10.12.3 Dvostruko oslobađanje . . . 137
10.12.4 Pogrešan oblik operatora delete . . . 137
10.12.5 Pristup izvan granica polja . . . 137
10.12.6 Neinicijalizirani i nevaljani pokazivači . . . 137
10.13 Kako otkrivati memorijske pogreške . . . 138
10.14 Dinamička memorija u jeziku C . . . 138
10.15 Od ručnoga upravljanja prema RAII-ju . . . 139

11 Klase i namespace 142
11.1 Definicija funkcijskih članova . . . 144
11.2 Konstruktori i destruktori . . . 146
11.2.1 Preopterećenje konstruktora . . . 148
11.3 Pokazivači na klase . . . 150
11.3.1 Klase definirane ključnom riječju `struct` . . . 151
11.4 Preopterećenje operatora . . . 152
11.4.1 Preopterećenje operatora `+` . . . 152
11.4.2 Preopterećenje operatora `++` i `--` . . . 154
11.4.3 Preopterećenje operatora pridruživanja `=` . . . 155
11.5 Ključna riječ `this` . . . 156
11.6 Statički članovi . . . 157
11.7 Prijateljska funkcija . . . 158
11.7.1 Prijateljska klasa . . . 159
11.8 Nasljeđivanje između klasa . . . 160
11.8.1 Višestruko nasljeđivanje . . . 164
11.9 Pokazivači na osnovnu klasu . . . 165
11.10 Virtualni funkcijski članovi . . . 166
11.11 Apstraktne klase . . . 168
11.12 Scope resolution operator `::` . . . 170
11.12.1 Pristup imenu iz globalnoga područja . . . 171
11.12.2 Određivanje pripadnosti klasi . . . 171
11.12.3 Pristup članu osnovne klase . . . 171
11.12.4 Pristup članu namespacea . . . 172

11.13 Kvalifikator `volatile` . . . 172
11.13.1 Funkcijski članovi kvalificirani s `volatile` . . . 173
11.14 Namespace . . . 174
11.14.1 Ključna riječ `using` . . . 175
11.14.2 Ugniježđeni namespacei i zamjenska imena . . . 176

12 Predlošci (Templates) 178
12.1 Zašto su nam potrebni templates? . . . 178
12.2 Funkcijski template . . . 179
12.3 Kako prevoditelj određuje tip? . . . 180
12.3.1 Argumenti različitih tipova . . . 181
12.4 Izričito zadavanje tipa . . . 181
12.5 Više template parametara . . . 182
12.5.1 `typename` i `class` . . . 182
12.6 Template ne radi nužno sa svakim tipom . . . 183
12.7 Klasni template . . . 184
12.7.1 Određivanje template argumenata klase u C++17 . . . 185
12.8 Funkcijski članovi klasnoga templatea . . . 185
12.9 Klasni template s više tipova . . . 186
12.10 Veza između templatesa i STL-a . . . 188
12.11 Gdje se piše definicija templatea? . . . 188
12.12 Najčešće pogreške . . . 190
12.13 Kratki sažetak . . . 190

13 Rad s datotekama 192
13.1 Otvaranje i zatvaranje datoteke . . . 192
13.2 Zapisivanje u datoteku . . . 193
13.3 Čitanje iz datoteke . . . 194
13.4 Stanja toka i oporavak nakon pogreške . . . 195
13.5 Pozicioniranje u datoteci . . . 196
13.6 Binarne datoteke i funkcije `read()` i `write()` . . . 197

14 Osnove STL-a: `vector` i algoritmi 200
14.1 Zašto nam je potreban `vector`? . . . 200
14.2 Dodavanje i uklanjanje elemenata . . . 201
14.3 Pristup elementima . . . 202
14.4 Veličina i stanje spremnika . . . 203
14.4.1 Veličina, kapacitet i rezerviranje pohrane . . . 203
14.5 Prolazak kroz elemente . . . 203
14.6 Iteratori . . . 204
14.7 Osnovni STL algoritmi . . . 205
14.7.1 Sortiranje . . . 205
14.7.2 Traženje . . . 205
14.7.3 Brojanje . . . 205
14.7.4 Najveći i najmanji element . . . 206

14.7.5 Obrtanje redoslijeda . . . 206
14.8 Lambda funkcije i STL . . . 206
14.9 STL u jednoj stranici . . . 207

15 Lambda funkcije 210
15.1 Sintaksa lambda funkcije . . . 210
15.2 Prva lambda funkcija . . . 211
15.3 Lambda funkcije s parametrima . . . 211
15.4 Povratna vrijednost . . . 212
15.5 Generičke lambda funkcije . . . 212
15.6 Hvatanje varijabli iz okoline . . . 212
15.6.1 Hvatanje vrijednošću . . . 213
15.6.2 Hvatanje referencom . . . 214
15.7 Lambda funkcije u standardnim algoritmima . . . 214
15.7.1 Sortiranje . . . 214
15.7.2 Obrada svakoga elementa . . . 215
15.7.3 Brojanje elemenata koji zadovoljavaju uvjet . . . 215
15.7.4 Pretvorba elemenata . . . 215
15.8 Usporedba obične i lambda funkcije . . . 216

16 Pametni pokazivači 219
16.1 Zašto su potrebni pametni pokazivači? . . . 219
16.2 Načelo RAII . . . 220
16.3 Jedinstveno vlasništvo: `std::unique_ptr` . . . 220
16.3.1 Prijenos vlasništva funkcijom `std::move()` . . . 221
16.3.2 Dinamička polja . . . 222
16.4 Zajedničko vlasništvo: `std::shared_ptr` . . . 223
16.5 Promatranje bez vlasništva: `std::weak_ptr` . . . 224
16.6 Cikličko vlasništvo . . . 225
16.7 Vlasništvo i pristup nisu isto . . . 226
16.8 Koji pokazivač odabrati? . . . 227
16.9 Česte operacije . . . 227

A Zadatci za laboratorijske vježbe 230
A.1 Vježba 1 . . . 230
A.2 Vježba 2 . . . 233
A.3 Vježba 3 . . . 239
A.4 Vježba 4 . . . 243
A.5 Vježba 5 . . . 246
A.6 Vježba 6 . . . 248
A.7 Vježba 7 . . . 251
A.8 Vježba 8 . . . 256
A.9 Vježba 9 . . . 258
A.10 Vježba 10 . . . 260

B Priprema za kolokvij/ispit **262**
B.1 Brojevni sustavi, ASCII i osnove C++ programa . . . 262
B.2 Tipovi podataka i operatori . . . 263
B.3 Kontrola toka programa . . . 266
B.4 Funkcije . . . 268
B.5 Polja i C-znakovni nizovi . . . 271
B.6 Klasa `string` . . . 273
B.7 Pokazivači . . . 274
B.8 Reference . . . 276
B.9 Strukture, unije i pobrojeni tipovi . . . 278
B.10 Dinamička memorija . . . 279
B.11 Klase . . . 282
B.12 Namespace i scope resolution operator . . . 284
B.13 Templates . . . 285
B.14 STL: `vector`, iteratori i algoritmi . . . 286
B.15 Lambda funkcije . . . 288
B.16 Pametni pokazivači . . . 289
B.17 Rad s datotekama . . . 291

Literatura i normativni izvori **294**

Kazalo pojmova **295**

Predgovor ovom izdanju

Prethodno izdanje ove skripte pod naslovom Programiranje - Repetitorij s laboratorijskim vježbama iz 2010, autora Tonka Kovačevića nastalo je s namjerom da studentima olakša usvajanje temeljnih pojmova programiranja i posluži kao potpora nastavi iz predmeta Programiranje. Tijekom godina pokazala se potreba za njegovom temeljitom sadržajnom, stručnom, jezičnom i grafičkom doradom. Ovo izdanje stoga nije samo dopunjena inačica prethodnoga teksta, nego rezultat njegove cjelovite revizije uz dozvolu autora Tonka Kovačevića.

Programski su primjeri ponovno pregledani, ispravljene su uočene pogreške, a zastarjela ili danas neprikladna rješenja zamijenjena suvremenijima i usklađena sa standardom C++17. Uz temeljne jezične konstrukcije obuhvaćeni su `string`, reference, organizacija i upravljanje memorijom, *namespace*, *templates*, STL, lambda funkcije i pametni pokazivači. Ujednačeni su terminologija i prikaz programskoga koda, a novi grafički prikazi dodani su ondje gdje olakšavaju razumijevanje apstraktnih pojmova. Za stručne nazive koji se uobičajeno susreću u standardu, dokumentaciji i razvojnim alatima dodan je zaseban Terminološki pregled.

Skripta je ponajprije namijenjena studentima studijskih programa Elektronika i Elektroenergetika Sveučilišnoga odjela za stručne studije Sveučilišta u Splitu, ali se njome mogu koristiti i svi koji žele postupno usvojiti osnove programiranja u jeziku C++. Od čitatelja se ne očekuje prethodno programersko znanje. Gradivo se razvija od osnovnih pojmova o računalima, brojevnim sustavima, algoritmima i strukturi programa prema složenijim načinima organiziranja podataka i upravljanja resursima.

Cilj skripte nije samo upoznavanje sintakse jezika C++. Programiranje zahtijeva analiziranje problema, prepoznavanje njegovih bitnih elemenata, oblikovanje postupka rješavanja te pretvaranje toga postupka u ispravan i razumljiv program. Zbog toga tekst povezuje objašnjenja, algoritme, programske primjere i grafičke prikaze. Primjeri su namjerno usmjereni na pojam koji se u određenome trenutku obrađuje; provjera unosa, obrada pogrešaka i robusnija rješenja uvode se postupno, kada postanu predmet učenja.

Ciljevi učenja, napomene, upozorenja na česte pogreške i pitanja za samoprovjeru trebaju olakšati samostalan rad i povezivanje gradiva. Priprema za kolokvij i ispit obuhvaća sve cjeline, a zadaci za vježbu ostaju uz pripadna poglavlja. Laboratorijski je dio namjerno ograničen na deset vježbi, u skladu s prostornim i vremenskim okvirom kolegija. Njihov broj stoga nije zamišljen kao mjera opsega rukopisa, nego kao izvediv nastavni izbor.

Namjera je ovoga izdanja pružiti jasnu i pouzdanu osnovu za učenje, ali i potaknuti čitatelja na samostalno ispitivanje rješenja, uspoređivanje različitih pristupa i postupno razvijanje vlastitoga programerskog načina razmišljanja.

U uredničkoj i tehničkoj pripremi drugoga izdanja autori su surađivali s ChatGPT-om i Codexom (OpenAI), koji su korišteni kao alati za uređivanje LaTeX izvora, stručnu i terminološku provjeru, provjeru programskih primjera, izradu i obradu grafičkih priloga te pripremu pristupačne PDF inačice. Sve prijedloge i izmjene pregledali su i odobrili autori, koji snose odgovornost za konačni sadržaj.

Terminološki pregled

U skripti se za dio pojmova zadržavaju engleski stručni nazivi jer se u tome obliku pojavljuju u standardu jezika, dokumentaciji, razvojnim alatima i porukama prevoditelja. Pri prvome se pojavljivanju pojam objašnjava na hrvatskome, a poslije se dosljedno rabi oblik naveden u prvome stupcu. Nazivi elemenata standardne biblioteke pišu se kao programski identifikatori.

Naziv u skripti	Značenje	Napomena o uporabi
argument i parametar	Argument je vrijednost ili izraz naveden u pozivu funkcije, a parametar je ime navedeno u deklaraciji ili definiciji funkcije	Pri pozivu `zbroji(5, 3)` brojevi `5` i `3` argumenti su, a imena kojima ih funkcija prima njezini su parametri. Argument se može prenositi po vrijednosti ili po referenci.
deklaracija i definicija	Deklaracija uvodi ime i opisuje svojstva entiteta, dok definicija daje njegovo potpuno određenje, primjerice tijelo funkcije ili pohranu objekta	Svaka je definicija ujedno deklaracija, ali deklaracija ne mora biti definicija. Prototip funkcije jest deklaracija; njezino tijelo čini definiciju.
inicijalizacija i pridruživanje	Inicijalizacija određuje početno stanje objekta pri njegovu stvaranju, a pridruživanje mijenja vrijednost već postojećega objekta	U zapisu `int x{5};` objekt se inicijalizira. U naknadnom zapisu `x = 7;` nova mu se vrijednost pridružuje.
podrazumijevani i implicitno deklarirani konstruktor	Podrazumijevani je konstruktor svaki konstruktor koji se može pozvati bez argumenata. Implicitno deklarirani podrazumijevani konstruktor onaj je koji pod određenim uvjetima deklarira prevoditelj	Ako klasa deklarira bilo koji konstruktor, prevoditelj više ne deklarira implicitni podrazumijevani konstruktor. Potreban se podrazumijevani konstruktor može izričito deklarirati, primjerice `C() = default;`.
stack	memorijsko područje koje se uobičajeno rabi za pozive funkcija i objekte automatskoga trajanja	Hrvatski naziv je *stog*, ali se u skripti nakon prvoga objašnjenja rabi *stack*. C++ standard ne zahtijeva takvu fizičku organizaciju memorije.
heap	memorijsko područje iz kojega se uobičajeno pribavlja dinamička pohrana	Ne treba ga bez ograde poistovjetiti s pojmom *free store*.
free store	pohrana kojom u C++-u upravljaju izrazi `new` i `delete`	Pojam pripada jeziku i standardnoj terminologiji; uobičajeno se ostvaruje na *heapu*.

Nastavlja se na sljedećoj stranici.

Naziv u skripti	Značenje	Napomena o uporabi
stack frame	dio *stacka* povezan s jednim pozivom funkcije; može sadržavati parametre, lokalne objekte i podatke za povratak	U cijelome se tekstu dosljedno rabi naziv *stack frame*.
stack overflow	iscrpljivanje raspoloživa prostora *stacka*	Može nastati zbog preduboke rekurzije ili vrlo velikih lokalnih objekata.
namespace	mehanizam za organiziranje imena i sprječavanje njihovih sukoba	Hrvatski naziv je *imenski prostor*; u tekstu se zadržava *namespace*.
scope resolution operator	operator `::`, kojim se određuje područje kojem ime pripada	Zadržava se puni engleski naziv; skraćeni oblik *scope operator* rabi se nakon njegova uvođenja.
template	obrazac iz kojega prevoditelj stvara funkcije ili klase za konkretne tipove	Hrvatski naziv je *predložak*; u poglavlju o generičkom programiranju zadržava se *template*.
`std::string`, kraće `string`	klasa standardne biblioteke za rad s nizovima znakova	Budući da je riječ o nazivu tipa, piše se kao programski identifikator i ne prevodi se.
`std::vector`, kraće `vector`	sekvencijski spremnik standardne biblioteke čija se veličina može mijenjati	Ne naziva se *vektorom* u matematičkome smislu; piše se kao programski identifikator.
iterator	objekt kojim se prolazi kroz elemente spremnika	Zadržava se međunarodno prihvaćen naziv *iterator*.
lambda funkcija	neimenovani funkcijski objekt definiran na mjestu uporabe	Zadržava se naziv *lambda*; ne zamjenjuje se izrazom *anonimna funkcija* jer lambda u C++-u stvara objekt jedinstvenoga tipa.
RAII	načelo prema kojem objekt pribavlja resurs pri stvaranju i oslobađa ga pri uništavanju	Kratica dolazi od *Resource Acquisition Is Initialization* i ne prevodi se.
new-expression, u tekstu *new-izraz*	izraz koji pribavlja pohranu i u njoj stvara objekt	Ne treba ga poistovjetiti s alokacijskom funkcijom `operator new` koju može pozvati.
delete-expression, u tekstu *delete-izraz*	izraz koji uništava objekt i oslobađa pripadajuću pohranu	Razlikuje se od dealokacijske funkcije `operator delete`.

Napomena

Engleski naziv ne mijenja pravila hrvatske rečenice. Kada je potrebno, dodatak se piše izvan istaknutoga pojma: na *stacku*, s *heapa*, u *namespaceu*, funkcijskoga *templatea*. Programske ključne riječi i identifikatori, primjerice `namespace`, `template` i `std::vector`, uvijek se pišu u izvornome obliku.

1. Uvod u programiranje

Što ćete naučiti?

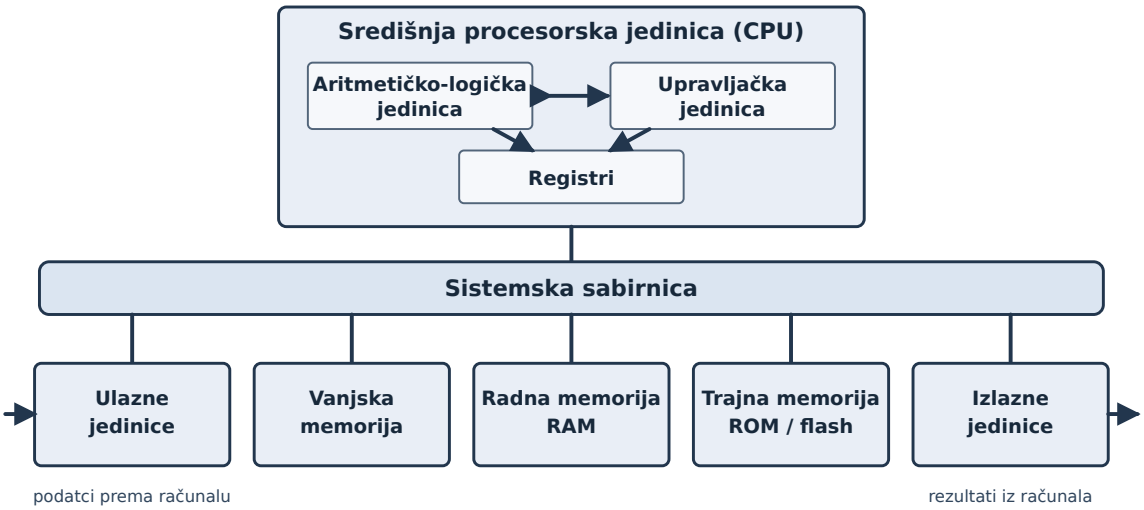
Nakon ovoga poglavlja moći ćete opisati osnovne sastavnice računalnoga sustava, objasniti prikaz podataka u binarnome, oktalnome, dekadskome i heksadekadskome brojevnom sustavu te razlikovati ASCII, Unicode i UTF-8. Također ćete moći opisati postupak nastanka izvršnoga programa, razlikovati osnovne vrste programskih pogrešaka i objasniti strukturu jednostavnoga programa napisanoga u jeziku C++.

Računalo (*computer*) jest uređaj koji prema zadanim programima obrađuje podatke. Računalni sustav čine *sklopovlje* ili *hardver* (*hardware*) te *programska podrška* ili *softver* (*software*) s pripadajućom dokumentacijom. Program je uređeni skup naredbi koje računalo treba izvršiti, a zapisuje se formalnim jezikom koji se naziva *programski jezik* (*programming language*).

Prema načinu prikaza i obrade podataka računala se mogu podijeliti na analogna i digitalna. Analogna računala obrađuju kontinuirane fizikalne veličine, dok digitalna računala obrađuju podatke prikazane diskretnim vrijednostima. U digitalnim se računalima informacije najčešće prikazuju dvama stanjima, koja se označuju znamenkama 0 i 1. Programeru i korisniku nisu važna sama elektronička stanja, nego informacije koje su njima prikazane.

Brojevni sustav sa znamenkama 0 i 1 naziva se *binarni brojevni sustav*. Svaka binarna znamenka naziva se *bit* (*binary digit*). Nizovima bitova mogu se prikazati programske naredbe, brojevi, znakovi, slike, zvukovi i druge vrste podataka.

Osnovne sastavnice digitalnoga računala prikazane su na slici 1.1.



Slika 1.1: Pojednostavnjen prikaz sastavnica digitalnoga računala povezanih sistemskom sabirnicom

Središnja procesorska jedinica – CPU (*Central Processing Unit*) izvršava programske naredbe te upravlja obradom podataka. U osobnim računalima CPU je najčešće izveden kao mikroprocesor, odnosno

integrirani elektronički sklop na jednome čipu ili u kućištu koje može sadržavati više jezgri i čipleta. Naziv *procesor* uobičajeno se rabi za CPU. Program koji se upravo izvodi, zajedno s pripadajućim stanjem i dodijeljenim resursima, naziva se *proces*.

Radna memorija ili RAM (*Random Access Memory*) privremeno pohranjuje programske naredbe i podatke potrebne tijekom rada računala. S programerskoga gledišta memorija se može promatrati kao uređeni niz adresabilnih memorijskih jedinica. Najmanja adresabilna jedinica naziva se *bajt*; u suvremenim osobnim računalima bajt u pravilu sadržava osam bitova. Svaka memorijska lokacija ima adresu koja određuje njezin položaj u memoriji.

U procesoru se nalaze brze memorijske jedinice koje se nazivaju *registri*. One privremeno čuvaju podatke, adrese, upravljačke informacije i rezultate operacija. Procesori sadržavaju registre različitih namjena i veličina. Pojam *riječ* (*word*) označuje prirodnu podatkovnu jedinicu određene računalne arhitekture. Oznake 32-bitna i 64-bitna arhitektura odnose se na ključna svojstva procesorske arhitekture, primjerice širinu općih registara i adresnoga prostora, a ne na veličinu svakoga registra.

Trajno pohranjivanje podataka omogućuju uređaji poput tvrdih i poluvodičkih diskova, memorijskih kartica i optičkih medija. *Ulazne jedinice*, primjerice tipkovnica, miš, mikrofon i skener, služe za unos podataka. *Izlazne jedinice*, primjerice monitor, pisač i zvučnici, služe za prikaz ili reprodukciju rezultata obrade.

Operacijski sustav – OS (*Operating System*) skup je programa koji upravlja sklopovljem i računalnim resursima te drugim programima pruža usluge potrebne za izvođenje. Upravlja procesima, radnom memorijom, datotekama, ulaznim i izlaznim uređajima te korisničkim računima i zaštitom sustava. Pri pokretanju računala firmware, primjerice UEFI, inicijalizira sklopovlje i započinje učitavanje operacijskoga sustava s uređaja za pohranu u radnu memoriju. Firmware se nalazi u trajnoj flash-memoriji računala; nije dio operacijskoga sustava.

Programska podrška uobičajeno se dijeli na sistemsku i aplikacijsku. Sistemskoj programskoj podršci pripadaju operacijski sustav, upravljački programi, uslužni programi te razvojni alati kao što su prevoditelj, povezivač i program za otkrivanje pogrešaka. Aplikacijsku programsku podršku čine programi namijenjeni izvršavanju korisničkih zadataka, primjerice uređivači teksta, statistički i matematički programi te programi za rad s bazama podataka. Programi koji se izrađuju u sklopu ovoga kolegija pripadaju aplikacijskoj programskoj podršci.

1.1 Brojevni sustavi

Brojevni sustav određuje skup znamenaka i pravila prema kojima se njima zapisuju brojevi. U pozicijskome brojevnom sustavu doprinos pojedine znamenke ukupnoj vrijednosti broja ovisi o njezinoj vrijednosti i položaju. Rimski je zapis pretežno nepozicijski, aditivno-suptraktivni sustav: znakovi poput `I`, `V` i `X` imaju unaprijed određene vrijednosti, a vrijednost se zapisa određuje pravilima njihova slaganja.

Rimski broj	Arapski broj
I	1
II	2
III	3
IV	4
V	5

Znak `I` ima vrijednost jedan gdje god se nalazi. U dekadskome zapisu znamenka 1 također ima vrijednost jedan, ali njezin doprinos ukupnoj vrijednosti ovisi o položaju:

Arapski broj	Rimski broj
1	I
10	X
100	C

U zapisima 1, 10 i 100 znamenka 1 nalazi se na mjestu jedinica, desetica odnosno stotica. Njezini su doprinosi zato $1 \cdot 10^0$, $1 \cdot 10^1$ i $1 \cdot 10^2$.

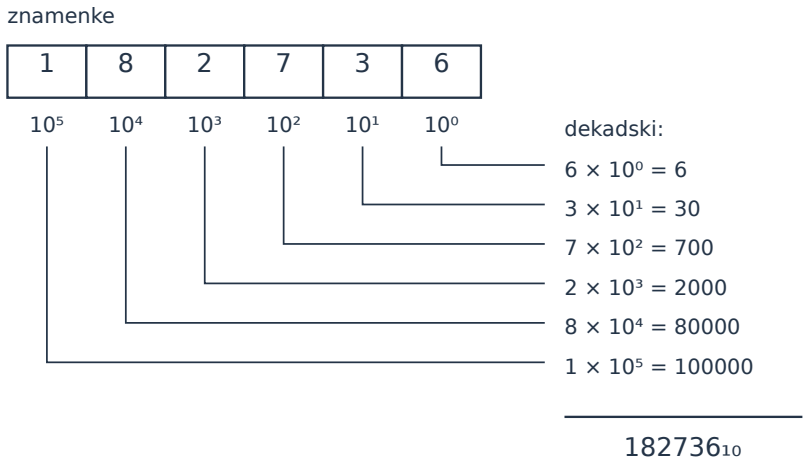
Primjerice, vrijednost broja 275 nije $2 + 7 + 5$, nego $200 + 70 + 5$:

$$\begin{array}{r} 200 \\ +70 \\ +5 \\ \hline 275 \end{array}$$

Prema tome,

$$275 = 2 \cdot 10^2 + 7 \cdot 10^1 + 5 \cdot 10^0.$$

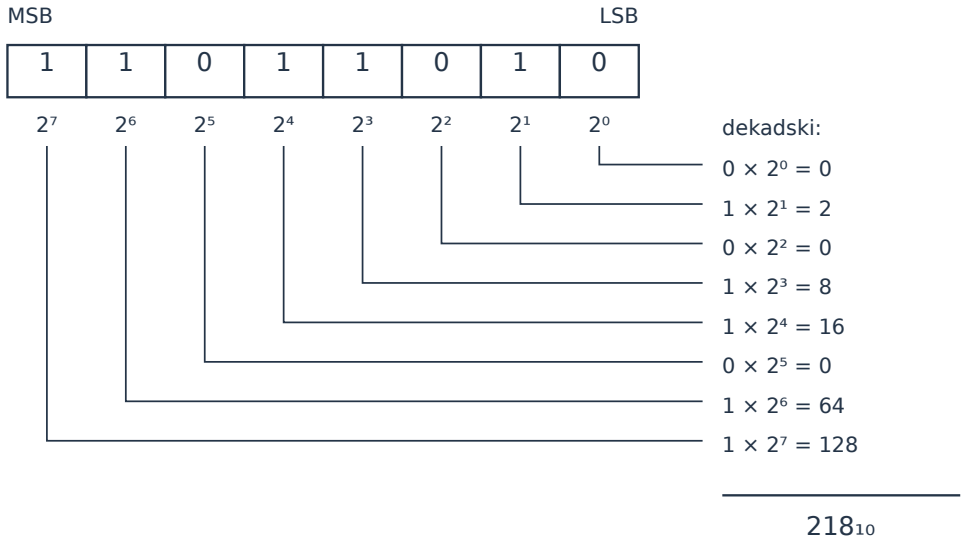
Općenito, u pozicijskome sustavu s bazom b položaji znamenaka zdesna nalijevo imaju težine $b^0, b^1, b^2, \ldots$. Dekadski sustav ima bazu 10 i znamenke od 0 do 9. Mjesne vrijednosti u dekadskome zapisu prikazane su na slici 1.2.



Slika 1.2: Dekadski brojevni sustav

1.1.1 Binarni brojevni sustav

Kada je baza sustava 2, dopuštene su samo znamenke 0 i 1. Vrijednost binarnoga broja 11011010_2 dobiva se zbrajanjem umnožaka znamenaka i pripadajućih potencija baze 2, kao što je prikazano na slici 1.3.



Slika 1.3: Pretvorba binarnoga broja u dekadski

Rezultat pretvorbe jest 218_{10}. U binarnome zapisu LSB (*Least Significant Bit*) označuje bit najmanje mjesne vrijednosti, a MSB (*Most Significant Bit*) bit najveće mjesne vrijednosti.

Pretvaranje dekadskoga cijelog broja u binarni zapis može se provesti uzastopnim dijeljenjem s 2 i

bilježenjem ostataka. Za broj 57 postupak izgleda ovako:

$$
\begin{aligned}
57 : 2 &= 28, \text{ ostatak } 1 \\
28 : 2 &= 14, \text{ ostatak } 0 \\
14 : 2 &= 7, \text{ ostatak } 0 \\
7 : 2 &= 3, \text{ ostatak } 1 \\
3 : 2 &= 1, \text{ ostatak } 1 \\
1 : 2 &= 0, \text{ ostatak } 1
\end{aligned}
$$

Ostatci se čitaju odozdo prema gore pa vrijedi $57_{10} = 111001_2$.

Od standarda C++14 binarni cjelobrojni literal može se zapisati prefiksom `0b` ili `0B`, primjerice `0b11011010`.

1.1.2 Oktalni brojevni sustav

Oktalni brojevni sustav ima bazu 8 i znamenke od 0 do 7. U matematičkom zapisu baza se može označiti indeksom, primjerice $21_8 = 17_{10}$. U izvornome kodu jezika C++ oktalni cjelobrojni literal započinje znamenkom `0`, pa se ista vrijednost zapisuje kao `021`.

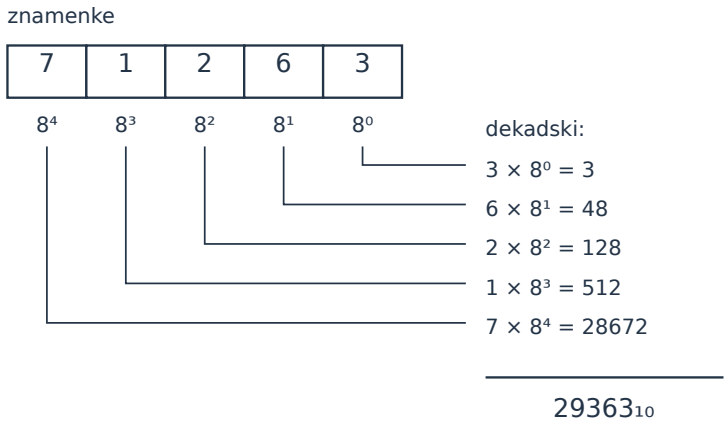
Napomena

Početna nula u zapisu cjelobrojnoga literala nije bez značenja: u jeziku C++ označuje oktalni literal. Zato `021` ima dekadsku vrijednost 17, a ne 21. Literal koji sadržava znamenku 8 ili 9 ne može biti oktalan pa će, primjerice, zapis `089` uzrokovati pogrešku pri prevođenju.

Tablica 1.1: Oktalni cjelobrojni literali i njihove dekadske vrijednosti

Literal	Vrijednost	Literal	Vrijednost
`0`	0	`011`	9
`01`	1	`012`	10
`02`	2	`013`	11
`03`	3	`014`	12
`04`	4	`015`	13
`05`	5	`016`	14
`06`	6	`017`	15
`07`	7	`020`	16
`010`	8	`021`	17

Pretvorba matematički zapisanoga oktalnog broja 71263_8 u dekadski prikazana je na slici 1.4.



Slika 1.4: Pretvorba oktalnoga broja u dekadski

Iz prikazanoga slijedi $71263_8 = 29363_{10}$. Kao C++ literal isti bi se oktalni broj zapisao `071263`.

Budući da je $8 = 2^3$, binarni se broj može brzo pretvoriti u oktalni grupiranjem bitova zdesna nalijevo u skupine po tri:

$$\begin{array}{ccc} 11 & 011 & 010 \\ 3 & 3 & 2 \end{array}$$

Stoga je $11011010_2 = 332_8$. Obrnutim se postupkom svaka oktalna znamenka zamjenjuje skupinom od triju bitova.

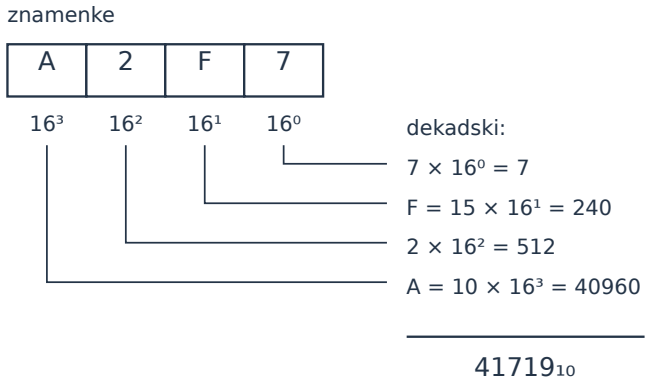
1.1.3 Heksadekadski brojevni sustav

Heksadekadski brojevni sustav ima bazu 16. Rabi znamenke od 0 do 9 i slova A, B, C, D, E i F za vrijednosti od 10 do 15. U matematičkom zapisu baza se označuje indeksom, primjerice DA_{16}. U C++-u heksadekadski cjelobrojni literal započinje prefiksom `0x` ili `0X`, primjerice `0xDA`.

Tablica 1.2: Heksadekadske znamenke i njihove dekadske vrijednosti

Znamenka	Vrijednost	Znamenka	Vrijednost
0	0	8	8
1	1	9	9
2	2	A	10
3	3	B	11
4	4	C	12
5	5	D	13
6	6	E	14
7	7	F	15

Na slici 1.5 prikazana je pretvorba heksadekadskoga broja u dekadski.



Slika 1.5: Pretvorba heksadekadskoga broja u dekadski

Budući da je $16 = 2^4$, binarni se broj pretvara u heksadekadski grupiranjem bitova zdesna nalijevo u skupine po četiri:

$$\begin{array}{cc} 1101 & 1010 \\ \mathrm{D} & \mathrm{A} \end{array}$$

Stoga isti broj ima matematičke zapise

$$11011010_2 = 218_{10} = 332_8 = \mathrm{DA}_{16}.$$

Odgovarajući C++ cjelobrojni literali jesu:

```
218          // dekadski literal
0332         // oktalni literal
0xDA         // heksadekadski literal
0b11011010   // binarni literal (C++14 i noviji)
```

Jedna heksadekadska znamenka odgovara skupini od četiri bita. Takva se skupina naziva *polubajt* (*nibble*). Dvije heksadekadske znamenke mogu prikazati sve vrijednosti jednoga osmobitnog bajta, od `0x00` do `0xFF`, pa se heksadekadski zapis često rabi u računalstvu.

1.2 ASCII kod

Računalo sve podatke pohranjuje u binarnome obliku. Da bi se slovima, znamenkama i drugim znakovima mogli pridružiti brojčani kodovi, razvijen je ASCII (*American Standard Code for Information Interchange*).

Standardni ASCII koristi 7 bitova i sadržava 128 znakova, čiji su kodovi u rasponu od 0 do 127. Obuhvaća kontrolne znakove, razmak, znamenke, velika i mala slova engleske abecede, interpunkcijske znakove i druge simbole.

Tablica 1.3: Standardni ASCII kod

	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	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	razmak	!	”	#	$	%	&	’	(	)	*	+	,	-	.	/
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

Tablica je organizirana prema heksadekadskim kodovima: oznaka retka predstavlja prvu, a oznaka stupca drugu heksadekadsku znamenku. Primjerice, veliko slovo `A` nalazi se u retku 4 i stupcu 1 pa je njegov ASCII kod `0x41`, odnosno 65 u dekadskome i `1000001` u binarnome sustavu.

ASCII obuhvaća samo znakove engleske abecede. Unicode je standard koji znakovima različitih pisama pridružuje jedinstvene kodne točke, dok je UTF-8 jedan od načina kodiranja tih kodnih točaka u bajtove. Za hrvatska slova i znakove drugih pisama danas se najčešće rabi Unicode kodiran kao UTF-8.

1.3 Programiranje u C++

Razvoj programskoga proizvoda (*software product*) uobičajeno obuhvaća sljedeće faze:

1. Analiza i definiranje zahtjeva (*requirements analysis and specification*);
2. Oblikovanje sustava (*software design*);
3. Implementacija (*implementation*);
4. Testiranje (*testing*);
5. Isporuka i uvođenje u uporabu (*deployment*);
6. Održavanje (*maintenance*).

Navedene faze predstavljaju osnovne aktivnosti životnoga ciklusa razvoja programske podrške (*software development life cycle, SDLC*). Ovisno o odabranom modelu razvoja, one se mogu izvoditi slijedno ili se tijekom razvoja mogu preklapati i ponavljati. Njihovim se planskim provođenjem povećava kvaliteta programskoga proizvoda i olakšava njegovo kasnije održavanje.

U fazi analize i definiranja zahtjeva utvrđuje se što programski proizvod treba omogućiti korisniku. Zahtjevi se određuju u suradnji s korisnicima, naručiteljem i članovima razvojnoga tima. Pritom se opisuju funkcionalnosti sustava, ograničenja te drugi funkcionalni i nefunkcionalni zahtjevi koje proizvod treba ispuniti.

U fazi oblikovanja određuje se način na koji će zahtjevi biti ostvareni. Definira se arhitektura sustava, njegove sastavnice, način obrade i pohrane podataka, korisnička sučelja te veze među pojedinim dijelovima sustava. U ovoj se fazi mogu izrađivati funkcijski i blok-dijagrami, modeli podataka, algoritmi i druga tehnička dokumentacija potrebna za implementaciju.

Implementacija obuhvaća izradu programskoga proizvoda pisanjem izvornoga koda u odabranom programskom jeziku, primjerice jeziku C++. Programski se kod izrađuje na temelju prethodno definiranih zahtjeva i rješenja utvrđenih u fazi oblikovanja.

U fazi testiranja provjerava se radi li programski proizvod u skladu s postavljenim zahtjevima. Testiraju se pojedine sastavnice i sustav kao cjelina, otkrivaju se programske pogreške te se, kada je to potrebno, provjerava kompatibilnost s prethodnim verzijama.

Isporuka i uvođenje u uporabu obuhvaćaju pripremu konačne verzije programskoga proizvoda, njegovu instalaciju ili objavljivanje, konfiguriranje sustava te predaju potrebne korisničke dokumentacije. Ovisno o vrsti proizvoda, programska se podrška može distribuirati putem mreže, repozitorija, trgovine aplikacija ili drugoga odgovarajućeg načina isporuke.

Održavanje započinje nakon uvođenja programskoga proizvoda u uporabu. Ono obuhvaća uklanjanje uočenih pogrešaka, prilagodbu promjenama u radnom okruženju, poboljšavanje sigurnosti i učinkovitosti te dodavanje novih funkcionalnosti. Rezultat održavanja mogu biti ispravci, nadogradnje i nove verzije programskoga proizvoda.

Razvoj programa započinje pisanjem *izvornoga koda* (*source code*) u uređivaču teksta ili integriranom razvojnom okruženju. Izvorni kod programa napisanoga u jeziku C++ najčešće se pohranjuje u datoteke s nastavkom `.cpp`.

Program može uključivati i *datoteke zaglavlja* (*header files*), koje sadržavaju deklaracije funkcija, tipova i drugih elemenata potrebnih programu. One se uključuju predprocesorskom direktivom `#include`. Prije prevođenja *predprocesor* (*preprocessor*) obrađuje tu i ostale predprocesorske direktive te priprema kod za prevoditelj.

Prevoditelj (*compiler*) prevodi pripremljeni izvorni kod u strojni kod i stvara *objektnu datoteku* (*object file*). Njezin nastavak ovisi o operacijskom sustavu i razvojnom okruženju, a najčešće je `.o` ili `.obj`. Objektna datoteka još nije samostalan izvršni program jer može sadržavati pozive funkcija čije se definicije nalaze u drugim objektnim datotekama ili bibliotekama.

Povezivač (*linker*) povezuje objektne datoteke s potrebnim *bibliotekama* (*libraries*), razrješava međusobne reference i stvara *izvršnu datoteku* (*executable file*). Nastavak izvršne datoteke ovisi o operacijskom sustavu; u sustavu Windows najčešće je `.exe`, dok na drugim sustavima izvršna datoteka ne mora imati poseban nastavak.

Pri pokretanju programa operacijski sustav učitava izvršnu datoteku u memoriju i započinje njezino izvođenje. Taj postupak obavlja *program za učitavanje* (*loader*), koji je dio operacijskoga sustava.

Zapamtite

Put od izvornoga koda do izvođenja programa može se sažeti ovim slijedom: izvorni kod obrađuje predprocesor, prevoditelj stvara objektnu datoteku, povezivač stvara izvršnu datoteku, a program za učitavanje prenosi je u memoriju i započinje njezino izvođenje.

Tijekom izrade i izvođenja programa mogu se pojaviti različite vrste pogrešaka:

- *pogreške pri prevođenju* (*compile-time errors*), primjerice sintaktički neispravne naredbe ili uporaba nedeklariranoga identifikatora;

- *pogreške pri povezivanju* (*link-time errors*), primjerice kada povezivač ne može pronaći definiciju pozvane funkcije;
- *pogreške pri izvođenju* (*runtime errors*), koje nastaju tijekom rada programa, primjerice zbog nedopuštenoga pristupa memoriji; te
- *logičke pogreške* (*logic errors*), zbog kojih se program izvodi, ali daje netočne rezultate.

Najčešće pogreške

Uspješno prevođenje i povezivanje ne znače nužno da je program ispravan. Program može sadržavati pogrešku pri izvođenju ili logičku pogrešku koju prevoditelj ne može otkriti. Zbog toga program nakon prevođenja treba pokrenuti i sustavno testirati različitim ulaznim podatcima.

Prevoditelj uz pogreške može ispisati i *upozorenja* (*warnings*). Program se unatoč upozorenjima često može prevesti, ali ih treba pažljivo provjeriti jer mogu upućivati na moguću pogrešku.

Za pronalaženje i uklanjanje pogrešaka koristi se *program za otkrivanje pogrešaka* (*debugger*). On omogućuje zaustavljanje programa na odabranome mjestu, izvođenje programa korak po korak te praćenje vrijednosti varijabli tijekom izvođenja. Postupak pisanja, prevođenja, povezivanja, testiranja i ispravljanja ponavlja se sve dok program ne počne raditi u skladu s postavljenim zahtjevima.

Na laboratorijskim vježbama studenti će izrađivati manje programe. Prije pisanja programa potrebno je analizirati zadatak i osmisliti rješenje, a zatim napisani program prevesti, testirati i ispraviti uočene pogreške. Izvorni kod treba biti pregledan i, prema potrebi, popraćen odgovarajućim komentarima.

1.4 Standardna zaglavlja

Standardna biblioteka jezika C++ sadržava velik broj gotovih funkcija, tipova, klasa i drugih elemenata. Svaki element standardne biblioteke deklariran je ili definiran u odgovarajućem *standardnom zaglavlju* (*standard header*). Zaglavlja se prema potrebi uključuju u program predprocesorskom direktivom `#include`.

Zaglavlja uvedena u standardnu biblioteku jezika C++ nemaju nastavak `.h`. Primjerice, za rad sa standardnim ulaznim i izlaznim tokovima uključuje se zaglavlje `<iostream>`, a za rad s tekstom zaglavlje `<string>`. Standardna biblioteka radi kompatibilnosti sadržava i zaglavlja iz standardne biblioteke jezika C, među kojima su i oblici s nastavkom `.h`.

Standardna biblioteka jezika C++ sadržava i zaglavlja koja omogućuju uporabu funkcionalnosti preuzetih iz standardne biblioteke jezika C. Njihovi nazivi najčešće započinju slovom `c` i nemaju nastavak `.h`. Primjerice, zaglavljima `<stdio.h>`, `<stdlib.h>` i `<math.h>` odgovaraju zaglavlja `<cstdio>`, `<cstdlib>` i `<cmath>`. U programima napisanima u jeziku C++ prednost se daje tim oblicima.

Nekoliko standardnih zaglavlja uključuje se ovako:

```
1 #include <iostream>
```

```
#include <string>
#include <vector>
#include <algorithm>
```

Najčešće upotrebljavana zaglavlja uključuju:

- `<iostream>` – standardni ulaz i izlaz;
- `<string>` i `<string_view>` – rad s tekstom (`<string_view>` od standarda C++17);
- `<vector>`, `<array>`, `<list>` i `<map>` – spremnici podataka;
- `<algorithm>` i `<ranges>` – algoritmi za pretraživanje, uređivanje i obradu podataka (`<ranges>` od standarda C++20);
- `<cmath>` i `<numeric>` – matematičke i numeričke operacije;
- `<fstream>` – rad s datotekama;
- `<chrono>` – mjerenje vremena i rad s datumima;
- `<random>` – generiranje pseudoslučajnih brojeva;
- `<memory>` – upravljanje dinamički stvorenim objektima;
- `<filesystem>` – rad s datotekama i direktorijima (od standarda C++17);
- `<format>` – oblikovanje tekstualnoga ispisa (od standarda C++20); te
- `<stdexcept>` – standardne iznimke.

Pojedina se zaglavlja uključuju onda kada su programu potrebni elementi koji su u njima deklarirani. Uključivanje nepotrebnih zaglavlja treba izbjegavati jer može nepotrebno produljiti vrijeme prevođenja programa.

1.4.1 Namespace `std`

Većina imena iz standardne biblioteke jezika C++ nalazi se u namespaceu `std`. Nakon uključivanja odgovarajućega zaglavlja ime se može navesti zajedno s oznakom namespacea. Primjerice, puno ime standardnoga izlaznog toka jest `std::cout`, dok je puno ime klase za rad s tekstom `std::string`.

Prefiks `std::` može se izostaviti uporabom direktive:

```
using namespace std;
```

Direktiva `using namespace std;` omogućuje uporabu svih dostupnih imena iz namespacea `std` bez navođenja prefiksa `std::`. Primjerice, umjesto `std::cout`, `std::cin` i `std::string` mogu se pisati kraća imena `cout`, `cin` i `string`.

U većim programima direktiva `using namespace std;` može uzrokovati sukobe imena, pa se preporučuje uporaba punih imena ili pojedinačnih deklaracija `using`. Međutim, na vježbama i u svim primjerima u ovoj skripti koristit ćemo direktivu `using namespace std;` radi jednostavnijega i preglednijega prikaza programskoga koda.

Zaglavlja standardne biblioteke jezika C, poput `<stdio.h>`, dostupna su u C++-u radi kompatibilnosti. Ipak, u programima napisanima u jeziku C++ prednost se daje odgovarajućim C++ zaglavljima, primjerice `<cstdio>`, `<cstdlib>` i `<cmath>`. Makronaredbe predprocesora ne pripadaju prostoru imena.

1.5 Struktura C++ programa

Osnovnu strukturu programa napisanoga u jeziku C++ razmotrit ćemo na sljedećem primjeru:

```
1  // prvi program u C++-u
2  #include <iostream>
3
4  using namespace std;
5
6  int main()
7  {
8      cout << "Hello World!";
9      return 0;
10 }
```

Program se sastoji od komentara, predprocesorske direktive, direktive `using`, funkcije `main` i naredbi koje se izvode unutar njezina tijela.

1.5.1 Komentar

Prvi redak programa sadržava komentar:

```
1  // prvi program u C++-u
```

Znakovi `//` označuju početak komentara koji završava na kraju tekućega retka. Komentari ne utječu na izvođenje programa. Programeru služe za objašnjavanje pojedinih dijelova izvornoga koda i olakšavaju njegovo razumijevanje i održavanje.

1.5.2 Uključivanje zaglavlja

Sljedeći redak uključuje standardno zaglavlje `<iostream>`:

```
1  #include <iostream>
```

Direktiva `#include` predprocesorska je direktiva. Prije prevođenja programa obrađuje je predprocesor, čime programu postaju dostupne deklaracije potrebne za uporabu standardnih ulaznih i izlaznih tokova. U ovom je primjeru zaglavlje `<iostream>` potrebno zbog uporabe objekta `cout`.

1.5.3 Direktiva `using namespace std;`

Nakon uključivanja zaglavlja navedena je direktiva:

```
1 using namespace std;
```

Kao što je prethodno objašnjeno, ova direktiva omogućuje da se u programu umjesto punoga imena `std::cout` rabi kraće ime `cout`.

1.5.4 Funkcija `main`

Sljedeći redak predstavlja zaglavlje funkcije `main`:

```
1 int main()
```

Funkcija `main` glavna je funkcija programa i izvođenje programa započinje njezinim pozivanjem. Program koji se izvodi u uobičajenom hostiranom C++ okruženju mora sadržavati točno jednu funkciju `main`. U samostalnim (*freestanding*) implementacijama, primjerice na nekim ugradbenim sustavima, zahtjev za funkcijom `main` ovisi o implementaciji.

Riječ `int` označuje da funkcija `main` vraća cjelobrojnu vrijednost operacijskom sustavu. Ta vrijednost služi kao oznaka načina na koji je program završio izvođenje.

Prazne okrugle zagrade u zapisu `main()` znače da funkcija u ovom obliku nema parametara. To ne znači da program ne može primati podatke, nego samo da funkcija `main` ne prima argumente navedene pri pokretanju programa. Drugi dopušteni oblik funkcije `main` može sadržavati parametre za prihvaćanje argumenata naredbenoga retka.

Naredbe koje pripadaju funkciji `main` nalaze se unutar vitičastih zagrada. One čine *tijelo funkcije*.

1.5.5 Ispis podataka

Naredba

```
1 cout << "Hello World!";
```

ispisuje tekst `Hello World!` na standardni izlaz, koji je najčešće povezan sa zaslonom računala. Objekt `cout` standardni je izlazni tok deklariran u zaglavlju `<iostream>`. Njegovo puno ime jest `std::cout`, ali se zbog prethodno navedene direktive `using namespace std;` može rabiti kraće ime `cout`.

Operator `<<` usmjerava vrijednost koja se nalazi s njegove desne strane u izlazni tok. Tekst koji se ispisuje zapisan je kao znakovni niz unutar dvostrukih navodnika. Ova naredba završava točkom-zarezom.

1.5.6 Završetak programa

Naredbom

```
return 0;
```

završava izvođenje funkcije `main`, a vrijednost `0` vraća se operacijskom sustavu. Ta vrijednost prema dogovoru označuje uspješan završetak programa. Ona ne potvrđuje da program nema logičkih pogrešaka, nego samo signalizira da je završio bez prijavljene pogreške.

U funkciji `main` naredba `return 0;` može se i izostaviti. Dolazak do kraja tijela funkcije `main` ima isti učinak kao izvođenje naredbe `return 0;`. U početnim ćemo je primjerima ipak navoditi kako bi povratna vrijednost funkcije bila jasno vidljiva.

1.5.7 Oblikovanje izvornoga koda

Program iz prethodnoga primjera mogao bi se napisati i s cijelim tijelom funkcije `main` u jednom retku:

```
#include <iostream>

using namespace std;

int main() { cout << "Hello World!"; return 0; }
```

U ovom primjeru promjena broja razmaka i prijeloma redaka ne mijenja značenje programa. Ipak, razmaci se ne mogu uvijek proizvoljno izostavljati jer mogu biti potrebni za razdvajanje pojedinih elemenata jezika. Razmaci unutar znakovnoga niza također su dio teksta koji program ispisuje.

Radi bolje preglednosti i razumljivosti izvornoga koda preporučuje se svaku naredbu pisati u zasebnom retku te uvlačenjem jasno označiti tijela funkcija i drugih blokova.

1.5.8 Vrste komentara

U jeziku C++ postoje dvije vrste komentara:

```
// komentar koji završava na kraju retka

/* komentar koji se proteže
    na više redaka */
```

Komentar koji započinje znakovima `//` proteže se do kraja tekućega retka. Komentar koji započinje znakovima `/*` završava znakovima `*/` i može se protezati kroz više redaka.

Blokovski komentari ne mogu se gnijezditi, odnosno jedan blokovski komentar ne može sadržavati drugi blokovski komentar. Blokovski komentar može se rabiti i za privremeno isključivanje dijela programskoga koda tijekom traženja i otklanjanja pogrešaka. Pri tome treba voditi računa o tome da isključeni dio koda ne sadržava drugi blokovski komentar.

Sljedeći primjer prikazuje uporabu obiju vrsta komentara:

```
/* moj drugi program u C++-u
    s nešto više komentara */
```

```cpp

#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World!\n";       // prvi redak ispisa
    cout << "Ovo je C++ program.\n";
    return 0;
}
```

Rezultat izvođenja programa jest:

Output

```
Hello World!
Ovo je C++ program.
```

Napomena

Ako se u tekstu koji program ispisuje rabe hrvatska slova, izvorna datoteka mora biti spremljena u kodiranju UTF-8, a prevoditelj i izlazna konzola moraju upotrebljavati isto kodiranje. U Microsoft Visual Studiju datoteka se može spremiti naredbom `File` → `Save As` → `Save with Encoding` i odabirom kodiranja `Unicode (UTF-8 with signature) - Codepage 65001`. U svojstvima projekta zatim treba otvoriti `Configuration Properties`, odabrati `C/C++`, a potom `Command Line` te u polje `Additional Options` dodati opciju `/utf-8`. Ako konzola unatoč tomu ne prikazuje hrvatska slova pravilno, na početku funkcije `main` može se postaviti izlazna kodna stranica UTF-8:

```cpp
#include <iostream>
#include <windows.h>

using namespace std;

int main()
{
    SetConsoleOutputCP(CP_UTF8);

    cout << "Treći rezultat\n";
    return 0;
}
```

Zaglavlje `<windows.h>` i funkcija `SetConsoleOutputCP` specifični su za operacijski sustav Windows i nisu dio standardnoga C++-a. Poziv nije potreban ako razvojno okruženje već pravilno prikazuje UTF-8 tekst.

Kratki sažetak

Računalni sustav povezuje sklopovlje, programsku podršku i podatke. Digitalno računalo podatke prikazuje bitovima, dok pozicijski brojevni sustavi, kodovi znakova i standardi poput Unicodea određuju kako se njihove vrijednosti tumače. Izvorni program nastaje planiranjem i pisanjem algoritma, a zatim prolazi kroz predobradu, prevođenje, povezivanje i učitavanje. Jednostavan C++ program sadržava zaglavlja, funkciju `main`, naredbe, komentare i po potrebi elemente standardne biblioteke u namespaceu `std`. Pogreške mogu nastati pri prevođenju, povezivanju ili izvođenju, a logičke se pogreške otkrivaju provjerom ponašanja programa.

Pitanja za samoprovjeru

1. Koje su osnovne sastavnice digitalnoga računala i čemu služe?
2. Što su bit i bajt?
3. Kako se vrijednost broja određuje u pozicijskome brojevnom sustavu?
4. Koji se prefiksi rabe za binarne i heksadekadske cjelobrojne literale, a kako se prepoznaje oktalni literal?
5. Koja je razlika između ASCII-ja, Unicodea i UTF-8?
6. Koje osnovne faze obuhvaća razvoj programskoga proizvoda?
7. Koje su uloge predprocesora, prevoditelja, povezivača i programa za učitavanje?
8. Po čemu se razlikuju pogreške pri prevođenju, povezivanju i izvođenju te logičke pogreške?
9. Čemu služe standardna zaglavlja i namespace `std`?
10. Koja je uloga funkcije `main`, naredbe `return 0;` i komentara u programu?

2. Tipovi podataka i operatori

Što ćete naučiti?

Nakon ovoga poglavlja moći ćete razlikovati osnovne tipove podataka, pravilno deklarirati i inicijalizirati varijable i konstante, objasniti područje važenja varijable, primijeniti aritmetičke, relacijske, logičke i bit operatore te prepoznati pogreške povezane s pretvorbom tipova i prioritetom operatora.

Svaki program sadrži u sebi podatke koji se obrađuju. Oni se općenito mogu podijeliti na podatke koji su nepromjenjive *konstante*, odnosno promjenjive *varijable*. Varijable su objekti čija se vrijednost tijekom izvođenja programa općenito može mijenjati. Pretpostavimo da trebamo zapamtiti određeni broj, npr. broj 10. Zatim neka zapamtimo još broj 5. Ako sada dodamo na prvi broj 3, pamtimo 13 i 5. Ako sada oduzmemo od prvoga broja drugi, rezultat će biti 8. Ovaj primjer ilustrira na koji način računalo "radi" s varijablama. Ako ovaj primjer prikažemo u C++ kodu imamo:

```cpp
//primjer3.cpp
#include <iostream> //predprocesorska direktiva
using namespace std;
int main() //početak programa
{ //početak bloka naredbi
    //deklaracija (obznana) varijabli
    int a, b, rezultat;
    //pridruživanje početnih vrijednosti
    a = 10;
    b = 5;
    //algoritam
    a = a + 3;
    rezultat = a - b;
    //ispis rezultata
    cout << "Rezultat je: " << rezultat << "\n";
    return 0; //završetak programa
} //kraj bloka naredbi
```

Output

```
Rezultat je: 8
```

Ovaj jednostavan primjer prikazuje rad s tri *cjelobrojne varijable* (*integer*). To znači da podatak mora imati dodijeljenu oznaku *tipa* koja govori kako se podatak pohranjuje u memoriju računala, koji su njegovi dozvoljeni rasponi vrijednosti, koje su moguće operacije s tim podacima. Nakon toga potrebno je dati *ime*, tj. *identifikator* varijablama. Identifikatori u našem primjeru su: `a`, `b` i `rezultat`.

2.1 Identifikatori

U primjerima u ovoj skripti identifikatore zapisujemo slovima engleske abecede, znamenkama i znakom podvlake. Takva nastavna konvencija daje prenosiv i lako čitljiv kod. Suvremeni C++ dopušta i određene Unicode znakove u identifikatorima, ali njihova uporaba ovisi o kodiranju izvorne datoteke i podršci razvojnih alata. Pri davanju imena treba poštovati sljedeća pravila:

1. U primjerima identifikator se sastoji od slova engleske abecede, znamenki i znaka podvlake _ (*underscore*).
2. Prvi znak mora biti slovo ili _ (znak za podcrtavanje).
3. Duljina identifikatora nije unaprijed ograničena na 32 znaka.
4. Razmak i interpunkcijski znakovi poput `-`, `.`, `?` ili `@` ne smiju se upotrebljavati.
5. Nije dozvoljena uporaba jedne od ključnih riječi (*key words*):

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

Sljedeće riječi alternativni su zapisi operatorskih tokena i također imaju položaj ključnih riječi:

```
and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq,
```

Popis ključnih riječi proširivao se novim izdanjima standarda. U suvremenome C++-u ključne su riječi, među ostalima, i `alignas`, `alignof`, `char8_t`, `concept`, `constexpr`, `consteval`, `constinit`, `co_await`, `co_return`, `co_yield`, `decltype`, `noexcept`, `nullptr`, `requires`, `thread_local` i `static_assert`. Pojedini prevoditelji mogu rezervirati i dodatne riječi.

Napomena

C++ jezik razlikuje VELIKA i mala slova (*case sensitive*), što znači da identifikatori `REZULTAT`, `rezultat` i `Rezultat` predstavljaju različite varijable.

Zapamtite

Dobro ime opisuje značenje podatka. Imena `brojBodova` i `cijenaBezPoreza` čitljivija su od imena `x` i `y`. Svako ime koje sadržava dvije uzastopne podvlake rezervirano je za implementaciju. Rezervirana su i imena koja počinju podvlakom iza koje slijedi veliko slovo, dok su sva imena koja počinju podvlakom rezervirana u globalnome prostoru imena. Zato vlastite identifikatore

ne treba započinjati podvlakom.

2.2 Tipovi podataka

U memoriju računala spremaju se različiti podaci. Najmanja adresabilna jedinica memorije u C++-u jest bajt. Standard jamči da je `sizeof(char) == 1` i da bajt ima najmanje 8 bitova, ali ne propisuje da svaki bajt na svakoj platformi mora imati točno 8 bitova. Na današnjim osobnim računalima bajt je u pravilu oktet, odnosno skup od 8 bitova. Grupiranjem više bajtova mogu se prikazati složeniji tipovi podataka.

Podaci (*data*) u računalu sastoje se od:

- binarno kodiranih znakova i
- binarno kodiranih numeričkih vrijednosti.

Za primjenu u digitalnom računalu, kodiranje binarnoga niza je postupak kojim se određenome broju, slovu ili nekom kontrolnom signalu računala pridjeljuje određena kombinacija binarnih znamenki.

Znak (*character*) je tip podataka koji predstavlja element dogovorno usvojenoga skupa različitih simbola koji su namijenjeni obradi podataka (slova abecede, numeričke znamenke, znakovi interpunkcije itd.), što je navedeno u prethodnom poglavlju – ASCII kod. Niz znakova često se tretira kao cjelina i naziva se string.

Skupu pozitivnih cijelih brojeva ili kardinalnih brojeva (*unsigned integers*) pripadaju svi prirodni brojevi i nula. Način njihova kodiranja opisan je prije u poglavlju kod pretvorbe dekadskoga broja u binarni. Većina današnjih procesora za kodiranje cijelih brojeva s predznakom (*integers*) koristi tzv. komplementarni brojevni sustav. Pun komplement n-znamenkastoga broja N_x u brojevnome sustavu baze x matematički se definira izrazom:

$$N'_x = x^n - N_x.$$

Primjerice, u dekadskome sustavu komplement troznamenkastoga broja 666 jest $10^3 - 666 = 334$. Ako se n-znamenkasti broj i njegov komplement zbroje, u rezultatu dobivamo jedinicu nakon koje slijedi n nula. U našem primjeru vrijedi $666 + 334 = 1000$. U binarnome sustavu pun komplement naziva se komplement dvojke i vrijedi:

$$N'_2 = 2^n - N_2.$$

Tablica 2.1: Osnovni tipovi podataka i njihove uobičajene veličine

Ime	Tipično bajtova	Opis	Najmanji zajamčeni raspon ili napomena
`char`	1	znakovni ili mali cjelobrojni tip	tip je s predznakom ili bez njega, ovisno o implementaciji
`short`	2	kratki cijeli broj	najmanje -32767 do 32767
`int`	4	osnovni cjelobrojni tip	najmanje -32767 do 32767
`long`	4 ili 8	cijeli broj najmanje raspona tipa **`int`**	najmanje -2147483647 do 2147483647
`long long`	8	veliki cijeli broj	najmanje $-(2^{63}-1)$ do $2^{63}-1$
`float`	4	broj s pomičnim zarezom	preciznost najmanje 6 decimalnih znamenki
`double`	8	realni tip veće preciznosti	preciznost najmanje 10 decimalnih znamenki
`long double`	8, 12 ili 16	proširena realna preciznost	format ovisi o prevoditelju i platformi
`bool`	1	logički tip	vrijednosti **`false`** i **`true`**
`wchar_t`	2 ili 4	široki znakovni tip	veličina ovisi o platformi

Napomena

Veličine tipova ovise o implementaciji, prevoditelju i platformi. Stvarne vrijednosti provjeravaju se operatorom **`sizeof`**, konstantom `CHAR_BIT` iz zaglavlja `<climits>` te predloškom `std::numeric_limits` iz zaglavlja `<limits>`. Uz ove tipove podataka postoje još i **`void`** i pokazivači (*pointers*), o čemu će biti riječi kasnije.
Standard C++17 za svaki cjelobrojni tip propisuje najmanji raspon vrijednosti koji implementacija mora podržavati. Primjerice, za tip **`short`** standard jamči najmanje raspon od -32767 do 32767. Na sustavima na kojima taj tip ima 16 bitova i rabi prikaz komplementom dvojke njegov je uobičajeni raspon od -32768 do 32767. Stvarni raspon pojedinoga tipa može biti i širi.

Svojstva tipova mogu se provjeriti i u samome programu:

```
#include <climits>
#include <iostream>
#include <limits>

using namespace std;

int main()
{
    cout << "Bitova u bajtu: " << CHAR_BIT << '\n';
    cout << "sizeof(int): " << sizeof(int) << '\n';
    cout << "Najveci int: "
         << numeric_limits<int>::max() << '\n';
    return 0;
}
```

Puni komplement ili komplement dvojke koristi se za označavanje negativnih brojeva. Primjerice, komplement dvojke broja $+1$ za $n = 4$ iznosi:

$$N_2' = 2^4 - 1 = 10000 - 0001 = 1111.$$

Kada se broj i njegov komplement zbroje, rezultat bi trebao biti 0. U našem primjeru to vrijedi jer su

prva četiri bita zbroja jednaka nuli, dok je peti bit jednak jedinici, ali se on u 4-bitnom sustavu odbacuje. U sustavu komplementa dvojke pozitivni brojevi uvijek imaju MSB = 0, a negativni brojevi imaju MSB = 1. Pogledajmo na primjeru što će se dogoditi ako se zbroje dva pozitivna broja koji imaju bitove ispod MSB jednake jedinici ($4 + 5$):

$$\begin{array}{r} 0100\ +4 \\ +0101\ +5 \\ \hline 1001\ -7 \end{array}$$

Četverobitno sklopovsko zbrajalo odbacilo bi prijenos izvan četvrtoga bita, pa bi niz `1001`, protumačen u komplementu dvojke, predstavljao vrijednost -7. Time se pokazuje *prekoračenje raspona* četverobitnoga prikaza: matematički rezultat 9 ne može se prikazati četirima bitovima s predznakom. To nije opći opis rezultata C++ izraza. U C++-u prekoračenje cjelobrojnoga tipa s predznakom ima nedefinirano ponašanje, dok se aritmetika tipova bez predznaka izvodi modulo 2^n. Zbroj dvaju pozitivnih operanada može se prikazati u istome n-bitnom tipu s predznakom samo ako nije veći od $2^{n-1} - 1$.

Zbog toga je najveći pozitivan broj koji se može predstaviti u sustavu komplementa dvojke:

$$\text{max_int} = (01111 \ldots 1111) = 2^{n-1} - 1,$$

a najveći iznos negativnoga broja iznosi:

$$\text{min_int} = (10000 \ldots 0000) = -2^{n-1}.$$

Vidljivo je da se može prikazati jedna negativna vrijednost više nego pozitivnih vrijednosti. Razlog je to što komplement dvojke ima samo jedan zapis nule; nula nije ni pozitivan ni negativan broj.

Običan komplement binarnoga broja (komplement jedinice) dobije se zamjenom svih jedinica nulom i obratno. Iznos broja koji se dobije na ovaj način računa se prema izrazu:

$$\overline{N}_2 = 2^n - N_2 - 1.$$

Komplement jedinice povijesno se rabio za prikaz cijelih brojeva s predznakom, ali ima dva zapisa nule: pozitivnu nulu `0000` i negativnu nulu `1111`. Komplement jedinice ipak je koristan za jednostavno izračunavanje komplementa dvojke jer vrijedi:

$$N_2' = \overline{N}_2 + 1.$$

Pun komplement izračuna se tako da se najprije izračuna običan komplement te mu se doda jedinica. U 8-bitnoj notaciji iznosi, primjerice:

komplement dvojke broja 6		pun komplement od 0	
`00000110`	$+6$	`00000000`	0
`11111001`	običan komplement od $+6$	`11111111`	običan komplement od 0
`+ 1`	dodaj 1	`+ 1`	dodaj 1
`11111010`	-6 u komplementu dvojke	`100000000`	prijenos izvan osam bitova

U navedenome primjeru punoga komplementa od 0 jedinica predstavlja deveti bit, te se stoga u 8-bitnom sustavu odbacuje pa rezultat opet predstavlja nulu. Komplement dvojke omogućuje jednoznačno određivanje nule, pa je stoga izabran za predstavljanje cijelih brojeva, a ne običan komplement.

Realni brojevi u matematici i višim programskim jezicima zapisuju se na način da se cijeli i decimalan dio odvoje decimalnim zarezom (npr. 57,495), a koristi se i eksponentski format (*scientific format*), $(0{,}57495 \cdot 10^2)$. U programskome jeziku C++ koristi se sličan zapis kao i u matematici, s razlikom da se umjesto decimalnoga zareza koristi "decimalna točka", a potencija broja 10 označava se velikim slovom E ili malim slovom e (tablica 2.2).

Tablica 2.2: Matematički zapis i zapis realnih brojeva u C++

Matematički zapis	Ekvivalent u C++
$1{,}537$	`1.537`
$0{,}537$	`0.537` ili `.537`
$-153{,}7 \cdot 10^{-2}$	`-153.7e-2` ili `-153.7E-2`
$-0{,}01537 \cdot 10^2$	`-0.01537e2`, `-0.01537E2` ili `-0.01537e+2`

Eksponentski format sastoji se od dva dijela:

$$\text{eksponentski dekadski format} = \text{mantisa} \cdot 10^{\text{eksponent}}.$$

Mantisa se zapisuje kao običan dekadski broj s predznakom, a eksponent se zapisuje kao cijeli broj. Prednost uporabe eksponentskoga formata je u lakšemu zapisu vrlo velikih i vrlo malih brojeva. Uočimo da se promjenom eksponenta pomiče položaj decimalnoga zareza.

Kodiranje fiksnim položajem binarne točke (fixed point numbers)

Umjesto pojma decimalnoga zareza uvodi se pojam binarne točke. Opći oblik zapisivanja realnoga broja s fiksnim položajem binarne točke, u slučaju da se N znamenki koristi za označavanje cijelih vrijednosti, a n znamenki za označavanje razlomljenih vrijednosti po bazi 2, glasi:

$$\left(b_{N-1}b_{N-2}\ldots b_0.b_{-1}b_{-2}\ldots b_{-n}\right)_2,$$

a iznos mu se računa prema izrazu:

$$V = b_{N-1}2^{N-1} + \cdots + b_0 2^0 + b_{-1}2^{-1} + b_{-2}2^{-2} + \cdots + b_{-n}2^{-n}, \quad b_i \in \{0, 1\}.$$

Ako se vrijednost pomnoži s 2^n, može se pohraniti kao cijeli broj X, pri čemu zapis X predstavlja stvarnu vrijednost $X \cdot 2^{-n}$. Pri zbrajanju i oduzimanju brojeva s jednakim brojem razlomljenih bitova rezultat zadržava isto skaliranje. Pri množenju dvaju takvih zapisa nastaje skaliranje 2^{-2n}, pa rezultat treba ponovno skalirati, najčešće pomakom za n bitova udesno. Pri dijeljenju je, radi očuvanja razlomljenoga dijela, često potrebno prije dijeljenja dodatno skalirati djeljenik. Kodiranje fiksnim položajem binarne točke ima ograničen raspon i zahtijeva pažljivo upravljanje skaliranjem, ali se često rabi u ugrađenim sustavima i obradi signala kada su važni predvidljivo vrijeme izvođenja i nadzor nad preciznošću.

Kodiranje pomičnim položajem binarne točke (floating point numbers)

Eksponentski format uzet je kao temelj za kodiranje realnih brojeva i u binarnome brojevnom sustavu. Kodira se prema izrazu:

$$F = (-1)^s m \cdot 2^e,$$

gdje m predstavlja mantisu, e je eksponent dvojke, a $s \in \{0, 1\}$ određuje predznak broja.

Eksponent i mantisa kodiraju se u binarnome brojevnom sustavu. Eksponent se kodira kao cijeli broj, a mantisa kao binaran broj s fiksnim položajem binarne točke. Ideja je jednostavna: promjenom eksponenta pomiče se i položaj binarne točke iako se mantisa zapisuje fiksnim položajem binarne točke. Primjerice, neka je broj kodiran u obliku:

$$0.00001xxxxxxxxx \cdot 2^e,$$

gdje x može biti 0 ili 1. Ovaj oblik zapisa realnoga broja nije normaliziran. Pomaknemo li binarnu točku za pet mjesta udesno, dobije se ekvivalentan zapis:

$$1.xxxxxxxxx00000 \cdot 2^{e-5}.$$

Pomak binarne točke za pet mjesta udesno u zapisu mantise kompenzira se smanjenjem eksponenta za pet. Zapis u kojemu je znamenka neposredno lijevo od binarne točke uvijek jedinica naziva se normalizirani zapis. Njime se raspoloživi bitovi mantise iskorištavaju za postizanje najveće moguće preciznosti. Za prikaz vrijednosti vrlo bliskih nuli IEEE 754 rabi i nenormalizirani zapis, bez implicitne vodeće jedinice mantise. Time se omogućuje postupan prijelaz prema nuli, uz smanjenu preciznost.

Pozitivni i negativni eksponenti pohranjuju se u polju eksponenta kao nenegativan binarni broj. Umjesto stvarne vrijednosti eksponenta e zapisuje se pomaknuti eksponent:

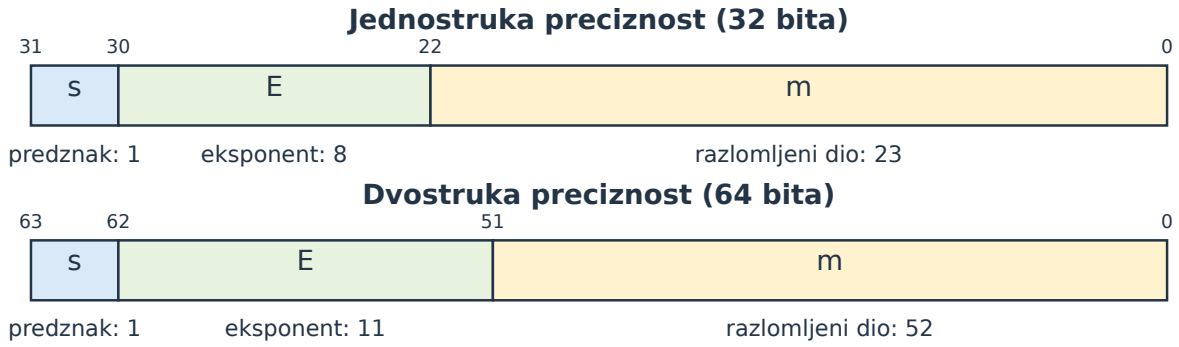
$$E = e + \text{pomak},$$

gdje je pomak (*bias*) konstantna pozitivna vrijednost. Za IEEE 754 jednostruku preciznost pomak iznosi 127, a za dvostruku preciznost 1023. Polje eksponenta sastavljeno od samih nula rezervirano je za nulu

i nenormalizirane brojeve, a polje sastavljeno od samih jedinica za beskonačnost i NaN. Ostali zapisi polja eksponenta pripadaju normaliziranim brojevima.

Standardni C++ ne zahtijeva da svaki realni tip na svakoj platformi bude zapisan prema IEEE 754, ali taj format koristi većina suvremenih računala. Svojstvo se može provjeriti izrazom `std::numeric_limits<T>::is_iec559`.

Raspored bitova za predznak, eksponent i mantisu u formatima jednostruke i dvostruke preciznosti prikazan je na slici 2.1.



Slika 2.1: Raspored polja u formatima IEEE 754

Tablica 2.3: Svojstva binarnih formata IEEE 754 jednostruke i dvostruke preciznosti

Parametar	jednostruki (SINGLE)	dvostruki (DOUBLE)
ukupan broj bitova	32	64
broj bitova eksponenta	8	11
broj zapisanih bitova razlomljenoga polja	23	52
efektivna preciznost normalizirane mantise	24 bita	53 bita
pomak	+127	+1023
$E_{\max}$	255	2047
$E_{\min}$	0	0
najmanja pozitivna nenormalizirana vrijednost	$1{,}4 \cdot 10^{-45}$	$4{,}94 \cdot 10^{-324}$
najmanja pozitivna normalizirana vrijednost	$1{,}175 \cdot 10^{-38}$	$2{,}225 \cdot 10^{-308}$
najveća konačna vrijednost	$3{,}4028 \cdot 10^{38}$	$1{,}797 \cdot 10^{308}$

Primjer: pretvorba broja $-1313{,}3125$ u 32-bitni format IEEE 754

Broj je negativan, pa je bit predznaka jednak 1. Ostala se polja određuju iz apsolutne vrijednosti $1313{,}3125$.

Najprije pretvaramo cijeli dio:

$$1313_{10} = 10100100001_2.$$

Razlomljeni dio pretvara se uzastopnim množenjem s 2. Cijeli dio svakoga rezultata daje sljedeći binarni bit, a postupak se nastavlja s preostalim razlomljenim dijelom:

Tablica 2.4: Pretvorba razlomljenoga dijela $0{,}3125$ u binarni zapis

Korak	Množenje	Dobiveni bit	Preostali razlomljeni dio
1	$0{,}3125 \cdot 2 = 0{,}625$	0	$0{,}625$
2	$0{,}625 \cdot 2 = 1{,}25$	1	$0{,}25$
3	$0{,}25 \cdot 2 = 0{,}5$	0	$0{,}5$
4	$0{,}5 \cdot 2 = 1{,}0$	1	0

Čitanjem dobivenih bitova odozgo prema dolje slijedi:

$$0{,}3125_{10} = 0.0101_2,$$

pa je:

$$1313{,}3125_{10} = 10100100001.0101_2.$$

Broj zatim normaliziramo pomicanjem binarne točke za deset mjesta ulijevo:

$$10100100001.0101_2 = 1.01001000010101_2 \cdot 2^{10}.$$

Stvarni je eksponent $e = 10$, pa pomaknuti eksponent iznosi:

$$E = e + 127 = 10 + 127 = 137_{10} = 10001001_2.$$

Vodeća jedinica normalizirane mantise ne zapisuje se. Bitovi iza binarne točke dopunjuju se nulama do ukupno 23 bita:

$$m = 01001000010101000000000.$$

Tri polja 32-bitnoga zapisa stoga su:

Tablica 2.5: Polja 32-bitnoga zapisa broja $-1313{,}3125$

Polje	Broj bitova	Vrijednost
predznak	1	`1`
eksponent	8	`10001001`
mantisa	23	`01001000010101000000000`

Spajanjem polja dobiva se konačni binarni zapis:

$$\underbrace{1}_{\text{predznak}}\ \underbrace{10001001}_{\text{eksponent}}\ \underbrace{01001000010101000000000}_{\text{mantisa}} = 11000100101001000010101000000000_2.$$

Grupiranjem bitova u skupine od četiri dobiva se i heksadekadski zapis:

$$1100\ 0100\ 1010\ 0100\ 0010\ 1010\ 0000\ 0000 = \text{C4A42A00}_{16}.$$

Prema tome:

$$-1313{,}3125_{10} = 11000100101001000010101000000000_2 = \text{C4A42A00}_{16}.$$

Polje eksponenta $E = 0$ rezervirano je za nulu i nenormalizirane brojeve, a vrijednost $E = E_{\text{max}}$ za NaN i beskonačnost. Zapis formata interpretira se na sljedeći način:

1. Ako je $E = E_{\text{max}}$ i $m \neq 0$, kodna riječ predstavlja NaN, bez obzira na vrijednost predznaka s.
2. Ako je $E = E_{\text{max}}$ i $m = 0$, kodna riječ predstavlja $(-1)^s \infty$.
3. Ako je $E_{\text{min}} < E < E_{\text{max}}$, kodna riječ jednostruke preciznosti predstavlja broj $(-1)^s 2^{E-127}(1.m)$, tj. predstavlja normalizirani realni broj. Vodeća jedinica mantise podrazumijeva se i ne zapisuje.
4. Ako je $E = 0$ i $m = 0$, kodna riječ predstavlja broj $(-1)^s(0)$.
5. Ako je $E = 0$ i $m \neq 0$, kodna riječ jednostruke preciznosti predstavlja broj $(-1)^s 2^{1-127}(0.m)$, tj. predstavlja nenormalizirani realni broj. Vodeća jedinica nije implicitna, pa se takvim zapisom prikazuju vrijednosti između nule i najmanje pozitivne normalizirane vrijednosti, uz postupno smanjivanje preciznosti. Minimalne vrijednosti obaju zapisa navedene su u tablici 2.3.

Normaliziranim zapisom postiže se približno 7 značajnih decimalnih znamenki za jednostruki format, odnosno približno 15–17 znamenki za dvostruki format.

Najčešće pogreške

Realni brojevi imaju ograničenu preciznost. Zbog binarnoga zapisa izraz `0.1 + 0.2` ne mora biti bit-po-bit jednak literalu `0.3`. Rezultati realnih izračuna zato se često uspoređuju uz prikladnu toleranciju, a ne operatorom == bez dodatne provjere.

Programer koji programira u višemu programskom jeziku ne mora znati kako se neki broj kodira u procesoru ili memoriji računala. Njega zanimaju:

- pravila za zapis literalnih konstanti,
- veličina zauzeća memorije,
- maksimalna vrijednost broja,
- minimalna vrijednost broja,
- te broj točnih decimala.

2.3 Deklaracije varijabli

Kako bismo mogli upotrebljavati varijable u programu, one najprije moraju biti deklarirane:

```
int a, b, rezultat;
float broj;
unsigned short broj_cipela;
signed int moj_racun; /* isto kao: int moj_racun; */
unsigned godina_rodjenja; /* isto kao: unsigned int godina_rodjenja; */
```

Ovo su valjani primjeri deklaracije varijabli. Ekvivalentni oblici navedeni su u komentarima kako se isti identifikator ne bi definirao dvaput u istome području važenja. Kao što se vidi iz primjera `primjer3.cpp`, moguće je nekoliko varijabli istoga tipa deklarirati u jednome retku koda.

2.4 Inicijalizacija varijabli

Ako se lokalna varijabla ugrađenoga tipa deklarira bez inicijalizatora, njezina početna vrijednost može biti neodređena (*indeterminate value*). Takvu vrijednost ne treba čitati prije nego što se varijabla valjano inicijalizira ili joj se dodijeli vrijednost. Zbog toga se preporučuje inicijalizirati varijable već pri njihovoj deklaraciji.

Inicijalizacija je zadavanje početne vrijednosti varijabli u trenutku njezina stvaranja. Ne treba je poistovjećivati s naknadnim pridruživanjem vrijednosti već postojećoj varijabli.

U C++-u se inicijalizacija može zapisati na nekoliko načina:

```
int a = 0;  // inicijalizacija znakom = (copy-initialization)
int b(0);   // izravna inicijalizacija (direct-initialization)
int c{0};   // inicijalizacija listom (list-initialization)
int d{};    // vrijednosna inicijalizacija: d ima vrijednost 0
```

Inicijalizacija vitičastim zagradama uvedena je standardom C++11. Njezin je službeni engleski naziv *list-initialization*, dok se u literaturi često susreću i nazivi *brace initialization* i *uniform initialization*. Naziv *uniform initialization* naglašava da se isti oblik zapisa može upotrebljavati za inicijalizaciju različitih vrsta objekata, ali nije službeni naziv toga oblika inicijalizacije u C++ standardu.

U suvremenom C++-u često se preporučuje inicijalizacija vitičastim zagradama. Takav zapis ujednačuje način inicijalizacije različitih objekata i sprječava neželjene sužavajuće pretvorbe (*narrowing conversions*). Primjerice:

```
double x = 3.8;

int a(x);  // dopusteno: a dobiva vrijednost 3
int b{x};  // pogreska: suzavajuca pretvorba
```

Ako je sužavajuća pretvorba namjerna, treba je izričito naznačiti:

```
int b{static_cast<int>(x)};
```

Inicijalizacija oblim zagradama ispravna je, ali se za jednostavnu inicijalizaciju često izbjegava jer određeni zapisi mogu nalikovati deklaraciji funkcije. Primjerice:

```
Tip objekt(); // deklaracija funkcije, a ne stvaranje objekta
```

Ovaj redak ne stvara objekt naziva `objekt`, nego deklarira funkciju bez parametara koja vraća vrijednost tipa `Tip`. Uporabom vitičastih zagrada dvosmislenost se uklanja:

```
Tip objekt{}; // stvara i inicijalizira objekt
```

Ipak, oble i vitičaste zagrade nisu uvijek međusobno zamjenjive. Kod razreda koji imaju konstruktor s parametrom tipa `initializer_list`, inicijalizacija vitičastim zagradama može odabrati drugi konstruktor i tako promijeniti značenje naredbe. To se može vidjeti na primjeru spremnika `vector`:

```
vector<int> a(10);    // 10 elemenata vrijednosti 0
vector<int> b{10};    // jedan element vrijednosti 10

vector<int> c(10, 2); // 10 elemenata vrijednosti 2
vector<int> d{10, 2}; // dva elementa: 10 i 2
```

U takvim se slučajevima oble zagrade upotrebljavaju kako bi se odabrao željeni konstruktor. One, dakle, nisu potrebne zato što se objekt na drugi način ne bi mogao inicijalizirati, nego zato što bi inicijalizacija vitičastim zagradama imala drukčije značenje.

Za jednostavne varijable ugrađenih tipova u pravilu se može dati prednost inicijalizaciji vitičastim zagradama:

```
int broj{0};
double prosjek{0.0};
char znak{};
bool uvjet{false};
```

Prazne vitičaste zagrade vrijednosno inicijaliziraju navedene varijable: cjelobrojni i realni tipovi dobivaju vrijednost nula, `char` dobiva vrijednost znaka s kodom nula, a `bool` vrijednost `false`.

2.5 Područje važenja i trajanje objekata

Područje važenja (*scope*) određuje u kojemu je dijelu programa neko ime vidljivo, trajanje pohrane određuje koliko dugo postoji prostor namijenjen objektu, a životni vijek objekta razdoblje je od njegova nastanka do prestanka postojanja. Ta su tri pojma povezana, ali nisu istovjetna.

Ime varijable definirane izvan svih funkcija ima područje važenja od mjesta deklaracije do kraja prevodive jedinice, uz uobičajena pravila skrivanja imena. Pristup takvoj varijabli iz druge prevodive jedinice zahtijeva odgovarajuću deklaraciju i ovisi o povezivanju. Globalne varijable mogu biti korisne kada podatak mora biti dostupan većemu broju funkcija, ali ih treba rabiti odmjereno zbog veće povezanosti dijelova programa.

```cpp
// primjer vidljivosti varijabli // komentar
#include <iostream>             // zaglavlje
using namespace std;

int Cjelobrojna;                // globalne varijable
char Znak;                      // za brojevne tipove one se automatski
char Niz_Znakova[20];           // inicijaliziraju na 0 ako im drugačije
unsigned Broj_Djece;            // ne postavimo početnu vrijednost.
int main()                      // glavna i nužna funkcija
{                               // početak funkcije
    unsigned short godine;      // lokalne varijable
    float abroj;                // njihova vrijednost neodređena je
    float drugi_broj;           // prilikom deklaracije
    cout << "Unesi broj godina";     // naredba
    cin >> godine;                    // naredba
    // ... tijelo funkcije
    return 0;
}                               // kraj funkcije
```

Ime lokalne varijable vidljivo je u bloku u kojemu je deklarirano, od mjesta deklaracije nadalje, te u ugniježđenim blokovima ako nije skriveno deklaracijom istoga imena. Lokalni objekt po zadanome ima automatsko trajanje pohrane i njegov životni vijek završava izlaskom iz bloka. Implementacije takve objekte često smještaju na *stack*, ali C++ standard ne propisuje takav fizički smještaj. Lokalna varijabla može biti deklarirana i kao **static** ili `thread_local`; zato nisu sve lokalne varijable automatske.

```cpp
auto i = 42;       // tip int izvodi se iz inicijalizatora
auto x = 3.14;     // tip double izvodi se iz inicijalizatora
auto znak = 'A';   // tip char izvodi se iz inicijalizatora
```

Napomena

U starijim inačicama jezika ključna riječ **auto** označavala je automatsko trajanje lokalne varijable. Od standarda C++11 ona služi za izvođenje tipa iz inicijalizatora. Zapisi `auto int i;` i `auto double x;` više nisu valjani C++.

Objekti definirani u prostoru imena i lokalni objekti deklarirani riječju **static** imaju statičko trajanje pohrane. Lokalna statička varijabla zadržava vrijednost između poziva funkcije. Takve se varijable često rabe za brojače ili druge podatke koji moraju preživjeti završetak poziva.

U sljedećem primjeru varijabla `brojac` inicijalizira se samo pri prvom pozivu funkcije, a njezina se vrijednost zadržava između poziva:

```cpp
int sljedeci_broj()
{
    static int brojac{0};
    return ++brojac;
}
```

Prvi poziv funkcije vraća 1, drugi 2, treći 3 itd. Bez ključne riječi **static** varijabla `brojac` pri svakome bi se pozivu ponovno inicijalizirala na 0 pa bi funkcija uvijek vraćala 1.

Prilikom pisanja većih programa izvorni kod može se nalaziti u više datoteka. Područje u kojemu vrijedi ime globalne varijable počinje od mjesta deklaracije. Ako se varijabla definira u jednoj, a rabi u drugoj prevodivoj jedinici, u drugoj je potrebna deklaracija s `extern`. Primjerice, ako je u jednoj datoteci definirana globalna varijabla:

```
int x{42};
```

druga datoteka koja je želi rabiti mora sadržavati deklaraciju:

```
extern int x;
```

U starijim inačicama jezika ključnom riječju **register** prevoditelju se sugeriralo da varijablu smjesti u procesorski registar. Suvremeni prevoditelji sami optimiraju raspored podataka, a **register** se od standarda C++17 više ne upotrebljava kao specifikator pohrane. Programer ne treba pokušavati tom riječju upravljati registrima.

2.6 Literali i konstante

Literali su neposredni zapisi vrijednosti u izvornome kodu. Objekti deklarirani riječima `const` ili `constexpr` imaju imenovanu vrijednost koja se programom ne mijenja.

2.6.1 Cjelobrojni literali

Primjeri cjelobrojnih literala jesu:

```
200
7
273  // u izrazu -273 znak - jest unarni operator
```

Također se konstante mogu definirati i u drugim brojevnim sustavima:

```
75   // dekadska
0113 // oktalna
0x4b // heksadekadska
```

2.6.2 Realni literali

Primjeri realnih konstanti su:

```
3.14159 // 3.14159
6.02e23 // 6.02 * 10^23
1.6e-19 // 1.6 * 10^-19
```

```
3.0     // 3.0
```

2.6.3 Znakovni i stringovni literali

Znakovni literal zapisuje jedan znak između jednostrukih navodnika, a stringovni literal niz znakova između dvostrukih navodnika:

```
'z'                  // znakovni literal
'p'                  // znakovni literal
"Hello world"        // stringovni literal
"Koliko imas godina?"
```

Postoje i izlazne sekvence (*escape sequences*) za zapis znakova s posebnim značenjem. Neke su najčešće navedene u tablici 2.6.

Tablica 2.6: Izlazne sekvence

Kod	Značenje
`\n`	novi redak (newline)
`\r`	povratak na početak tekućega retka (carriage return)
`\t`	horizontalni tabulator (tabulation)
`\v`	vertikalni tabulator (vertical tabulation)
`\b`	povratak kursora za jedno mjesto natrag (backspace)
`\f`	pomak za jednu stranicu naprijed (page feed)
`\a`	zvučni signal upozorenja (alert/beep)
`\'`	jednostruki znak navodnika (single quotes)
`\"`	dvostruki znak navodnika (double quotes)
`\?`	znak upitnika (question mark)
`\\`	obrnuta kosa crta (backslash)
`\0`	znak s vrijednošću nula; često označuje kraj znakovnoga niza
`\ooo`	znak čiji je kôd zadan oktalno s 1, 2 ili 3 znamenke
`\xhh…`	znak čiji je kôd zadan jednom ili više heksadekadskih znamenki

```
'\n' // novi redak
'\t' // horizontalni tabulator
```

2.6.4 Imenovane konstante

Kada u programu koristimo konstante, zgodno ih je definirati na početku programa. Razlog za to je što, ako se nakon nekog vremena treba promijeniti vrijednost konstante, tada se ne mora u cijelom kodu mijenjati vrijednost (na svim mjestima gdje se konstanta upotrebljava), već samo na jednom mjestu, tamo gdje je definirana. U suvremenome C++-u prednost imaju `constexpr` i `const`, jer takve konstante imaju tip i poštuju pravila područja važenja. Primjeri:

```
constexpr double pi{3.141592653589793};
```

```
constexpr int sirina{100};
constexpr int duzina{100};
const int ptt{21000};
```

Tip konstante mora biti naveden ili izveden pomoću riječi `auto`; C++ ne pretpostavlja automatski tip `int` kada je tip izostavljen.

2.6.5 Makronaredbe

Pretprocesorska direktiva `#define` omogućuje tekstualnu zamjenu prije prevođenja:

```
#define PI 3.14159265 // bez točke-zareza na kraju
```

Pretprocesor će svako pojavljivanje imena `PI` zamijeniti navedenim nizom znakova. Takva makronaredba nema tip i ne poštuje uobičajena pravila područja važenja.

Zapamtite

Makro `#define` prikladan je ponajprije za uvjetno prevođenje i druge zadaće pretprocesora, a ne kao zamjena za tipizirane C++ konstante.

2.7 Operatori

2.7.1 Operator pridruživanja (=)

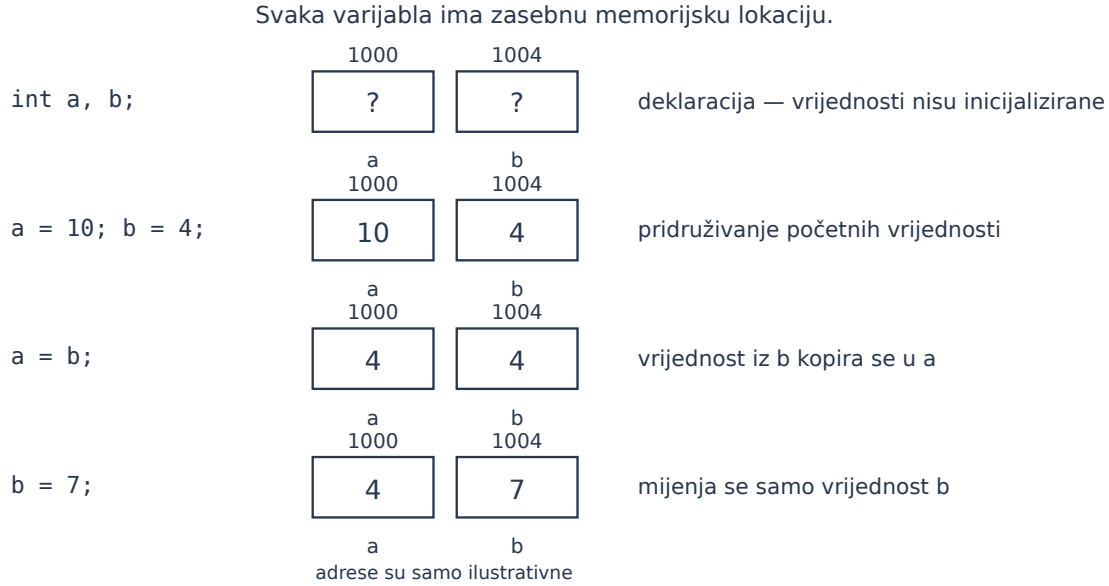
Operator pridruživanja (*assignment*) pridružuje vrijednost varijabli:

```
a = 5;
```

na način da se vrijednost 5 pridruži varijabli `a`.

- Izraz lijevo od operatora pridruživanja mora označavati objekt koji se može mijenjati (*modifiable lvalue*), primjerice nekonstantnu varijablu.
- Izraz desno od operatora pridruživanja daje vrijednost koja se pridružuje lijevoj strani. To može biti literal, varijabla, funkcijski poziv, rezultat operacije ili njihova kombinacija.

Smjer pridruživanja vrijednosti prikazan je na slici 2.2.



Slika 2.2: Deklaracija varijabli i naknadno pridruživanje vrijednosti

Vrijednost s desne strane uvijek se pridružuje na lijevo i nikada obratno:

```
a = b;
```

gdje se vrijednost varijable `b` pridružuje varijabli `a`.

U složenome izrazu pridruživanje se najprije obavlja unutar zagrada:

```
a = 2 + (b = 5);
```

ekvivalentan je izrazu:

```
b = 5;
a = 2 + b;
```

```
a = b = c = 5;
```

prvo se vrijednost 5 pridruži varijabli `c`, zatim varijabli `b` i na kraju varijabli `a`.

2.7.2 Aritmetički operatori

Postoje sljedeći aritmetički operatori:

Operator	Značenje
`+`	zbrajanje
`-`	oduzimanje
`*`	množenje
`/`	dijeljenje
`%`	modulo operator

Operator `%` daje ostatak cjelobrojnoga dijeljenja i zahtijeva cjelobrojne operande. Tako se izrazom `a = 11 % 3;` varijabli `a` pridružuje vrijednost 2. Pri cjelobrojnome dijeljenju razlomljeni se dio odbacuje prema nuli, pa izraz `11 / 3` daje 3. Dijeljenje i računanje ostatka s nulom nisu dopušteni. Pri radu s negativnim operandima količnik se zaokružuje prema nuli, a za ostatak vrijedi $(a/b) \cdot b + a\%b = a$ kada je količnik prikaziv odgovarajućim tipom.

Binarno zbrajanje obavlja se kao i dekadsko zbrajanje, osim što se prijenos na sljedeće značajnije mjesto ne obavlja nakon postignutog zbroja 10, već nakon 2 ($1 + 1$).

Primjer:

$$\begin{array}{r} 101 = 5_{10} \\ +010 = 2_{10} \\ \hline 111 = 7_{10} \end{array} \qquad \begin{array}{r} 111 = 7_{10} \\ +101 = 5_{10} \\ \hline 1100 = 12_{10} \end{array}$$

Ako se pri zbrajanju dvaju bitova zanemari prijenos, rezultat je zbroj po modulu 2. Ta je operacija u logičkome smislu ekvivalentna ekskluzivnoj disjunkciji (XOR) i može se prikazati tablicom istine (tablica 2.7).

Tablica 2.7: Zbrajanje dvaju bitova po modulu 2

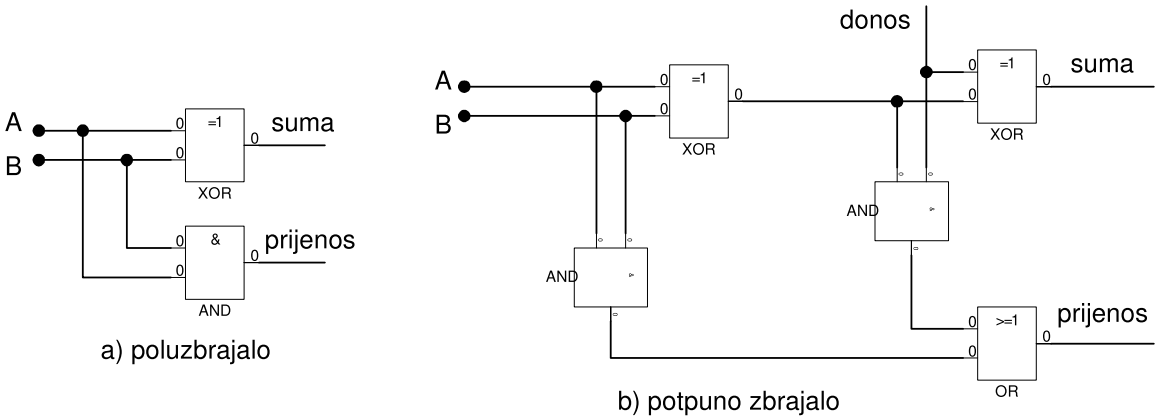
A	B	zbroj $A \oplus B$	prijenos A & B
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Digitalni sklop koji obavlja ovu operaciju naziva se poluzbrajalo (*half adder*) i prikazan je na slici 2.3(a). Pri zbrajanju ostalih bitova treba pribrojiti i bit donosa, pa vrijedi tablica istinitosti 2.8. Digitalni sklop koji realizira ovu funkciju naziva se potpuno zbrajalo i prikazan je na slici 2.3(b). Očito je da se upotrebom više ovakvih sklopova može "izračunati" zbroj dva binarna niza, na način da se "prijenos" sa zbrajala bitova manjega značaja prenosi kao "donos" u zbrajalo bitova većega značaja.

Na prikazu potpunoga zbrajala dva vodoravna voda slijeva predstavljaju ulaze A i B, a okomiti vod odozgo ulazni donos. Brojčane oznake uz vodove prikazuju trenutačne logičke vrijednosti signala.

Tablica 2.8: Tablica istinitosti 1-bitnog zbrajala

A	B	donos	zbroj	prijenos
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Slika 2.3: Sklopovska izvedba 1-bitnog zbrajala: a) poluzbrajalo; b) potpuno zbrajalo

Binarno oduzimanje također se može izvesti slično pravilima dekadskoga sustava, međutim rezultat oduzimanja mogu biti i negativni brojevi. Kako se kodiraju negativni brojevi objašnjeno je u poglavlju tipova podataka (komplement dvojke).

Kod binarnoga množenja za svaku znamenku množitelja zapisuje se jedan djelomični umnožak. Prvi se zapisuje bez pomaka, a svaki sljedeći pomiče se za jedno mjesto ulijevo. Ako je znamenka množitelja 0, djelomični umnožak sastoji se od nula, a ako je 1, jednak je množeniku. Prazna mjesta s desne strane djelomičnih umnožaka prikazuju njihov pomak ulijevo; ona nisu dopisane znamenke množenika. Naprimjer:

$$5 \cdot 5 = 25$$

$$\begin{array}{rrrrrr} & & 1 & 0 & 1 & (5) \\ \times & & 1 & 0 & 1 & (5) \\ \hline & & 1 & 0 & 1 & \\ & 0 & 0 & 0 & & \\ 1 & 0 & 1 & & & \\ \hline 1 & 1 & 0 & 0 & 1 & (25) \end{array}$$

$$5 \cdot 10 = 50$$

$$\begin{array}{rrrrrrr} & & & 1 & 0 & 1 & (5) \\ \times & & 1 & 0 & 1 & 0 & (10) \\ \hline & & & 0 & 0 & 0 & \\ & & 1 & 0 & 1 & & \\ & 0 & 0 & 0 & & & \\ 1 & 0 & 1 & & & & \\ \hline 1 & 1 & 0 & 0 & 1 & 0 & (50) \end{array}$$

Binarno dijeljenje u računalu izvodi se kombinacijom pomicanja bitova i oduzimanja, prema načelu sličnomu pisanom dijeljenju dekadskih brojeva. Procesori tu operaciju mogu ostvarivati sklopovski, mikrokodom ili odgovarajućom strojnom instrukcijom.

2.7.3 Skraćeni izrazi pridruživanja

Postoje sljedeći skraćeni izrazi pridruživanja: `+=`, `-=`, `*=`, `/=`, `%=`, `>>=`, `<<=`, `&=`, `^=` i `|=`.

Sljedeći primjeri pokazuju primjene ovih operatora:

```
a -= 5;          // ekvivalentno izrazu a = a - 5;
a /= b;          // ekvivalentno izrazu a = a / b;
cijena *= kol + 1;// ekvivalentno izrazu cijena = cijena * (kol + 1);
```

2.7.4 Operatori povećavanja i smanjenja

U C++-u postoje operatori povećavanja `++` (*increment*) i smanjenja `--` (*decrement*). Njihov operand mora biti promjenjiva vrijednost aritmetičkoga tipa različitoga od `bool` ili pokazivač na potpuni objektni tip. U sljedećim primjerima povećavaju odnosno smanjuju cjelobrojnu vrijednost za jedan.

```
a++;
a += 1;
a = a + 1;
```

Ako je `a` obična cjelobrojna varijabla i navedeni se izrazi rabe kao samostalne naredbe, sva tri imaju isti učinak: vrijednost varijable `a` uvećava se za jedan. Sami izrazi ipak nisu općenito zamjenjivi. Postfiksni izraz `a++` daje staru vrijednost, dok `a += 1` i `a = a + 1` imaju drukčija pravila pretvorbe za neke tipove. Operatori povećavanja i smanjenja imaju dva oblika:

- prefiksni operator: `++i; --i;`
- postfiksni operator: `i++; i--;`

Rezultat može biti različit zbog semantike prefiksnoga i postfiksnoga oblika. Prefiksni oblik najprije promijeni operand pa daje novu vrijednost, dok postfiksni oblik daje prethodnu vrijednost, uz promjenu operanda. Razlika je prikazana u sljedećim primjerima:

Primjer 1	Primjer 2
`B = 3;`	`B = 3;`
`A = ++B;`	`A = B++;`
`// A je 4, B je 4`	`// A je 3, B je 4`

2.7.5 Relacijski operatori

U C++-u postoje sljedeći relacijski operatori:

Operator	Značenje
==	jednako
!=	različito
>	veće od
<	manje od
>=	veće od ili jednako
<=	manje od ili jednako

```
bool r1{7 == 5}; // false
bool r2{5 > 4};  // true
bool r3{3 != 2}; // true
bool r4{6 >= 6}; // true
bool r5{5 < 5};  // false
```

Sljedeći primjeri pokazuju primjenu ovih operatora. Uz numeričke konstante mogu se upotrebljavati i izrazi koji sadrže varijable, kao što je prikazano u sljedećem primjeru za $a = 2$, $b = 3$ i $c = 6$:

```
int a{2};
int b{3};
int c{6};

bool r1{a == 5};        // false
bool r2{a * b >= c};    // true: 2 * 3 >= 6
bool r3{b + 4 > a * c}; // false: 3 + 4 nije vece od 2 * 6
bool r4{(b = 2) == a};  // true; varijabli b pridruzuje se 2
```

Napomena

Operator s jednim znakom jednakosti (=) operator je pridruživanja, dok je operator s dva znaka jednakosti (==) operator jednakosti. Zato izraz `(b = 2) == a` najprije pridružuje vrijednost 2 varijabli `b`, a tek zatim uspoređuje `b` i `a`. Relacijski operatori u C++-u daju vrijednost tipa **`bool`**: **`true`** ili **`false`**. Pri pretvorbi u cjelobrojni tip te vrijednosti postaju 1 odnosno 0.

2.7.6 Logički operatori

Postoje sljedeći logički operatori:

!	negacija, logički NE operator
&&	logički I operator
||	logički ILI operator

a	b	`a && b`	`a || b`
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Sljedeći primjeri pokazuju primjenu ovih operatora:

```
bool r1{!(10 == 10)};          // false
bool r2{!(8 <= 4)};            // true
bool r3{!true};                // false
bool r4{!false};               // true
bool r5{(5 == 5) && (3 > 6)}; // false
bool r6{(5 == 5) || (3 > 6)}; // true
```

Operatori && i || primjenjuju skraćeno vrednovanje. Desni se operand ne vrednuje ako je rezultat izraza već poznat iz lijevoga operanda. Stoga se u izrazu `uvjet && funkcija()` funkcija ne poziva kada je `uvjet` neistinit, a u izrazu `uvjet || funkcija()` ne poziva se kada je `uvjet` istinit.

2.7.7 Uvjetni operator

Opći oblik uvjetnog operatora je:

```
uvjet ? rezultat1 : rezultat2;
```

Ako je uvjet istinit, vrijednost izraza jest `rezultat1`; u suprotnome je `rezultat2`.

```
int a{2};
int b{3};

int r1{10 == 5 ? 4 : 3}; // 3
int r2{8 == 5 + 3 ? 4 : 3}; // 4
int r3{6 > 3 ? a : b}; // vrijednost varijable a
int r4{a > b ? a : b}; // veća vrijednost između a i b
```

Ovdje se može uočiti sličnost s jednostavnom `if--else` naredbom, te se naravno ona može primijeniti umjesto ovoga operatora.

2.7.8 Bit operatori

Operator	Česta asemblerska oznaka	Opis
&	AND	bit I
|	OR	bit ILI
^	XOR	bit ekskluzivno ILI
~	NOT	jednostruki komplement (inverzija bita)
<<	SHL	pomak bita ulijevo
>>	SHR/SAR	pomak bita udesno

Nazivi asemblerskih instrukcija ovise o arhitekturi procesora. Oznaka `SHR` uobičajeno se odnosi na logički pomak udesno, a `SAR` na aritmetički pomak udesno s proširenjem predznaka. C++ izraz ne mora se prevesti u jednu instrukciju navedenoga imena.

Bit je osnovna jedinica informacije u digitalnom računalu i može imati vrijednost 0 ili 1. Ovisno o kontekstu, te dvije vrijednosti mogu predstavljati neistinu i istinu, isključeno i uključeno ili dva druga međusobno različita stanja.

Nad pojedinim bitovima ili parovima bitova mogu se izvoditi Booleove operacije. Ime potječe od matematičara Georgea Boolea (1815.–1864.), čiji je rad postavio temelje Booleove algebre. Slijedi popis osnovnih Booleovih operacija.

AND

Ova operacija se izvodi između dva različita bita (a i b). Rezultat je 1 ako su i a i b jednaki 1, a ako je bilo koji od njih 0, rezultat je 0.

a	b	$a \,\&\, b$
0	0	0
0	1	0
1	0	0
1	1	1

OR

Ova operacija se izvodi između dva različita bita (a i b). Rezultat je 1 ako je bilo koji od njih, a ili b, jednak 1. Ako nijedan od njih nije 1, rezultat je 0.

a	b	$a \mid b$
0	0	0
0	1	1
1	0	1
1	1	1

XOR (Ekskluzivno ILI)

Ova operacija izvodi se između dva različita bita (a i b). Rezultat je 1 ako je bilo koji od njih, a ili b, jednak 1, osim u slučaju ako su oba 1. Ako su oba ili jednaka 1 ili su oba jednaka 0, rezultat je 0.

Operacija ekskluzivne disjunkcije (XOR) često se koristi u bit operacijama i u programima s bit-mapiranim grafičkim algoritmima. Zanimljivo svojstvo ove operacije jest da, ako se na neki binarni niz A dva puta uzastopno primijeni bit-značajna ekskluzivna disjunkcija s nizom B, rezultatni niz jednak je nizu A. Primjerice, neka je niz $A = 1010$, a niz $B = 0110$. Tada je:

$$A \oplus B = 1100, \qquad (A \oplus B) \oplus B = 1010 = A.$$

Tako se jednostavnom XOR maskom početni niz može promijeniti i zatim vratiti. To svojstvo samo po sebi ne čini sigurnu enkripciju: poznata ili ponavljajuća maska ne pruža kriptografsku zaštitu.

a	b	$a \wedge b$
0	0	0
0	1	1
1	0	1
1	1	0

NOT

Ova se operacija izvodi na jednom bitu. Njezin rezultat jest inverzija trenutačne vrijednosti bita: ako je vrijednost bila 1, prelazi u 0 i obratno.

a	$\sim a$
0	1
1	0

Ovo su četiri osnovne Booleove operacije (AND, OR, XOR i NOT). Kombinacijom tih operacija bilo koji željeni rezultat moguće je postići. U C++-u binarni operatori `&`, `|` i `^` primjenjuju se na dva operanda, a unarni operator `~` na jedan. Operandi moraju biti cjelobrojnoga tipa ili nabrojivoga tipa bez vlastitoga područja važenja (*unscoped enumeration*). Operacija se izvodi nad odgovarajućim bitovima operanada nakon uobičajenih cjelobrojnih promocija i, za binarne operatore, uobičajenih aritmetičkih pretvorbi.

Na primjer, pretpostavimo dvije varijable: `a` i `b`, obje tipa `char`. `a` sadržava vrijednost 195 (11000011 binarno), a `b` sadržava vrijednost 87 (01010111 binarno). Ako napišemo sljedeći kod:

```
unsigned char a = 195;
unsigned char b = 87;
unsigned char c;
c = a & b;
```

to znači da smo izvršili bit operaciju AND između `a` i `b`. Rezultat (sadržaj `c`) bio bi sljedeći. Upotrijebljen je `unsigned char` jer obični **`char`** na nekim platformama ne može prikazati vrijednost 195:

$$\begin{array}{rcl} a & : & 11000011 \\ & & \& \\ b & : & 01010111 \\ \hline c & : & 01000011 \end{array}$$

Binarno 01000011 jednako je dekadski 67.

Operatore pomaka vidjet ćemo kroz sljedeće primjere:

```
unsigned int i = 26;           // donjih 8 bitova: 00011010 (26)
unsigned int lijevo = i << 3; // donjih 8 bitova: 11010000 (208)
unsigned int desno = i >> 3;  // donjih 8 bitova: 00000011 (3)
```

U prikazanim vrijednostima tipa `unsigned int` oslobođena se mjesta popunjavaju nulama. Pomak za tri bita ulijevo odgovara množenju s $2^3 = 8$ ako je rezultat prikaziv tipom ($26 \cdot 8 = 208$), dok pomak udesno odgovara cjelobrojnom dijeljenju s $2^3 = 8$ ($26/8 = 3$). Broj mjesta pomaka ne smije biti negativan ni veći ili jednak širini promoviranoga lijevog operanda. Za tipove s predznakom pravila ovise o vrijednosti i smjeru pomaka. Od standarda C++20 desni pomak negativne vrijednosti zaokružuje rezultat prema negativnoj beskonačnosti, što na uobičajenome prikazu komplementom dvojke odgovara proširenju predznaka; u starijim standardima rezultat je ovisio o implementaciji. Zbog toga su za bit operacije najprikladniji tipovi bez predznaka.

Ciklička rotacija nije C++ operator: zapisi `<<<` i `>>>` nisu valjani. Od standarda C++20 rotacije su dostupne funkcijama `std::rotl` i `std::rotr` iz zaglavlja `<bit>`.

2.7.9 Pretvorbe tipova

U mnogim slučajevima C++ automatski izvodi pretvorbu tipa podataka. Programer može i eksplicitno zadati pretvorbu. U suvremenome C++-u za namjernu pretvorbu prednost imaju imenovani operatori, među kojima je za brojčane pretvorbe najčešći **`static_cast`**:

```
double f{3.14};
int i{static_cast<int>(f)};
```

Varijabla `i` dobiva vrijednost 3 jer se pri pretvorbi realnoga broja u cijeli razlomljeni dio odbacuje prema nuli. C++ je iz jezika C naslijedio i C-stil pretvorbe:

```
int j = (int)f;
```

Postoji i funkcijski zapis:

```
int k = int(f);
```

Imenovani operator jasnije pokazuje namjeru i lakše se pronalazi u izvornome kodu, pa ga treba rabiti u nastavnim primjerima.

2.7.10 Operator `sizeof`

Operator **`sizeof`** primjenjuje se na izraz ili na tip zapisan u zagradama te vraća veličinu odgovarajućega tipa ili objekta u bajtovima. Rezultat je tipa `std::size_t`, definiranoga u zaglavlju `<cstddef>`:

```
std::size_t a{sizeof(char)};
std::size_t b{sizeof(int)};
```

Varijabla `a` uvijek dobiva vrijednost 1 jer standard definira `sizeof(char)` kao 1. Vrijednost varijable `b` ovisi o implementaciji. C++ bajt na uobičajenim platformama ima 8 bitova, ali standard dopušta i drukčiju širinu. Također postoje i drugi operatori, ali će se oni opisati kasnije kada se ukaže potreba za nekima od njih.

2.7.11 Manipulatori tokova

Manipulatori `flush`, `endl` i `ends` djeluju na izlazne tokove, dok `ws` djeluje na ulazne tokove. Manipulator `endl` umeće oznaku novoga retka i zatim prazni izlazni međuspremnik. Za običan prijelaz u novi red dovoljno je ispisati znak `'\n'`; `endl` treba rabiti kada je potrebno i trenutačno pražnjenje međuspremnika. Detaljnija izvedba manipulatora razmatrat će se uz tokove i funkcije.

2.7.12 Prioritet operatora

Sljedeća tablica prikazuje prioritet najčešćih operatora počevši od najvećeg prioriteta. Operatori u istome retku imaju jednak prioritet, a stupac *Pridruživanje* pokazuje redoslijed grupiranja.

Tablica 2.9: Prioritet najčešćih operatora

Prioritet	Operator	Opis	Pridruživanje
1	`::`	područje važenja	nije primjenjivo
2	`() [] -> . ++ --`	poziv, indeksiranje, pristup članu, postfiksni ++/--	lijevo
3	`++ -- + - ! ~ * & sizeof`	unarni operatori	desno
4	`.* ->*`	pristup članu preko pokazivača	lijevo
5	`* / %`	množenje, dijeljenje, ostatak	lijevo
6	`+ -`	zbrajanje i oduzimanje	lijevo
7	`<< >>`	bit pomak	lijevo
8	`< <= > >=`	relacijski operatori	lijevo
9	`== !=`	jednakost i nejednakost	lijevo
10	`&`	bit I	lijevo
11	`^`	bit XOR	lijevo
12	`|`	bit ILI	lijevo
13	`&&`	logički I	lijevo
14	`||`	logički ILI	lijevo
15	`?:`	uvjetni operator	desno
16	`= += -= *= /= %= >>= <<= &= ^= |=`	pridruživanje	desno
17	`,`	operator zareza	lijevo

Napomena

Ako nismo sigurni u prioritet nekog od operatora, najbolje je upotrijebiti zagrade `()` kao i u matematici.

Zapamtite

Tip podatka određuje skup vrijednosti i dopuštenih operacija. Varijable treba inicijalizirati pri deklaraciji, za nepromjenjive vrijednosti prednost dati riječima `const` i `constexpr`, a eksplicitne pretvorbe pisati imenovanim C++ operatorima poput `static_cast`. Zagrade su poželjne kada njima izraz postaje jasniji, čak i kada poznajemo prioritet operatora.

Kratki sažetak

Tip podatka određuje kako se vrijednost prikazuje u memoriji, koji joj je raspon i koje se operacije nad njom mogu izvoditi. Varijable i konstante treba deklarirati odgovarajućim tipom i, kad god je moguće, odmah inicijalizirati. Inicijalizacija nije isto što i naknadno pridruživanje, a područje važenja određuje gdje se neko ime može rabiti. Pretvorbe tipova mogu biti implicitne ili eksplicitne, pri čemu inicijalizacija vitičastim zagradama sprječava neke gubitke podataka. Aritmetički, relacijski, logički i bit operatori obrađuju vrijednosti prema pravilima prioriteta i pridruživanja; zagrade treba rabiti kada njima značenje izraza postaje jasnije.

Pitanja za samoprovjeru

1. Koje informacije o podatku određuje njegov tip?
2. Zašto veličinu tipa `int` ne treba unaprijed pretpostaviti i kako je možemo provjeriti?
3. Koja je razlika između deklaracije, inicijalizacije i pridruživanja vrijednosti?
4. Što u suvremenome C++-u znači ključna riječ `auto`? Napišite valjan primjer.
5. Zašto se za simboličke konstante preporučuju `constexpr` i `const` umjesto makroa `#define`?
6. Koja je razlika između operatora = i ==?
7. Koji je rezultat izraza `11 % 3` i što taj operator računa?
8. Objasnite razliku između prefiksnog i postfiksnog oblika operatora ++.
9. Što se događa s lokalnom varijablom deklariranom riječju `static` između dvaju poziva funkcije?
10. Stupčano izračunajte binarni umnožak $(101)_2 \cdot (110)_2$ i objasnite pomicanje djelomičnih umnožaka.
11. Zašto operatori <<< i >>> nisu valjani C++ operatori i kako se u C++20 izvodi ciklička rotacija bitova?
12. Što se događa pri izrazu `static_cast<int>(3.99)`?
13. Izračunajte vrijednost izraza `2 + 3 * 4` i zatim `(2 + 3) * 4`.
14. Zašto neposredna usporedba realnih rezultata operatorom == može biti nepouzdana?

3. Kontrola toka programa

Što ćete naučiti?

Nakon ovoga poglavlja moći ćete primijeniti uvjetne naredbe i programske petlje te odabrati odgovarajući oblik grananja. Također ćete pravilno upotrebljavati naredbe **`break`** i **`continue`**.

U ovom poglavlju bit će obrađene naredbe koje kontroliraju tok izvođenja programa.

3.1 Naredba `if-else`

Naredba se upotrebljava za izvršavanje naredbe ili bloka naredbi samo ako je uvjet (engl. *condition*) zadovoljen. To znači da se ova naredba upotrebljava za izražavanje odluke. Sintaksa je sljedeća:

```
if (uvjet)
{
    naredba;
}
```

Primjerice, sljedeći dio programa ispisuje: "x je 100", samo ako je vrijednost 100 spremljena u varijablu x.

```
if (x == 100)
{
    cout << "x je 100";
}
```

Ako se želi izvršiti više od jedne naredbe u slučaju da je uvjet zadovoljen, tada se blok naredbi piše unutar vitičastih zagrada { }:

```
if (x == 100)
{
    cout << "x je ";
    cout << x;
}
```

Ako se želi izvršiti neka naredba ili blok naredbi u slučaju da uvjet nije zadovoljen, tada se upotrebljava ključna riječ **`else`**:

```
if (uvjet)
{
    naredba1;
}
else
{
    naredba2;
```

```
}
```

Primjerice:

```
if (x == 100)
{
    cout << "x je 100";
}
else
{
    cout << "x nije 100";
}
```

dio programa ispisuje x je 100 ako je vrijednost 100 spremljena u varijablu x, a ako nije ispisuje x nije 100.

Za izvršavanje različitih blokova naredbi u ovisnosti o nizu različitih uvjeta možemo upotrijebiti strukturu višestrukoga uvjetnog grananja. To je prikazano u sljedećem primjeru, u kojem se provjerava je li varijabla x pozitivna, negativna ili jednaka nuli:

```
if (x > 0)
{
    cout << "x je pozitivan";
}
else if (x < 0)
{
    cout << "x je negativan";
}
else
{
    cout << "x je 0";
}
```

Napomena

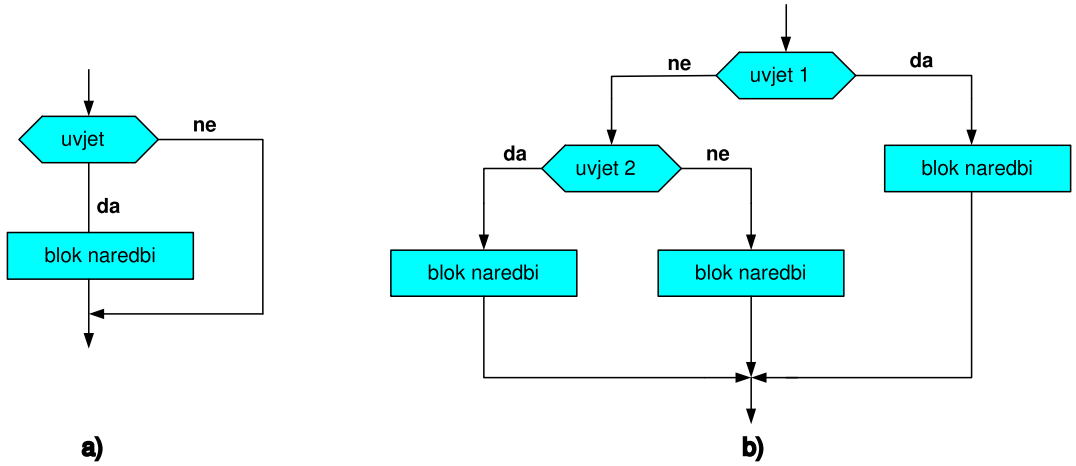
C++ dopušta izostavljanje vitičastih zagrada kada grana sadrži samo jednu naredbu. U ovoj ćemo ih skripti ipak dosljedno pisati jer se time smanjuje mogućnost pogreške pri naknadnome dodavanju naredbi.

Programski jezik C++ dopušta gniježđenje naredbi **if**, što je prikazano u sljedećem primjeru:

```
if (a != b)
{
    if (a == c)
    {
        cout << "\n a je razlicito od b i jednako c";
    }
    else
    {
        cout << "\n a je razlicito od b i od c \n";
```

```
    }
}
```

Blok-dijagrami grananja prikazani su na slici 3.1.



Slika 3.1: Blok-dijagrami: a) jednostruko i b) višestruko grananje

3.2 Naredba `switch`

Naredba **`switch`** prikladna je za višestruki odabir na temelju vrijednosti izraza cjelobrojnoga ili pobrojenoga tipa. Iza ključne riječi **`case`** mora se navesti konstantna vrijednost poznata tijekom prevođenja programa, primjerice cijeli broj, znak ili imenovana konstanta. Obična varijabla ne može se rabiti kao vrijednost uz **`case`**.

Sintaksa naredbe **`switch`** jest sljedeća:

```
switch (izraz)
{
    case konstantni_izraz1:
        blok_naredbi1;
        break;
    case konstantni_izraz2:
        blok_naredbi2;
        break;
    // ...
    default:
        blok_naredbi;
}
```

Ako vrijednost izraza odgovara jednom od konstantnih izraza iza ključne riječi **`case`**, tada se izvodi pripadajući blok naredbi. Ovaj blok naredbi najčešće završava naredbom **`break`**. Ako se ona izostavi, izvođenje se nastavlja u sljedećoj grani (*fallthrough*). Neobvezna grana **`default`** izvodi se ako vrijednost izraza ne odgovara nijednoj oznaci **`case`**. Može se nalaziti na bilo kojem mjestu unutar naredbe **`switch`**,

premda se uobičajeno piše posljednja.

Sljedeći primjer prikazuje primjenu naredbe **switch** i njezin ekvivalent ostvaren naredbom **if–else**.

```
switch (x)
{
    case 1:
        cout << "x je 1";
        break;
    case 2:
        cout << "x je 2";
        break;
    default:
        cout << "vrijednost x je nepoznata";
}
```

```
if (x == 1)
{
    cout << "x je 1";
}
else if (x == 2)
{
    cout << "x je 2";
}
else
{
    cout << "vrijednost x je nepoznata";
}
```

Također možemo izvršiti isti blok naredbi za različite vrijednosti izraza kao u sljedećem primjeru:

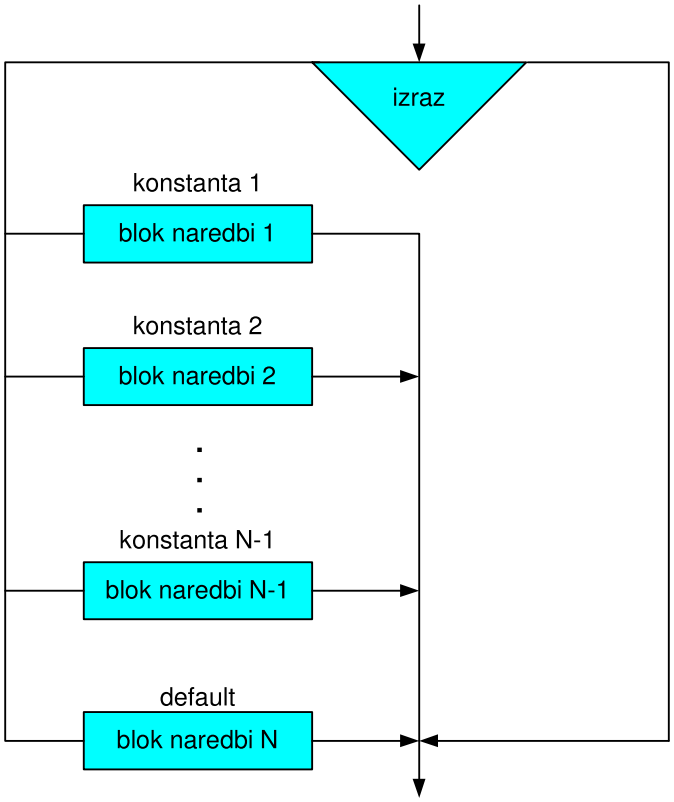
```
switch (x)
{
    case 1:
    case 2:
    case 3:
        cout << "x je 1, 2 ili 3";
        break;
    default:
        cout << "x nije 1, 2 niti 3";
}
```

Napomena

U slučaju kada višeznačnu odluku nije moguće temeljiti na konstantnim vrijednostima, mora se primijeniti naredba **if–else**. Sljedeće oznake **case** nisu dopuštene:

```
case (n * 2):   // n nije konstantni izraz
case (1 ... 3): // raspon nije dio C++ sintakse
```

Tok izvođenja prikazan je na slici 3.2.



Slika 3.2: Blok-dijagram naredbe `switch`

3.3 Programske petlje

Programske se petlje upotrebljavaju za višestruka ponavljanja blokova naredbi. S obzirom na mjesto ispitivanja uvjeta ponavljanja postoje dvije skupine programskih petlji. U prvu se skupinu ubrajaju petlje s ispitivanjem uvjeta ponavljanja na početku bloka naredbi (**`for`** i **`while`**), dok se u drugu skupinu ubrajaju petlje s ispitivanjem uvjeta ponavljanja na kraju bloka naredbi (**`do`**–**`while`**).

3.3.1 Petlja `while`

Sintaksa petlje **`while`** jest sljedeća:

```
while (izraz)
{
    naredba;
}
```

i njezina je funkcija ponavljanje naredbe dok je uvjet istinit (*true*).

Razmotrimo primjenu petlje **while** na sljedećem primjeru:

```
#include <iostream>

using namespace std;

int main()
{
    int n;

    cout << "Unesi pocetnu vrijednost > ";
    cin >> n;

    while (n > 0)
    {
        cout << n << ", ";
        --n;
    }

    cout << "KRAJ!";
    return 0;
}
```

Output

```
Unesi pocetnu vrijednost > 8
8, 7, 6, 5, 4, 3, 2, 1, KRAJ!
```

Ovaj se primjer može objasniti u sljedećih pet koraka:

1. Korisnik postavlja početnu vrijednost brojača (varijabli n pridružuje se vrijednost).
2. Naredba **while** provjerava je li `n > 0`. Ako je uvjet istinit, izvršava se korak 3; ako je neistinit, program nastavlja korakom 5.
3. Izvršavaju se naredbe:

   ```
   cout << n << ", ";
   --n;
   ```

   Ispisuje se vrijednost varijable n, a zatim se n umanjuje za 1.
4. Na kraju bloka program se vraća na korak 2.
5. Program se nastavlja iza bloka naredbi: ispisuje se KRAJ!, a zatim program završava.

Petlja **while** izvršava se sve dok je izraz istinit. Izraz obično sadrži relacijske i logičke operatore koji kao rezultat daju vrijednost istine (true, vrijednost različita od 0) ili laži (false, 0). Ako je odmah na početku u petlji izraz neistinit, blok naredbi unutar petlje se uopće neće izvršiti.

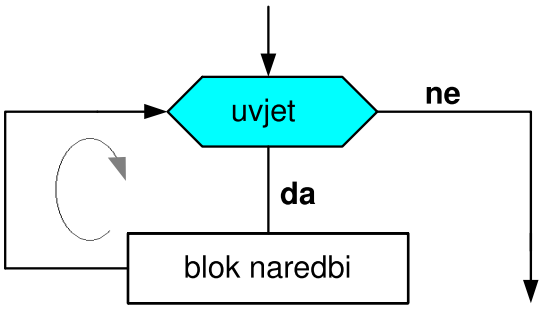
Kako izraz određuje trenutak kada će izvođenje petlje prestati, algoritam unutar bloka naredbi petlje **while** trebao bi mijenjati vrijednost varijable koja se nalazi u izrazu. Ako ovo nije zadovoljeno, petlja **while** postat će beskonačna. U slučaju da se ovo dogodi, bez namjere programera, tada će se morati zaustaviti izvođenje programa na način koji je uobičajen za odgovarajući operacijski sustav (kombinacije tipki CTRL+BREAK, CTRL+c i sl.).

Ponekad se javlja potreba za pisanjem beskonačne petlje i to se može izvesti na sljedeći način:

```
while (true)
{
    // naredbe
}
```

Takva se petlja ponavlja sve dok se iz nje izričito ne izađe. U njezinu tijelu zato treba predvidjeti uvjet izlaska i, primjerice, naredbu **break** kojom se petlja prekida kada je taj uvjet zadovoljen.

Tok petlje **while** prikazan je na slici 3.3.



Slika 3.3: Blok-dijagram petlje `while`

3.3.2 Petlja do–while

Petlja **do–while** slična je petlji **while**, ali se uvjet provjerava na kraju petlje. To osigurava izvođenje petlje barem jednom. Sintaksa je sljedeća:

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

Sljedeći primjer pokazuje primjenu ove petlje.

```
#include <iostream>

using namespace std;

int main()
{
    unsigned long n;
```

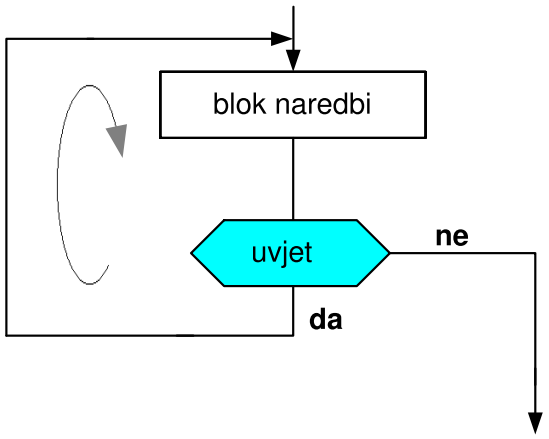
```
8
9       do
10      {
11          cout << "Unesi broj (0 za kraj): ";
12          cin >> n;
13          cout << "Unijeli ste: " << n << "\n";
14      } while (n != 0);
15
16      return 0;
17  }
```

Output

```
Unesi broj (0 za kraj): 5689
Unijeli ste: 5689
Unesi broj (0 za kraj): 3333
Unijeli ste: 3333
Unesi broj (0 za kraj): 0
Unijeli ste: 0
```

Petlja **do–while** obično se upotrebljava kada je uvjet koji određuje kraj izvršavanja naredbi unutar petlje definiran unutar algoritma u samoj petlji. Tako u prethodnom primjeru korisnički ulaz unutar bloka naredbi određuje kraj izvršavanja petlje. Ako korisnik nikada ne unese vrijednost 0 petlja nikada neće završiti.

Tok petlje **do–while** prikazan je na slici 3.4.



Slika 3.4: Blok-dijagram petlje do–while

3.3.3 Petlja for

Petlja **for** objedinjuje inicijalizaciju, uvjet nastavka i izraz koji se izvodi nakon svake iteracije. Često se rabi kada je broj ponavljanja unaprijed poznat, ali to nije njezino ograničenje.

Sintaksa petlje **for** jest:

```
for (inicijalizacija; uvjet; promjena_vrijednosti)
{
    naredba;
}
```

gdje je inicijalizacija dio u kojem se postavlja početna vrijednost kontrolne varijable petlje, uvjet je uvjet koji kontrolna varijabla petlje mora zadovoljiti da bi se izvršio blok naredbi, a promjena_vrijednosti je dio u kojem se najčešće mijenja vrijednost kontrolne varijable. Kontrolna varijabla utječe na uvjet nastavka petlje, a često služi i kao njezin indeks. Dijelovi petlje mogu se izostaviti, ali dvije točke sa zarezom u zaglavlju ostaju obvezne. Izostavljeni uvjet smatra se istinitim.

Nastavak izvođenja petlje **for** ispituje se na početku petlje. Ako je uvjet neistinit odmah kada petlja počinje, ona se neće nijednom izvršiti. Petljom:

```
for (;;)
{
    // naredbe
}
```

ostvaruje se beskonačna petlja ili ponavljanje operacija dok se ne ispuni određeni uvjet u tijelu petlje. U ovom slučaju se petlja obično prekida naredbom **break**.

Rad petlje **for** analizirat će se na sljedećem primjeru.

```
#include <iostream>

using namespace std;

int main()
{
    for (int n = 10; n >= 0; --n)
    {
        cout << n << ", ";
    }

    cout << "KRAJ!\n";
    return 0;
}
```

Output

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, KRAJ!
```

Ovaj program radi na sljedeći način:

1. deklarira se kontrolna varijabla n i inicijalizira vrijednošću 10;

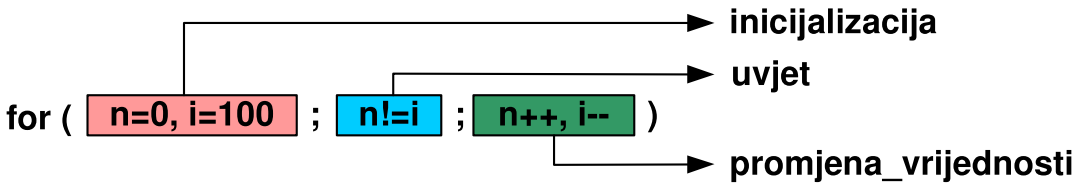
2. provjerava se uvjet; ako je istinit, nastavlja se izvođenje petlje, a inače petlja završava;
3. ispisuje se vrijednost varijable `n`;
4. mijenja se vrijednost kontrolne varijable (umanjuje se za 1), a program se ponovno vraća na korak 2.

Također se može upotrijebiti operator zareza za navođenje više od jednoga izraza u prvom ili trećem dijelu petlje **`for`**. To je prikazano u sljedećem primjeru:

```
for (n = 0, i = 100; n != i; n++, i--)
{
    // naredbe
}
```

Ova će se petlja izvršiti 50 puta ako se `n` i `i` ne mijenjaju unutar njezina tijela.

Dijelovi zaglavlja petlje **`for`** prikazani su na slici 3.5.



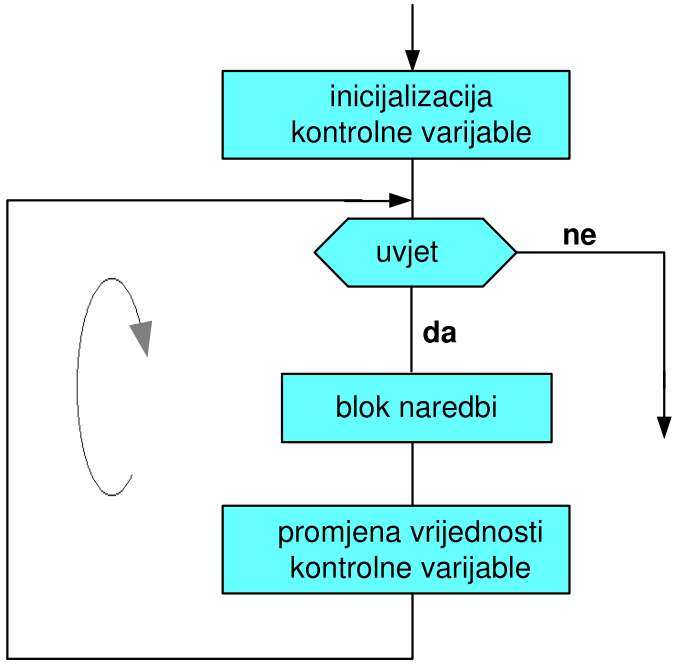
Slika 3.5: Dijelovi zaglavlja petlje `for`

Varijabla `n` počinje vrijednošću 0, a `i` vrijednošću 100. Uvjet `n != i` zadovoljen je sve dok se njihove vrijednosti ne izjednače. Budući da se `n` povećava, a `i` smanjuje za 1, uvjet će postati neistinit nakon 50. izvršenja petlje, kada obje varijable poprime vrijednost 50.

Napomena

Programski jezik C++ dopušta gniježđenje programskih petlji. To znači da se unutar jedne petlje može nalaziti druga petlja.

Opći tok petlje **`for`** prikazan je na slici 3.6.



Slika 3.6: Blok-dijagram petlje `for`

3.4 Naredba `break`

Naredba **`break`** osigurava prekid izvođenja i izlaz iz petlje (**`for`**, **`while`** i **`do–while`**) ili naredbe **`switch`**. Primjena ove naredbe dana je u sljedećem primjeru.

```
#include <iostream>

using namespace std;

int main()
{
    for (int n = 10; n > 0; --n)
    {
        cout << n << ", ";

        if (n == 3)
        {
            cout << "zaustavi brojac!\n";
            break;
        }
    }

    return 0;
}
```

Output

```
10, 9, 8, 7, 6, 5, 4, 3, zaustavi brojac!
```

3.5 Naredba `continue`

Naredba **`continue`** upotrebljava se za prekid tekuće iteracije petlje i skok na početak sljedeće iteracije. Kod petlje **`for`** kontrola toka programa prebacuje se na izraz za promjenu vrijednosti, dok se kod petlji **`while`** i **`do–while`** prebacuje na izraz uvjeta petlje.

```
#include <iostream>

using namespace std;

int main()
{
    for (int n = 10; n > 0; --n)
    {
        if (n == 5)
        {
            continue; // preskoči broj 5
        }

        cout << n << ", ";
    }

    cout << "KRAJ!";
    return 0;
}
```

Output

```
10, 9, 8, 7, 6, 4, 3, 2, 1, KRAJ!
```

3.6 Naredba `goto`

Naredba **`goto`** jest naredba bezuvjetnoga skoka. Izvršavanjem ove naredbe program se grana na zadani programski redak. Sintaksa je sljedeća:

```
goto oznaka;
```

gdje je `oznaka` simbolička oznaka programskoga retka. Simbolička oznaka definirana je identifikatorom sa znakom dvotočke.

Napomena

Prilikom uporabe ove naredbe mora se jako paziti da se ne izgubi kontrola toka programa. Također kod pretjerane uporabe smanjuje se razumljivost i preglednost izvornog koda. Zbog svega ovoga ima programera koji smatraju da se ova naredba ne bi trebala upotrebljavati. Kako je C++ programski jezik više razine, program se gotovo uvijek može jasnije napisati bez ove naredbe. U posebnim slučajevima ona može pojednostavniti izlaz iz više ugniježđenih petlji, ali prije nje treba razmotriti izdvajanje koda u funkciju i izlaz naredbom `return`.

Sljedeći primjer ilustrira primjenu ove naredbe.

```
#include <iostream>

using namespace std;

int main()
{
    int n = 10;

ponovi:
    cout << n << ", ";
    --n;

    if (n > 0)
    {
        goto ponovi;
    }

    cout << "KRAJ!";
    return 0;
}
```

Output

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, KRAJ!
```

Zapamtite

Iako C++ dopušta izostavljanje vitičastih zagrada kada grana ili tijelo petlje sadrži samo jednu naredbu, njihova dosljedna uporaba smanjuje mogućnost pogreške pri kasnijemu dodavanju naredbi.

Kratki sažetak

Kontrola toka određuje redoslijed izvršavanja programskih naredbi. Naredbe `if-else` i `switch` omogućuju grananje, dok petlje `while`, `do-while` i `for` ponavljaju naredbe dok vrijedi zadani uvjet.

Naredba `break` prekida najbližu petlju ili naredbu `switch`, a `continue` preskače ostatak tekuće iteracije. Naredba `goto` dopušta bezuvjetni skok, ali se u pravilu izbjegava jer strukturirane naredbe daju pregledniji program.

Pitanja za samoprovjeru

1. Kada se upotrebljava `if`, a kada `switch`?
2. Koja je razlika između petlji `while` i `do-while`?
3. Što rade `break` i `continue`?
4. Zašto se `goto` u pravilu izbjegava?

4. Funkcije

Što ćete naučiti?

Nakon ovoga poglavlja moći ćete deklarirati, definirati i pozvati funkciju, objasniti ulogu funkcije `main`, razlikovati prijenos po vrijednosti i po referenci te primijeniti prototipove, zadane vrijednosti parametara, preopterećivanje i rekurziju.

Složeniji programski zadaci zahtijevaju stotine i tisuće redaka izvornoga koda. Pritom se isti postupak često primjenjuje na različitim mjestima u programu. Takav je dio koda dobro izdvojiti u zasebnu logičku cjelinu, dodijeliti joj ime i pozvati je svaki put kada je potrebna. U C++-u se takve cjeline oblikuju kao *funkcije*.

Najvažnije su prednosti uporabe funkcija:

1. izbjegavanje ponavljanja istoga koda,
2. bolja preglednost i lakše razumijevanje programa,
3. jednostavnije ispitivanje pojedinih dijelova programa i
4. lakše pronalaženje i otklanjanje pogrešaka.

Funkcija se može zamisliti kao *crna kutija* (*black box*): pozivatelj mora znati koje podatke funkciji treba predati, što funkcija radi i koji rezultat daje, ali ne mora pri svakome pozivu poznavati pojedinosti njezine izvedbe.

Podaci se funkciji predaju argumentima, a funkcija ih prima preko svojih parametara. *Parametri* su imena navedena u deklaraciji i definiciji funkcije, dok su *argumenti* vrijednosti ili izrazi navedeni u njezinu pozivu. U svakodnevnom se govoru pojmovi *parametar* i *argument* često zamjenjuju, ali ih je u stručnome tekstu potrebno razlikovati.

U sljedećem su primjeru `a` i `b` parametri funkcije, dok su `5` i `3` argumenti navedeni pri njezinu pozivu:

```
1 int zbroji(int a, int b)
2 {
3     return a + b;
4 }
5
6 int rezultat{zbroji(5, 3)};
```

Funkcija može vratiti jednu vrijednost naredbom **`return`**. Za dobivanje više rezultata mogu se upotrijebiti parametri koji se prenose po referenci ili složeni povratni tip. Funkcija koja ne vraća vrijednost ima povratni tip **`void`**. Definicija jedne funkcije ne može se nalaziti unutar definicije druge funkcije.

4.1 Definicija i poziv funkcije

Opći oblik definicije funkcije koja vraća vrijednost jest:

```
povratni_tip ime_funkcije(tip1 parametar1, tip2 parametar2)
{
    naredbe;
    return vrijednost;
}
```

U općem obliku:

- *povratni tip* određuje tip vrijednosti koju funkcija vraća;
- *ime funkcije* identifikator je kojim se funkcija poziva;
- *popis parametara* određuje broj, redoslijed i tipove podataka koje funkcija prima; pojedini parametri odvajaju se zarezom;
- naredbe između vitičastih zagrada čine *tijelo funkcije*.

Vrijednost koju funkcija vraća mora se moći pretvoriti u njezin povratni tip. Naredbom **`return`** završava izvođenje funkcije i kontrola se vraća na mjesto njezina poziva.

Funkcija koja ne vraća vrijednost ima povratni tip **`void`**. U njezinoj definiciji nema naredbe **`return`** s vrijednošću:

```
void ime_funkcije(tip1 parametar1, tip2 parametar2)
{
    naredbe;
}
```

Naredba **`return`**`;` bez vrijednosti može se upotrijebiti ako izvođenje funkcije tipa **`void`** treba završiti prije završne vitičaste zagrade.

4.2 Funkcija `main`

Svaki C++ program koji se izvodi u uobičajenome okruženju ima točno jednu funkciju `main`. Od nje počinje izvođenje programa, a njezin je povratni tip **`int`**. Najjednostavniji je oblik funkcije `main` bez parametara:

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

Vrijednost `0` označava uspješan završetak programa. Dolazak do završne vitičaste zagrade funkcije `main` također se tumači kao **`return`** `0;`, ali je u početnim primjerima korisno povratnu naredbu pisati izričito.

Funkcija `main` može primati i argumente naredbenoga retka:

```
int main(int argc, char* argv[])
```

Parametar `argc` sadržava broj argumenata naredbenoga retka, uključujući ime kojim je program pokrenut. Parametar `argv` polje je pokazivača na znakovne nizove: `argv[0]` uobičajeno sadržava ime programa, a `argv[1]` prvi argument koji je korisnik predao programu. Prije pristupa pojedinome elementu potrebno je provjeriti je li predan odgovarajući broj argumenata.

Sljedeći program ispisuje svoje ime i sve argumente naredbenoga retka:

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

int main(int argc, char* argv[])
{
    if (argc > 0)
    {
        cout << "Ime programa: " << argv[0] << '\n';
    }

    for (int i = 1; i < argc; ++i)
    {
        cout << "Argument " << i << ": " << argv[i] << '\n';
    }

    return 0;
}
```

Ako se program pod imenom `program` pokrene naredbom `program Split 2026`, rezultat je:

Output

```
Ime programa: program
Argument 1: Split
Argument 2: 2026
```

Globalne se varijable deklariraju izvan svih funkcija, najčešće prije funkcije `main`, kako je objašnjeno u poglavlju 2. Njihovu uporabu treba ograničiti na situacije u kojima su doista potrebne.

4.3 Prototipovi funkcija (*function prototypes*)

Funkcija mora biti deklarirana prije mjesta na kojemu se poziva. Ako se njezina potpuna definicija nalazi poslije funkcije `main` ili u drugoj izvornoj datoteci, prije poziva navodi se *prototip funkcije*.

Prototip je deklaracija koja opisuje sučelje funkcije. Mora sadržavati:

- povratni tip funkcije,

- ime funkcije,
- broj, redoslijed i tipove njezinih parametara te
- točku sa zarezom na kraju.

Imena parametara u prototipu nisu obvezna, ali ih je korisno navesti ako pojašnjavaju značenje pojedinoga parametra. Prototip nema tijelo funkcije:

```
int zbroj(int a, int b);
double sredina(double, double);
void ispisi_poruku();
```

Na temelju prototipa prevoditelj već na mjestu poziva može provjeriti broj i tipove argumenata te pravilno obraditi povratnu vrijednost. Prototip zato nije samo sredstvo za urednije slaganje koda: on omogućuje odvojenu deklaraciju i definiciju funkcije, provjeru ispravnosti poziva te organiziranje većega programa u više izvornih datoteka. Prototipovi funkcija koje trebaju biti dostupne u više izvornih datoteka najčešće se smještaju u zaglavlja.

U sljedećem programu prototip omogućuje poziv funkcije `ispisi_parnost` iz funkcije `main` iako je njezina definicija napisana poslije funkcije `main`:

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

void ispisi_parnost(int broj);

int main()
{
    int i;

    do
    {
        cout << "Unesi broj (0 za izlaz): ";
        cin >> i;

        if (i != 0)
        {
            ispisi_parnost(i);
        }
    }
    while (i != 0);

    return 0;
}

void ispisi_parnost(int broj)
{
    if (broj % 2 == 0)
    {
        cout << "Broj je paran.\n";
    }
```

```
    else
    {
        cout << "Broj je neparan.\n";
    }
}
```

Output

```
Unesi broj (0 za izlaz): 6
Broj je paran.
Unesi broj (0 za izlaz): 17
Broj je neparan.
Unesi broj (0 za izlaz): 0
```

4.4 Prijenos argumenta po vrijednosti

Kada se argument prenosi po vrijednosti, funkcija dobiva kopiju njegove vrijednosti. Promjena parametra unutar funkcije utječe samo na tu kopiju i ne mijenja varijablu navedenu kao argument pri pozivu funkcije.

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

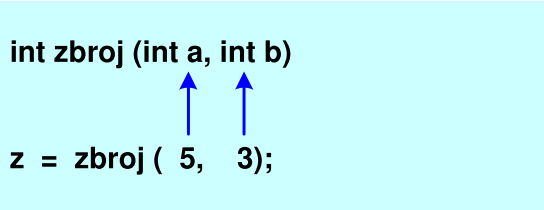
int zbroj(int a, int b)
{
    int r; /* lokalna varijabla */
    r = a + b;
    return r; /* vraća izračunanu vrijednost */
}

int main()
{
    int z;
    z = zbroj(5, 3); /* poziv funkcije */
    cout << "Rezultat je " << z << ".\n";
    return 0;
}
```

Output

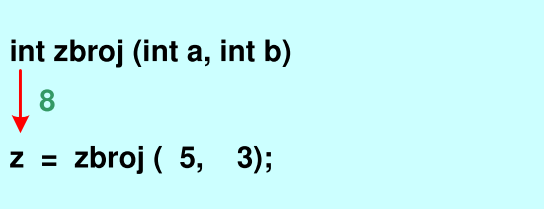
```
Rezultat je 8.
```

Program se počinje izvoditi ulaskom u funkciju `main`. Nakon deklaracije varijable `z` slijedi poziv `zbroj(5, 3)`. Argumenti `5` i `3` kopiraju se u parametre `a` i `b`, što je prikazano na slici 4.1.



Slika 4.1: Prijenos argumenta po vrijednosti

Funkcija izračuna zbroj te naredbom `return r;` vraća vrijednost `8`. Prijenos te vrijednosti i njezino pridruživanje varijabli `z` u funkciji `main` prikazani su na slici 4.2.



Slika 4.2: Pridruživanje povratne vrijednosti varijabli

Parametri `a` i `b` te varijabla `r` lokalni su funkciji `zbroj`. Varijabla `z` lokalna je funkciji `main`. Opća pravila područja važenja lokalnih i globalnih imena već su obrađena u poglavlju 2; ovdje je važno uočiti da svaki poziv funkcije dobiva vlastite parametre.

Argument funkcije može biti literal, varijabla ili složeniji izraz. Povratna vrijednost funkcije može se pridružiti varijabli, neposredno ispisati ili upotrijebiti kao dio drugoga izraza:

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

int razlika(int a, int b)
{
    return a - b;
}

int main()
{
    int x = 5;
    int y = 3;
    int z;

    z = razlika(7, 2);
    cout << "Prvi rezultat je " << z << '\n';
    cout << "Drugi rezultat je " << razlika(7, 2) << '\n';
    cout << "Treći rezultat je "
         << razlika(x, y) << '\n';

    z = 4 + razlika(x, y);
    cout << "Četvrti rezultat je "
```

```
23          << z << '\n';
24      return 0;
25  }
```

Output

```
Prvi rezultat je 5
Drugi rezultat je 5
Treći rezultat je 2
Četvrti rezultat je 6
```

4.4.1 Funkcije bez povratne vrijednosti

Ako funkcija ne vraća vrijednost, njezin je povratni tip **void**. Prazan popis parametara u C++-u piše se kao `()`. Pojmovi *funkcija bez povratne vrijednosti* i *funkcija bez parametara* nisu istoznačni: funkcija tipa **void** može imati parametre, a funkcija koja vraća vrijednost može imati prazan popis parametara.

```
1  #include <iostream>
2  using namespace std;
3
4  void ispisi()
5  {
6      cout << "Ja sam funkcija!\n";
7  }
8
9  int main()
10 {
11     ispisi();
12     return 0;
13 }
```

Output

```
Ja sam funkcija!
```

Funkcija `main` ne smije imati povratni tip **void**.

4.5 Prijenos argumenta po referenci

Kada se argument prenosi po referenci, parametar postaje drugo ime za varijablu navedenu kao argument. Promjena parametra unutar funkcije zato izravno mijenja tu varijablu.

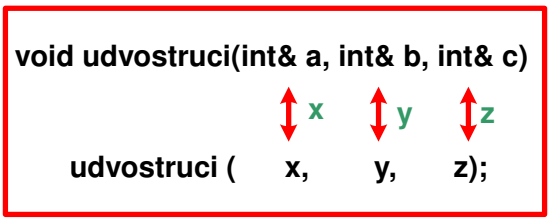
```
1  #include <iostream>
2  using namespace std;
```

```
3
4  void udvostruci(int& a, int& b, int& c)
5  {
6      a *= 2;
7      b *= 2;
8      c *= 2;
9  }
10
11 int main()
12 {
13     int x = 1;
14     int y = 3;
15     int z = 7;
16
17     udvostruci(x, y, z);
18     cout << "x=" << x << ", y=" << y << ", z=" << z << '\n';
19     return 0;
20 }
```

Output

```
x=2, y=6, z=14
```

Znak `&` uz tip parametra označava da je parametar referenca na varijablu navedenu kao argument, a ne kopija njezine vrijednosti. U prethodnome primjeru parametri `a`, `b` i `c` odnose se na objekte `x`, `y` i `z` iz funkcije `main`. Njihov je odnos prikazan na slici 4.3.



Slika 4.3: Prijenos argumenta po referenci

Prijenos argumenta po referenci svojstvo je C++-a. U jeziku C isti bi se učinak ostvario pokazivačima, koji će biti obrađeni u kasnijem poglavlju.

Prijenos po referenci može se upotrijebiti kada funkcija treba promijeniti više objekata pozivatelja i tako dati više rezultata:

```
1  #include <iostream>
2  using namespace std;
3
4  void prethodni_i_sljedeci(int x, int& prethodni, int& sljedeci)
5  {
6      prethodni = x - 1;
```

```
    sljedeci = x + 1;
}

int main()
{
    int broj = 30;
    int prethodni;
    int sljedeci;

    prethodni_i_sljedeci(broj, prethodni, sljedeci);
    cout << "Prethodni=" << prethodni
         << ", Sljedeći="
         << sljedeci << '\n';
    return 0;
}
```

Output

```
Prethodni=29, Sljedeći=31
```

Napomena

Ako funkcija veći objekt treba samo čitati, uobičajeno se upotrebljava parametar oblika `const T&`. Time se izbjegava kopiranje, a funkciji se onemogućuje mijenjanje izvornoga objekta. Za male aritmetičke tipove prijenos po vrijednosti ostaje jednostavan i prikladan.

4.6 Zadane (*default*) vrijednosti parametara

Parametru se u deklaraciji funkcije može pridružiti zadana (*default*) vrijednost. Ona se upotrebljava kada se u pozivu izostavi odgovarajući argument. Izostavljati se mogu samo argumenti s kraja popisa. Zato, nakon prvoga parametra sa zadanom vrijednošću, svi parametri koji slijede u istoj deklaraciji također moraju imati zadane vrijednosti.

Kada se prototip navodi odvojeno od definicije, zadane se vrijednosti uobičajeno pišu u prototipu, gdje su vidljive na mjestu poziva, i ne ponavljaju se u definiciji:

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

int dijeljenje(int a, int b = 2);

int main()
{
    cout << dijeljenje(12) << '\n';
    cout << dijeljenje(20, 4) << '\n';
    return 0;
}
```

```
12
13 int dijeljenje(int a, int b)
14 {
15     return a / b;
16 }
```

Output

```
6
5
```

U pozivu `dijeljenje(12)` izostavljen je drugi argument, pa parametar `b` dobiva zadanu vrijednost `2`. U pozivu `dijeljenje(20, 4)` drugi je argument izričito naveden, pa parametar `b` dobiva vrijednost `4`. Funkcija provodi cjelobrojno dijeljenje jer su oba parametra tipa **`int`**.

4.7 Preopterećivanje (*overloading*) funkcija

Više funkcija može imati isto ime ako se njihovi popisi parametara razlikuju po broju parametara ili po njihovim tipovima. Povratni tip sam po sebi nije dovoljan za razlikovanje funkcija. Funkcije istoga imena s različitim popisima parametara nazivaju se *preopterećene funkcije* (*overloaded functions*), a ovo se svojstvo u C++ jeziku naziva *preopterećivanje funkcija* (*function overloading*).

Pri pozivu prevoditelj uspoređuje predane argumente s dostupnim verzijama funkcije i odabire onu koja im najbolje odgovara.

```
1  #include <iostream>
2  using namespace std;
3
4  int dijeljenje(int a, int b)
5  {
6      return a / b;
7  }
8
9  double dijeljenje(double a, double b)
10 {
11     return a / b;
12 }
13
14 int main()
15 {
16     int x = 5;
17     int y = 2;
18     double n = 5.0;
19     double m = 2.0;
20
21     cout << dijeljenje(x, y) << '\n';
22     cout << dijeljenje(n, m) << '\n';
```

```
    return 0;
}
```

Output

```
2
2.5
```

Prvi poziv odabire funkciju s parametrima tipa `int`, pa je rezultat cjelobrojnoga dijeljenja `2`. Drugi poziv odabire funkciju s parametrima tipa `double`, pa je rezultat `2.5`.

4.8 Ostala svojstva funkcija

4.8.1 Specifikator `inline`

Specifikator `inline` povijesno je uveden kao naznaka da je funkcija prikladna za zamjenu poziva njezinim tijelom. Suvremeni prevoditelji samostalno odlučuju hoće li provesti takvu optimizaciju, neovisno o tome je li funkcija označena specifikatorom `inline`.

Prevodiva jedinica jest jedna izvorna datoteka zajedno sa sadržajem svih zaglavlja uključenih u nju, nakon obrade pretprocesora. Svaka se prevodiva jedinica prevodi zasebno u objektnu datoteku, a povezivač zatim objektne datoteke spaja u program.

Funkcija označena specifikatorom `inline` može se definirati u zaglavlju. Ako je isto zaglavlje uključeno u više izvornih datoteka, definicija funkcije pojavit će se u svakoj od njih. Za običnu funkciju takvo bi ponavljanje definicije izazvalo pogrešku pri povezivanju. Specifikator `inline` dopušta postojanje jednake definicije u više prevodivih jedinica, pri čemu se sve te definicije odnose na istu funkciju. Definicija funkcije mora biti jednaka u svim prevodivim jedinicama.

```
inline int kvadrat(int x)
{
    return x * x;
}
```

U malim programima u jednoj datoteci specifikator `inline` najčešće nije potreban.

4.8.2 Rekurzivne funkcije

Svojstvo funkcije da izravno ili posredno poziva samu sebe naziva se *rekurzija*. Rekurzivna funkcija mora imati uvjet zaustavljanja; bez njega bi se pozivi nastavili sve dok program ne iscrpi raspoložive resurse.

Sljedeća funkcija računa faktorijel nenegativnoga cijelog broja od 0 do 20. Vrijednost `0!` jednaka je `1` i čini osnovni slučaj rekurzije:

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

unsigned long long faktorijel(unsigned int n)
{
    if (n <= 1)
    {
        return 1;
    }

    return n * faktorijel(n - 1);
}

int main()
{
    int broj;

    cout << "Unesi cijeli broj od 0 do 20: ";

    if (!(cin >> broj) || broj < 0 || broj > 20)
    {
        cout << "Neispravan unos.\n";
        return 1;
    }

    unsigned int n = static_cast<unsigned int>(broj);
    cout << broj << "! = " << faktorijel(n) << '\n';
    return 0;
}
```

Output

```
Unesi cijeli broj od 0 do 20: 10
10! = 3628800
```

Tip `unsigned long long` omogućuje veći raspon od tipa `int`, ali ni on ne može prikazati proizvoljno velik faktorijel. Vrijednost `20!` još stane u taj tip, dok bi `21!` prekoračio njegov raspoloživi raspon. Zato program prije poziva funkcije provjerava je li uneseni broj u intervalu od 0 do 20.

Kratki sažetak

Funkcije raščlanjuju program na imenovane cjeline koje se mogu zasebno razumjeti, ispitati i ponovno upotrijebiti. Deklaracija opisuje sučelje funkcije, a definicija sadržava njezino tijelo. Parametri pripadaju deklaraciji i definiciji, dok se argumenti navode pri pozivu. Prijenosom po vrijednosti funkcija dobiva kopiju vrijednosti argumenta, a referentni se parametar veže uz varijablu navedenu kao argument. Prototipovi omogućuju poziv prije definicije, zadane vrijednosti pojednostavnjuju neke

pozive, a preopterećivanje dopušta funkcije istoga imena s različitim popisima parametara. Rekurzivna funkcija poziva samu sebe i mora imati uvjet zaustavljanja.

Pitanja za samoprovjeru

1. Koja je razlika između deklaracije i definicije funkcije?
2. Koje podatke mora sadržavati prototip funkcije i čemu on služi?
3. Koji su standardni oblici funkcije `main`?
4. Koja je razlika između parametra i argumenta?
5. Kada se argument prenosi po vrijednosti, a kada po referenci?
6. Kako se zadaju zadane (*default*) vrijednosti parametara?
7. Kako prevoditelj bira preopterećenu funkciju?
8. Zašto rekurzivna funkcija mora imati uvjet zaustavljanja?

5. Izvedeni oblici podataka

Što ćete naučiti?

Nakon ovoga poglavlja moći ćete deklarirati i obraditi jednodimenzijska i višedimenzijska polja, prenijeti polje funkciji te raditi s C-znakovnim nizovima.

U ovome poglavlju obrađuju se polja podataka: jednodimenzijska i višedimenzijska polja te polja znakova koja se koriste kao C-znakovni nizovi. Klasa `string` obrađuje se zasebno u sljedećemu poglavlju.

5.1 Polja podataka

Često u programu trebamo koristiti skupove istovrsnih podataka. Za označavanje i rukovanje ovakvim podacima bilo bi vrlo nespretno upotrijebiti zasebne varijable $x1$, $x2$, $x3$, ...

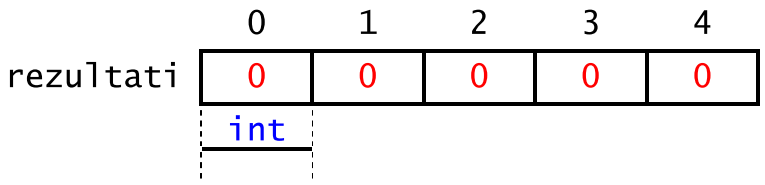
Da bismo to razumjeli, pretpostavimo obradu rezultata nekoga mjerenja koje sadrži 100 podataka, pri čemu svaki podatak treba proći isti algoritam (niz računskih operacija). Ako bismo deklarirali i upotrijebili 100 različitih varijabli, tada bi se program sastojao od 100 istih blokova naredbi koje bi se razlikovale samo u imenu varijable koja se u dotičnome bloku naredbi obrađuje. Očito je da je ovakav način pisanja programa neučinkovit. Puno bolje rješenje je sve podatke smjestiti pod isto ime, a pojedini rezultat dohvaćati pomoću brojčanoga indeksa. Ovakav način organizacije podataka omogućuju polja (*arrays*) podataka.

Polje je konačan niz elemenata istoga tipa. Ti elementi mogu biti bilo kojega tipa:

- ugrađenoga (`int`, `float`, `double`, `char` i sl.) ili
- korisnički definiranoga tipa.

Pojedinim članovima polja može se pristupiti pomoću cjelobrojnoga indeksa te ih mijenjati neovisno o ostalim članovima polja.

Primjerice polje od 5 cjelobrojnih vrijednosti imenom rezultati može se prikazati na sljedeći način:



Slika 5.1: Polje od pet cjelobrojnih elemenata

Svaki pravokutnik predstavlja jedan element polja, a svi su elementi u ovome primjeru cjelobrojnoga tipa. Indeksi su od 0 do 4, što znači da prvi element polja uvijek ima indeks 0, neovisno o duljini polja.

Tipična deklaracija polja u C++-u jest:

```
tip ime[broj_elemenata];
```

gdje je tip valjani tip podataka, ime je valjani identifikator, a unutar zagrada dolazi broj elemenata polja. To znači da bi za naš prethodni primjer ispravna deklaracija bila:

```
int rezultati[5];
```

Napomena

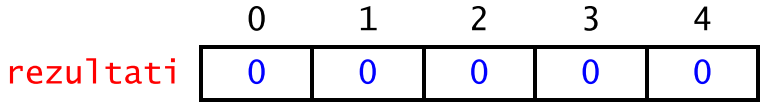
Broj elemenata ugrađenoga polja mora biti zadan cjelobrojnim konstantnim izrazom. Veličina takvoga polja određuje se pri prevođenju i ne može se mijenjati tijekom njegova postojanja. Lokalno polje deklarirano unutar funkcije uobičajeno se pohranjuje na *stacku*. Nastaje tijekom izvođenja funkcije i prestaje postojati izlaskom iz područja u kojem je deklarirano. Globalno polje i lokalno polje deklarirano ključnom riječi `static` imaju statičko trajanje pohrane te ostaju sačuvani do završetka programa.

5.1.1 Inicijalizacija polja

Kada unutar funkcije deklariramo lokalno polje elemenata ugrađenoga tipa, primjerice tipa `int`, bez inicijalizatora, vrijednosti njegovih elemenata neodređene su. Čitanje takve vrijednosti prije pridruživanja valjane vrijednosti pogreška je. Ako polje ugrađenoga tipa ima statičko trajanje pohrane, primjerice ako je globalno ili deklarirano ključnom riječi `static`, njegovi se elementi inicijaliziraju nulom. Za polja objekata klasnih tipova početno stanje određuju konstruktori elemenata. Primjerice, ako je `rezultati` globalno polje tipa `int`:

```
int rezultati[5];
```

svakome elementu polja pridružit će se 0 kao početna vrijednost:

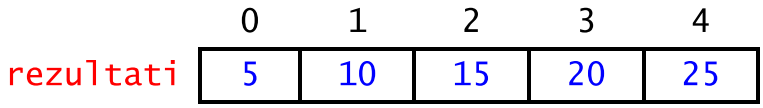


Slika 5.2: Početne vrijednosti elemenata globalnoga polja

Također, kod same deklaracije imamo mogućnost pridružiti početnu vrijednost svakome elementu polja unutar vitičastih zagrada {}. Primjerice:

```
int rezultati[5] = {5, 10, 15, 20, 25};
```

što znači da smo kreirali sljedeće polje:



Slika 5.3: Inicijalizirano polje

Broj inicijalizatora ne smije biti veći od deklariranoga broja elemenata. Ako ih je manje, preostali se elementi inicijaliziraju nulom.

Budući da ovo predstavlja ponavljanje iste informacije (*broj elemenata polja*), C++ ostavlja mogućnost izostavljanja broja elemenata polja unutar pravokutnih zagrada [], kada inicijaliziramo polje na ovaj način:

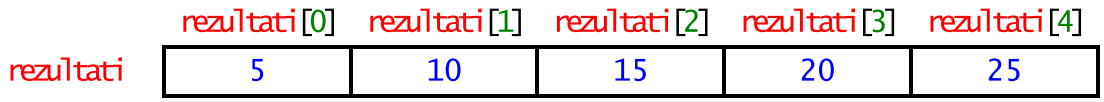
```
int rezultati[] = {5, 10, 15, 20, 25};
```

5.1.2 Pristupanje elementima polja

Bilo gdje u programu gdje je polje vidljivo, možemo pristupiti pojedinome elementu polja kako bismo pročitali njegovu vrijednost ili je promijenili na način:

```
ime[indeks]
```

Tako za naš prethodni primjer, pojedinim elementima polja može se pristupiti na sljedeći način:



Slika 5.4: Pristup elementima polja pomoću indeksa

Tako, ako želimo pridružiti vrijednost 45 drugome elementu to možemo učiniti na način:

```
rezultati[1] = 45;
```

i ovu vrijednost sada možemo pridružiti nekoj varijabli na način:

```
a = rezultati[1];
```

To znači da izraz rezultati[1] ima potpuno isto značenje i svojstva kao i svaka druga cjelobrojna varijabla.

Napomena

Indeks elementa polja za jedan je manji od njegova rednoga broja. To znači da kada želimo pristupiti primjerice *trećemu* elementu polja, to radimo na sljedeći način:

```
rezultati[2];
```

Pristup izvan granica polja nije dopušten: uzrokuje nedefinirano ponašanje. Prevoditelj takvu pogrešku ne mora otkriti, pa se indeks uvijek mora nalaziti u rasponu od 0 do broja elemenata umanjenoga za jedan.

Također, vrlo je bitno uočiti razliku u primjeni pravokutnih zagrada kod polja. Jedna primjena predstavlja navođenje veličine polja kod deklaracije, dok druga mogućnost uporabe indeksa preko kojega se pristupa pojedinome elementu polja:

```
int rezultati[5]; // deklaracija polja
rezultati[1] = 45; // pridruživanje vrijednosti 2. elementu polja
```

Druge valjane operacije s poljima su:

```
rezultati[0] = a;
rezultati[a] = 75;
b = rezultati[a + 2];
rezultati[rezultati[a]] = rezultati[2] + 5;
```

Polje se može upotrijebiti ovako:

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

int main()
{
    int rezultati[] = {16, 2, 77, 40, 12071};
    int zbroj = 0;

    for (int n = 0; n < 5; ++n)
        zbroj += rezultati[n];

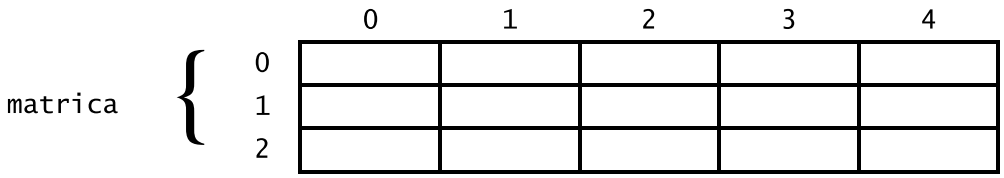
    cout << zbroj << '\n';
    return 0;
}
```

Output

```
12206
```

5.1.3 Višedimenzijska polja

Do sada smo razmatrali jednodimenzijska polja. Međutim često nam trebaju višedimenzijska polja, primjerice matrice. Tako se dvodimenzijsko polje može shvatiti kao matrica:



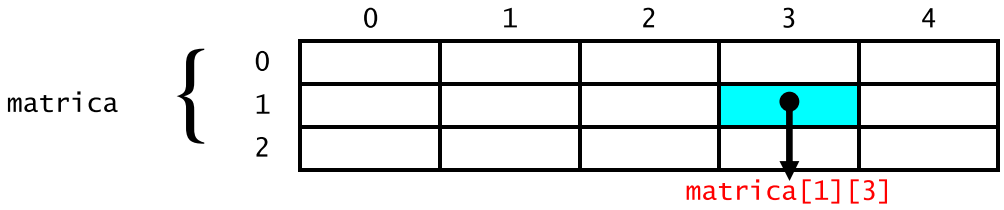
Slika 5.5: Dvodimenzijsko polje s tri retka i pet stupaca

Slika 5.5 prikazuje polje *matrica* dimenzija 3×5, čiji je svaki element cjelobrojnoga tipa. Polje deklarira se na sljedeći način:

```
1 int matrica[3][5];
```

te ako želimo pristupiti elementu u drugome retku i četvrtome stupcu to možemo učiniti na sljedeći način:

```
1 matrica[1][3]
```



Slika 5.6: Element u drugome retku i četvrtome stupcu polja

Položaj elementa `matrica[1][3]` označen je na slici 5.6.

Napomena

Indeksi počinju od nule.

Višedimenzijska polja nisu ograničena samo na dvije dimenzije. Ona mogu imati onoliko dimenzija koliko je potrebno, ali rijetko će se upotrebljavati polja s više od tri dimenzije. Jedan od glavnih razloga za ovo jest potrebna količina memorije, primjerice:

```
char stoljece_sekunde[100][365][24][60][60];
```

gdje pridružujemo jedan znak za svaku sekundu u jednom stoljeću, što rezultira:

$$100 \cdot 365 \cdot 24 \cdot 60 \cdot 60 = 3\,153\,600\,000$$

Kako je `sizeof(char)` jednak jednom bajtu, polje bi zahtijevalo oko 3,15 GB pohrane, ne računajući druga ograničenja programa. Primjer je zato neprikladan kao lokalno polje i služi samo ilustraciji brzoga rasta potrebne memorije.

Višedimenzijsko polje pohranjuje elemente susjednih redaka uzastopno. Iste se vrijednosti mogu ručno rasporediti u jednodimenzijsko polje:

```
int matrica[3][5];
```

ima isti broj elemenata kao

```
int matrica[15];
```

ali tipovi tih dvaju polja nisu jednaki. U prvome slučaju indeksiranje `matrica[r][s]` prevoditelj računa prema poznatoj duljini retka, a u drugome slučaju indeks `r * SIRINA + s` računamo sami.

Višedimenzijsko polje:

```
#include <iostream>

using namespace std;

constexpr int SIRINA = 5;
constexpr int VISINA = 3;

int main()
{
    int matrica[VISINA][SIRINA]{};

    for (int redak = 0; redak < VISINA; ++redak)
        for (int stupac = 0; stupac < SIRINA; ++stupac)
            matrica[redak][stupac] = (redak + 1) * (stupac + 1);

    for (int redak = 0; redak < VISINA; ++redak)
    {
        for (int stupac = 0; stupac < SIRINA; ++stupac)
            cout << matrica[redak][stupac] << ' ';
        cout << '\n';
    }
    return 0;
```

```
}
```

Isti se raspored može prikazati jednodimenzijskim poljem:

```
#include <iostream>

using namespace std;

constexpr int SIRINA = 5;
constexpr int VISINA = 3;

int main()
{
    int matrica[VISINA * SIRINA]{};

    for (int redak = 0; redak < VISINA; ++redak)
        for (int stupac = 0; stupac < SIRINA; ++stupac)
            matrica[redak * SIRINA + stupac] =
                (redak + 1) * (stupac + 1);

    for (int redak = 0; redak < VISINA; ++redak)
    {
        for (int stupac = 0; stupac < SIRINA; ++stupac)
            cout << matrica[redak * SIRINA + stupac] << ' ';
        cout << '\n';
    }
    return 0;
}
```

Oba ova primjera pridružuju vrijednost elementima polja matrica kao što je prikazano na slici:

matrica

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

Slika 5.7: Vrijednosti elemenata dvodimenzijskoga polja

Kao što je već rečeno, ovdje su definirane konstante s `constexpr` koje pojednostavljuju moguću modifikaciju programa. Primjerice, ako želimo povećati broj redaka matrice s 3 na 5 dovoljno je promijeniti samo jedan redak koda:

```
constexpr int VISINA = 3;
```

s

```
constexpr int VISINA = 5;
```

te nisu potrebne nikakve modifikacije u samome programu.

5.1.4 Polja kao parametri funkcije

Kada se polje navede kao parametar funkcije, zapis s uglatim zagradama prilagođava se pokazivaču na prvi element polja. Funkcija zato ne dobiva kopiju cijeloga polja niti iz samoga parametra može odrediti njegovu duljinu. Sljedeći primjer prikazuje deklaraciju takve funkcije:

```
void proracun(int arg[])
```

gdje prenosimo u funkciju cjelobrojno polje arg kao parametar, dok je polje deklarirano na sljedeći način:

```
int polje[40];
```

te je funkcijski poziv:

```
proracun(polje);
```

Polje kao parametar funkcije:

```
#include <iostream>
using namespace std;
void ispisiPolje(const int arg[], int duljina)
{
    for (int n = 0; n < duljina; ++n)
        cout << arg[n] << ' ';
    cout << '\n';
}

int main()
{
    int polje1[] = {5, 10, 15};
    int polje2[] = {2, 4, 6, 8, 10};

    ispisiPolje(polje1, 3);
    ispisiPolje(polje2, 5);
    return 0;
}
```

Output

```
5 10 15
2 4 6 8 10
```

Kao što se vidi iz prethodnoga primjera, prvi parametar `const int arg[]` prihvaća polje cjelobrojnih elemenata bez obzira na njegovu duljinu. Zbog toga se drugim parametrom `duljina` funkciji izričito predaje broj elemenata. Kvalifikator `const` sprječava funkciju da mijenja elemente polja.

Također u funkcijskoj deklaraciji moguće je uključiti višedimenzijska polja. Trodimenzijsko se polje deklarira ovako:

```
tip ime[][druga_dimenzija][treca_dimenzija]
```

što u konkretnome primjeru znači:

```
void proracun(int polje[][3][4])
```

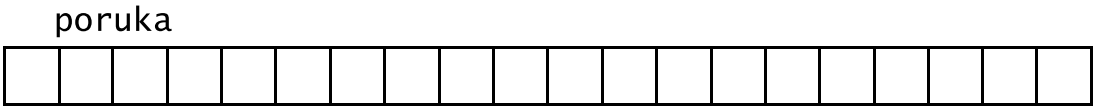
gdje se veličina prve dimenzije može izostaviti, dok se veličine ostalih dimenzija moraju navesti. To će postati jasnije kada se obrade pokazivači.

5.1.5 Znakovni nizovi

U C++-u znakovni niz nije ugrađeni tip podataka. Do sada smo susreli znakovne nizove kao literale pri ispisu. Međutim, u programiranju se često javlja potreba da varijable sadrže znakovne nizove. Rješenje za ovo predstavlja polje znakova:

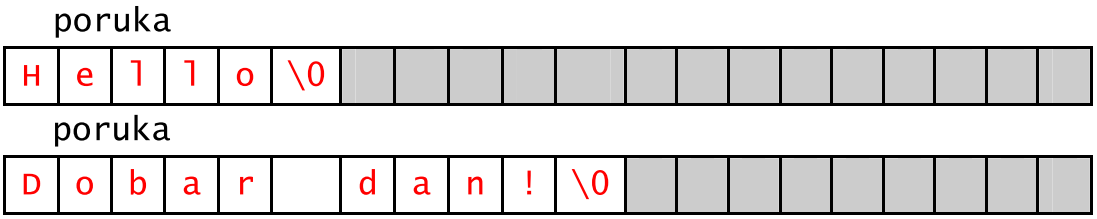
```
char poruka[20]{};
```

može spremiti C-znakovni niz s najviše 19 vidljivih znakova jer je jedno mjesto potrebno za završni nul-znak `'\0'`.



Slika 5.8: Polje znakova s dvadeset elemenata

Najveća dopuštena duljina C-znakovnoga niza ne mora se uvijek iskoristiti. Tako primjerice u znakovno polje poruka možemo spremiti znakovne nizove `"Hello"` ili `"Dobar dan!"` i sl.:



Slika 5.9: C-znakovni nizovi pohranjeni u polju znakova

gdje sivi kvadratići predstavljaju neiskorištena mjesta znakovnoga niza. Kao što je iz slike vidljivo, svaki znakovni niz mora završiti s nul-znakom `'\0'` (engl. *null character*), koji označuje kraj C-znakovnoga niza.

5.1.6 Inicijalizacija znakovnoga niza

Znakovni nizovi predstavljaju polja podataka pa stoga moraju zadovoljiti sva pravila koja se odnose na ovaj izvedeni tip podataka. Ako ih želimo inicijalizirati unaprijed definiranim vrijednostima, to možemo učiniti kod same deklaracije znakovnoga niza na način:

```
char poruka[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

U ovome slučaju deklarirali smo polje od šest elemenata znakovnoga tipa, inicijaliziran skupom znakova koji predstavljaju riječ Hello, te zaključen nul-znakom. Jednostavniji je način inicijalizacija pomoću literala znakovnog niza, koji se piše između dvostrukih navodnika:

```
"rezultat je: "
```

Prevoditelj na kraj literala znakovnog niza automatski dodaje nul-znak.

Polje `poruka` zato možemo inicijalizirati na dva istovrijedna načina:

```
char poruka1[] = {'H', 'e', 'l', 'l', 'o', '\0'};
char poruka2[] = "Hello";
```

U oba se slučaja polje sastoji od šest elemenata: pet znakova čini riječ `Hello`, a šesti je element `'\0'`.

Ovakva inicijalizacija cijeloga polja moguća je samo u trenutku njegove deklaracije. Naknadna pridruživanja prikazana u nastavku nisu valjana:

```
poruka = "Hello"; // nije dopušteno
poruka[] = "Hello"; // nije dopušteno
poruka = {'H', 'e', 'l', 'l', 'o', '\0'}; // nije dopušteno
```

Ovo će također postati jasnije kada se obrade pokazivači.

5.1.7 Pridruživanje vrijednosti C-znakovnom nizu

Budući da se cijelomu polju ne može naknadno pridružiti nova vrijednost, znakove možemo upisati pojedinačno:

```
poruka[0] = 'H';
poruka[1] = 'e';
poruka[2] = 'l';
poruka[3] = 'l';
poruka[4] = 'o';
poruka[5] = '\0';
```

Kao što se može vidjeti ovo nije baš praktična metoda pogotovo za dulje nizove. Za kopiranje C-znakovnoga niza postoji funkcija `strcpy` (*string copy*), deklarirana u zaglavlju `<cstring>`. Odredišno polje mora imati dovoljno mjesta za sve znakove i završni nul-znak jer funkcija sama ne provjerava njegovu veličinu:

```
strcpy(odrediste, izvor);
```

Ova funkcija kopira izvor u odredište. Izvor može biti polje, pokazivač ili literal znakovnog niza:

```
strcpy(poruka, "Hello");
```

Kopiranje C-znakovnoga niza:

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

using namespace std;

int main()
{
    char ime[20];
    strcpy(ime, "Ana Horvat");
    cout << ime << '\n';
    return 0;
}
```

Output

```
Ana Horvat
```

Najčešće pogreške

Funkcija `strcpy()` ne zna veličinu odredišnoga polja. Ako ono nema dovoljno mjesta za izvorni niz i završni nul-znak, dolazi do upisa izvan granica polja i nedefiniranoga ponašanja. Visual C++ zbog toga može prikazati upozorenje C4996. Funkcija `strcpy_s()` nije prenosivo rješenje u svim C++17 okruženjima; sigurnije je provjeriti veličinu ili upotrijebiti klasu `string` opisanu u sljedećemu poglavlju.

Funkciju za kopiranje možemo napisati i sami. Pritom također moramo osigurati da odredišno polje ima dovoljno mjesta:

Vlastita funkcija za kopiranje:

```
#include <iostream>

using namespace std;

void kopirajNiz(char odrediste[], const char izvor[])
{
    int i = 0;
    do
    {
```

```
        odrediste[i] = izvor[i];
    }
    while (izvor[i++] != '\0');
}

int main()
{
    char mojeIme[20];
    kopirajNiz(mojeIme, "Ivan Horvat");
    cout << mojeIme << '\n';
    return 0;
}
```

Output

```
Ivan Horvat
```

Vrijednost se u polje znakova može učitati i iz ulaznoga toka `cin`. Za unos cijeloga retka koristi se funkcijski član `getline()`:

```
cin.getline(spremnik, duljina, granicnik);
```

Prvi je argument polje u koje se sprema unos, drugi je njegova veličina, a ne najveći broj vidljivih znakova. Treći, neobvezni argument određuje graničnik unosa; zadana je vrijednost znak novoga retka `'\n'`.

Unos C-znakovnoga niza:

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

int main()
{
    char mojSpremnik[100];

    cout << "Unesite ime i prezime: ";
    cin.getline(mojSpremnik, 100);
    cout << "Bok, " << mojSpremnik << ".\n";

    cout << "Unesite naziv kluba: ";
    cin.getline(mojSpremnik, 100);
    cout << "I ja navijam za " << mojSpremnik << ".\n";
    return 0;
}
```

Output

```
Unesite ime i prezime: Ivan Horvat
Bok, Ivan Horvat.
Unesite naziv kluba: Hajduk
I ja navijam za Hajduk.
```

U primjeru smo dvaput upotrijebili `cin.getline()` s istim poljem `mojSpremnik`, pa drugi unos zamjenjuje prethodni sadržaj. Za unos s tipkovnice mogli bismo upotrijebiti i operator >>:

```
cin >> mojSpremnik;
```

što radi, ali ima sljedeća ograničenja:

- Može se unijeti samo jedna riječ, tj. unos se prekida čim unesemo razmak.
- Potrebno je paziti na veličinu polja kako unos ne bi prešao njegove granice.

5.1.8 Pretvorba C-znakovnoga niza u broj

U programiranju se može javiti potreba za pretvorbom C-znakovnoga niza u brojčani tip podataka. Za tu svrhu postoje sljedeće funkcije:

- `atoi()` pretvara C-znakovni niz u tip `int`;
- `atol()` pretvara C-znakovni niz u tip `long`;
- `atof()` pretvara C-znakovni niz u tip `double`.

Sve ove funkcije primaju C-znakovni niz. Ne omogućuju pouzdano razlikovanje nevaljanoga unosa od vrijednosti nula, pa se u robusnijemu kodu prednost daje funkcijama `strtol`/`strtod` ili, za objekte klase `string`, funkcijama `stoi`/`stod`.

Funkcije `atoi()` i `atof()`:

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

int main()
{
    char mojUnos[100];

    cout << "Unesite cijenu u eurima: ";
    cin.getline(mojUnos, 100);
    double cijena = atof(mojUnos);

    cout << "Unesite broj proizvoda: ";
    cin.getline(mojUnos, 100);
```

```
    int kolicina = atoi(mojUnos);

    cout << "Ukupna cijena: " << cijena * kolicina << " eura\n";
    return 0;
}
```

Output

```
Unesite cijenu u eurima: 23.99
Unesite broj proizvoda: 5
Ukupna cijena: 119.95 eura
```

Sljedeće funkcije za rad s C-znakovnim nizovima deklarirane su u zaglavlju `<cstring>`:

```
char* strcat(char* odrediste, const char* izvor);
```

Dodaje izvor na kraj odredišta i vraća pokazivač na odredište. Odredište mora imati dovoljno mjesta za prošireni niz.

```
int strcmp(const char* prvi, const char* drugi);
```

Leksikografski uspoređuje dva C-znakovna niza. Vraća nulu ako su jednaki, negativnu vrijednost ako je prvi manji te pozitivnu ako je prvi veći.

```
char* strcpy(char* odrediste, const char* izvor);
```

Kopira izvor u odredište i vraća pokazivač na odredište.

```
size_t strlen(const char* niz);
```

Vraća broj znakova prije završnoga nul-znaka; nul-znak se ne ubraja u duljinu.

Pokazivač `char*` i polje `char[]` blisko su povezani, ali nisu isti tip objekta: polje ima vlastitu pohranu i poznatu veličinu, dok pokazivač samo sadrži adresu.

Zapamtite

- Indeks prvoga elementa polja uvijek je 0.
- Pristup izvan granica polja uzrokuje nedefinirano ponašanje.
- Funkcija kojoj predajemo polje u pravilu mora zasebno dobiti i njegovu duljinu.
- C-znakovni niz završava nul-znakom `'\0'`.
- Funkcije iz zaglavlja `<cstring>` ne provjeravaju uvijek veličinu odredišnoga polja.

Klasa `string`, koju ćemo upoznati u sljedećemu poglavlju, automatski upravlja pohranom teksta i smanjuje opasnost od pogrešaka karakterističnih za C-znakovne nizove.

Kratki sažetak

Polje je niz elemenata istoga tipa kojima se pristupa indeksima od nule do veličine polja umanjene za jedan. Višedimenzijska polja organiziraju elemente u više indeksnih dimenzija, a pri prijenosu polja funkciji u pravilu treba zasebno navesti njegovu duljinu. C-znakovni niz jest polje znakova završeno nul-znakom `'\0'`. Funkcije iz zaglavlja `<cstring>` omogućuju kopiranje, spajanje, uspoređivanje i određivanje duljine takvih nizova, ali programer mora osigurati dovoljno prostora i spriječiti pristup izvan granica polja.

Pitanja za samoprovjeru

1. Koji je indeks prvoga elementa polja?
2. Zašto funkciji uz polje obično predajemo i njegovu duljinu?
3. Čemu služi završni znak \0 u C-znakovnom nizu?

6. Klasa string

Što ćete naučiti?

Nakon ovoga poglavlja moći ćete deklarirati i inicijalizirati objekte klase `string`, unositi i ispisivati tekst, pristupati pojedinim znakovima, spajati, uspoređivati, pretraživati i mijenjati tekst te sigurno prenositi string funkciji.

U prethodnome poglavlju upoznali smo polja znakova i C-znakovne nizove. Tip `char` omogućuje pohranu jednoga znaka, dok je u programima često potrebno pohraniti cijelu riječ, rečenicu ili tekst promjenjive duljine.

Standardna biblioteka C++-a za rad s tekstom nudi klasu `string`, definiranu u zaglavlju `<string>`. Njezin je puni naziv `std::string`, ali u primjerima se, u skladu s ostatkom skripte, koristi naredba `using namespace std;`. Za razliku od polja znakova, objekt klase `string` automatski upravlja vlastitom pohranom i omogućuje jednostavan unos, ispis, spajanje, uspoređivanje i pretraživanje teksta.

Tablica 6.1: Osnovna usporedba C-znakovnoga niza i klase `string`

Svojstvo	C-znakovni niz	Klasa `string`
Pohrana	Polje znakova fiksne veličine	Automatski se prilagođava sadržaju
Kraj teksta	Mora sadržavati nul-znak	Objekt sam vodi evidenciju o duljini
Pridruživanje	Ne može se pridružiti cijelomu polju	Podržano operatorom =
Spajanje	Funkcija `strcat()`	Operator + ili +=
Usporedba	Funkcija `strcmp()`	Relacijski operatori

6.1 Deklaracija i inicijalizacija

Za uporabu klase `string` potrebno je uključiti zaglavlje:

```
#include <string>
```

Objekt klase `string` može se deklarirati i inicijalizirati na više načina:

```
string ime;
string prezime = "Horvat";
string grad("Split");
string prazno{};
```

Vrijednost objekta može se mijenjati tijekom izvođenja programa:

```
ime = "Ana";
```

6.2 Unos i ispis

Za unos jedne riječi može se koristiti operator >>:

```
string ime;
cout << "Unesite ime: ";
cin >> ime;
cout << "Unijeli ste: " << ime << '\n';
```

Operator >> prekida unos na prvome razmaku ili drugome bijelom znaku. Zbog toga za unos cijeloga retka treba koristiti funkciju `getline()`:

```
string puno_ime;
cout << "Unesite ime i prezime: ";
getline(cin, puno_ime);
cout << "Unijeli ste: " << puno_ime << '\n';
```

Napomena

Ako se prije funkcije `getline()` koristi operator >>, u ulaznome spremniku ostaje znak novoga retka. Tada ga prije poziva `getline()` treba ukloniti. Nakon uključivanja zaglavlja `<limits>` može se napisati:

```
cin.ignore(numeric_limits<streamsize>::max(), '\n');
getline(cin, puno_ime);
```

6.3 Osnovne operacije

6.3.1 Pridruživanje i spajanje

Vrijednost jednoga stringa može se pridružiti drugome:

```
string a = "Dobar";
string b;
b = a;
```

Stringovi se mogu spajati operatorom +:

```
string ime = "Ana";
string prezime = "Horvat";
string puno_ime = ime + " " + prezime;
cout << puno_ime << '\n';
```

Operator += dodaje sadržaj postojećemu stringu:

```
puno_ime += " iz Splita";
```

6.3.2 Uspoređivanje

Stringovi se mogu uspoređivati operatorima ==, !=, <, > i ostalim relacijskim operatorima:

```
string a = "Ana";
string b = "Ivan";
if (a == b)
    cout << "Stringovi su jednaki.";
else
    cout << "Stringovi nisu jednaki.";
```

Usporedba operatorima < i > obavlja se leksikografski, prema vrijednostima znakova u znakovnome skupu. Zbog toga velika i mala slova nisu jednaka, a takva usporedba nije isto što i jezično abecedno razvrstavanje hrvatskih riječi.

6.4 Pristup pojedinim znakovima

Znakovima stringa pristupa se indeksima koji počinju od nule, kao i kod polja:

```
string grad = "Split";
cout << grad[0] << '\n';      // S
grad[0] = 's';
cout << grad << '\n';         // split
```

Operator `[]` ne provjerava nalazi li se indeks unutar granica stringa. Funkcijski član `at()` provjerava granice i pri nevaljanu indeksu baca iznimku `out_of_range`. U oba slučaja valjani su indeksi od 0 do `size() - 1`. Praznomu stringu ne smije se pristupati ni jednim od tih načina.

Svi se znakovi mogu proći rasponskom petljom `for`:

```
for (char znak : grad)
    cout << znak << ' ';
```

Napomena

Klasa `string` pohranjuje elemente tipa `char` i sama ne tumači Unicode znakove. Ako je tekst zapisan u UTF-8 kodiranju, hrvatsko slovo poput `č` može zauzimati više od jednoga elementa. Tada `size()` i `length()` vraćaju broj bajtova, a ne nužno broj vidljivih slova.

6.5 Najčešće korišteni članovi klase

Tablica 6.2: Najčešće korišteni članovi klase `string`

Član klase	Opis
`length()`	Vraća broj elemenata tipa `char` u stringu.
`size()`	Vraća broj elemenata tipa `char`; funkcionalno je jednaka funkciji `length()`.
`empty()`	Provjerava je li string prazan.
`clear()`	Briše cijeli sadržaj stringa.
`append()`	Dodaje tekst na kraj stringa.
`insert()`	Umeće tekst na zadano mjesto.
`erase()`	Briše dio ili cijeli sadržaj stringa.
`replace()`	Zamjenjuje dio stringa novim tekstom.
`substr()`	Izdvaja podniz, odnosno dio stringa.
`find()`	Pronalazi prvo pojavljivanje podniza ili znaka.
`compare()`	Uspoređuje dva stringa.
`at()`	Pristupa znaku uz provjeru granica.

Primjena nekih od navedenih članova prikazana je u sljedećem primjeru:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string tekst = "Programiranje u C++";
    cout << "Duljina: " << tekst.length() << '\n';
    cout << "Prvi znak: " << tekst.at(0) << '\n';
    if (!tekst.empty())
        cout << "String nije prazan.\n";
    cout << "Podniz: " << tekst.substr(0, 13) << '\n';
    return 0;
}
```

Funkcija `find()` vraća indeks prvoga pronađenog znaka ili podniza. Ako traženi sadržaj nije pronađen, vraća posebnu vrijednost `string::npos`:

```
string tekst = "Programiranje u C++";
size_t polozaj = tekst.find("C++");

if (polozaj != string::npos)
    cout << "Indeks: " << polozaj << '\n';
else
```

```
7     cout << "Nema rezultata.\n";
```

Sljedeći primjer pokazuje kako se sadržaj može umetati, brisati i zamjenjivati:

```
1 string tekst = "programiram Java.";
2 tekst.insert(0, "Rado ");
3 tekst.replace(17, 4, "C++");
4 tekst.erase(tekst.size() - 1, 1);
5 cout << tekst << '\n';
```

Output

```
Rado programiram C++
```

6.6 Pregled ostalih članova klase

Tablica 6.3: Pregled ostalih članova klase `string`

Član klase	Namjena
`front()`	Vraća prvi znak stringa; string ne smije biti prazan.
`back()`	Vraća posljednji znak stringa; string ne smije biti prazan.
`push_back()`	Dodaje jedan znak na kraj stringa.
`pop_back()`	Uklanja posljednji znak; string ne smije biti prazan.
`resize()`	Mijenja duljinu stringa.
`capacity()`	Vraća trenutačno rezervirani kapacitet memorije.
`reserve()`	Rezervira memoriju za budući sadržaj.
`shrink_to_fit()`	Zahtijeva smanjenje kapaciteta; implementacija zahtjev ne mora prihvatiti.
`swap()`	Zamjenjuje sadržaj dvaju stringova.
`copy()`	Kopira dio stringa u polje znakova; ne dodaje završni nul-znak.
`c_str()`	Vraća C-znakovni niz tipa `const char*`.
`data()`	Vraća pokazivač na neprekinuti niz znakova; u C++17 nekonstantni objekt omogućuje mijenjanje postojećih znakova, ali ne i promjenu veličine stringa.
`rfind()`	Traži posljednje pojavljivanje podniza ili znaka.
`find_first_of()`	Traži prvi znak iz zadanoga skupa znakova.
`find_last_of()`	Traži posljednji znak iz zadanoga skupa znakova.
`find_first_not_of()`	Traži prvi znak koji nije iz zadanoga skupa.
`find_last_not_of()`	Traži posljednji znak koji nije iz zadanoga skupa.

6.7 Prijenos stringa funkciji

String se kao argument može prenijeti funkciji po vrijednosti ili po referenci. Pri prijenosu po vrijednosti stvara se kopija objekta. Za izbjegavanje kopiranja preporučuje se prijenos po konstantnoj referenci kada funkcija ne treba mijenjati sadržaj:

```
void ispisi(const string& tekst)
{
    cout << tekst << '\n';
}
```

Ako funkcija treba mijenjati izvorni string, koristi se obična referenca:

```
void dodajUsklicnik(string& tekst)
```

```
{
    tekst += "!";
}
```

Najčešće pogreške

- Zaboravljeno uključivanje zaglavlja `<string>`.
- Uporaba operatora >> kada string treba sadržavati razmake.
- Miješanje operatora >> i funkcije `getline()` bez čišćenja ulaznoga spremnika.
- Pristup znaku izvan granica stringa.
- Zamjena vrijednosti koje vraćaju funkcije `length()` i `capacity()`.
- Nepotrebno korištenje polja znakova kada je prikladniji objekt klase `string`.

Zapamtite

- Klasa `string` služi za pohranu i obradu tekstualnih podataka.
- Definirana je u zaglavlju `<string>`.
- Funkcija `getline()` omogućuje unos cijeloga retka teksta.
- Stringovi se mogu spajati operatorom +.
- Stringovi se mogu uspoređivati relacijskim operatorima.
- Neuspješno pretraživanje označuje se vrijednošću `string::npos`.
- Za prijenos stringa funkciji bez kopiranja koristi se konstantna referenca.

Kratki sažetak

U ovome poglavlju upoznali smo klasu `string`, koja omogućuje jednostavnu pohranu i obradu tekstualnih podataka. Prikazani su deklaracija, inicijalizacija, unos i ispis stringova te osnovne operacije pridruživanja, spajanja i uspoređivanja. Objašnjeni su najčešće korišteni članovi klase, uključujući funkcije za određivanje duljine, pretraživanje, izdvajanje i zamjenu dijelova stringa. Posebno je prikazana razlika između operatora >> i funkcije `getline()`, kao i prijenos stringa funkciji konstantnom referencom radi izbjegavanja nepotrebnoga kopiranja.

Pitanja za samoprovjeru

1. Čemu služi klasa `string`?
2. Koje je zaglavlje potrebno uključiti za rad sa stringovima?
3. Koja je razlika između operatora >> i funkcije `getline()`?

4. Kako se dva stringa mogu spojiti?
5. Kako se provjerava jesu li dva stringa jednaka?
6. Koja je razlika između funkcija `length()` i `capacity()`?
7. Čemu služi funkcija `substr()`?
8. Koju vrijednost vraća funkcija `find()` ako podniz nije pronađen?
9. Zašto se pri prijenosu stringa funkciji često koristi konstantna referenca?
10. Koja je razlika između funkcije `at()` i operatora `[]`?
11. Čemu služe funkcije `append()`, `insert()` i `replace()`?
12. Kada se koristi funkcija `c_str()`?

7. Pokazivači

Što ćete naučiti?

Nakon ovoga poglavlja moći ćete objasniti adresu objekta i pokazivač, pravilno dereferencirati pokazivač te povezati pokazivače i polja.

Do sada smo upoznali varijable kao imenovane objekte kojima možemo pristupiti preko njihova imena (identifikatora). Ti se objekti spremaju na određena mjesta u memoriji računala. *Minimalna veličina adresabilne memorije* u C++-ovu apstraktnom modelu jest jedan bajt. Bajt ima najmanje osam bitova; na suvremenim računalima u pravilu ima osam bitova i tada je jednak oktetu. Svakomu je bajtu pridružena adresa.

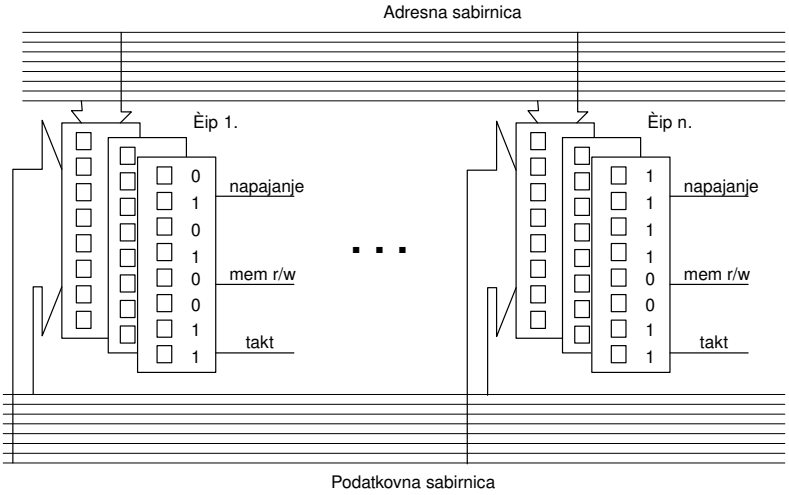
Memorija računala može se shvatiti kao jedna ulica u nekome gradu. U toj ulici sve kuće i zgrade imaju jedinstvene oznake (adrese), te primjerice ako trebamo naći kuću u ulici *Kavanjinova* 4 to ne bi trebalo predstavljati nikakav problem, jer postoji samo jedna kuća s tom adresom i nalazi se između brojeva 3 i 5.

Na isti se način memorija može zamisliti kao linearno uređen adresni prostor, te ako trebamo pristupiti memorijskoj lokaciji 1000 znamo da je ona jedinstvena i da se nalazi između adresa 999 i 1001. Prije nego se pokaže kako se na apstraktnoj razini manipulira podacima koji se nalaze u radnoj memoriji računala, potrebno je ukratko prikazati sklopovske karakteristike radne memorije računala. Elektronička memorijska jedinica može pamtiti 1 bit informacije. Najčešće se realizira kao poluvodički elektronički sklop. Više ovakvih sklopova pakira se u integrirani elektronički sklop koji se popularno naziva čip. U računalu se memorijski čipovi povezuju tako da se istovremeno može pristupiti skupini od 8 ili više temeljnih memorijskih jedinica, pa se sa programske točke motrišta može napisati da jedna memorijska ćelija sadrži 1 bajt (*byte*) digitalne informacije. Informacije između procesora i memorije ne prenose se serijski bit po bit, već se istovremeno (paralelno) prenosi više digitalnih znamenki (8, 16, 32 ili 64 bita) pomoću višestrukih (paralelnih) električnih vodova. To ovisi o dva čimbenika: širini sabirnice i vrsti procesora. Skupina električnih vodova kojima se paralelno prenose digitalne informacije, naziva se sabirnica (*bus*).

Karakteristične su tri vrste sabirnica:

1. Sabirnica podataka (*data bus*) koristi se za paralelan prijenos jednoga ili više bajtova digitalne informacije između procesora i skupine memorijskih ili ulazno-izlaznih jedinica.
2. Adresna sabirnica (*address bus*) koristi se za odabir memorijske ćelije radne memorije ili memorije ulazno-izlaznih jedinica u koju se, odnosno iz koje se, prenose digitalne informacije.
3. Upravljačka sabirnica (*control bus*) prenosi električne signale za upravljanje i vremensko usklađivanje rada računalnih komponenti.

   Na slici 7.1 prikazana je pojednostavnjena shema spajanja memorijskih integriranih strujnih krugova.



Slika 7.1: Pojednostavnjeni prikaz memorije računala

Upisivanje ili čitanje sadržaja memorijske skupine nadzire se upravljačkim signalom za čitanje odnosno pisanje (*memory read/write*). Rad memorije vremenski se usklađuje s ostalim sklopovima računala signalima generatora takta. Aktiviranje pojedine memorijske skupine određeno je binarnom kombinacijom naponskih stanja adresne sabirnice. Memorija je stoga linearno uređen prostor jer brojčana vrijednost na adresnoj sabirnici određuje položaj kojemu se pristupa. Na programskoj razini kod objekta razlikujemo:

- adresu početka područja memorije koje objekt zauzima i
- vrijednost pohranjenu u tome području.

Veličina *adresne sabirnice* i *sabirnice podataka* iskazuje se brojem bitova koje one u nekome trenutku prenose i određena je mogućnošću procesora odnosno sklopovskim svojstvima računala. Teorijski broj adresabilnih bajtova određen je izrazom:

$$2^{\text{broj bitova adresne sabirnice}} \text{ bajta.}$$

Primjerice, 8-bitni mikroprocesori (Z-80, Motorola 6800, Intel 8080) imaju 16-bitnu adresnu sabirnicu i 8-bitnu sabirnicu podataka, a maksimalan kapacitet njihove memorije iznosi 64 kilobajta. Prva osobna računala klase IBM PC XT imala su adresnu sabirnicu od 20 linija i 8-bitnu sabirnicu podataka, pa je maksimalan kapacitet memorije tih računala iznosio 1 MB.

Širina adrese ne mora biti jednaka širini sabirnice podataka ni veličini pokazivača koju program vidi. Teorijski adresni prostor s n adresnih bitova sadržava 2^n adresa, ali stvarno dostupan prostor ovisi o procesoru, operacijskome sustavu i implementaciji. Na temelju onoga što je do sada kazano o kodiranju sadržaja i sklopovskome ustrojstvu memorije, važne su sljedeće činjenice i definicije:

1. Objekt zauzima određeno područje memorije, ima svoj tip i pohranjuje vrijednost.
2. Adresa jednoznačno određuje položaj adresabilnoga bajta u memorijskome prostoru.

3. Značenje pohranjenih bitova ovisi o tipu objekta. Isti niz bitova može se različito tumačiti kao broj, znak ili logička vrijednost.
4. Vrijednost objekta može se čitati, a ako tip i svojstva objekta to dopuštaju, može se i mijenjati.
5. Varijable se uvode deklaracijama i imenima omogućuju pristup objektima u programu.

7.1 Adresni operator (&)

U trenutku kada deklariramo neku varijablu, za nju se mora osigurati određeno mjesto u memoriji računala. Mi u pravilu, ne znamo gdje će varijabla bit smještena; o tome vode računa prevoditelj, izvršno okruženje i operacijski sustav. Kada se za objekt osigura memorija, možemo saznati adresu gdje je ta varijabla spremljena. Ovo se može jednostavno učiniti tako što ispred imena varijable napišemo znak (&) (*ampersand*) koji u ovome kontekstu ima značenje *adresa od* (*address of*):

```
1 int* pokazivac1 = &varijabla1;
```

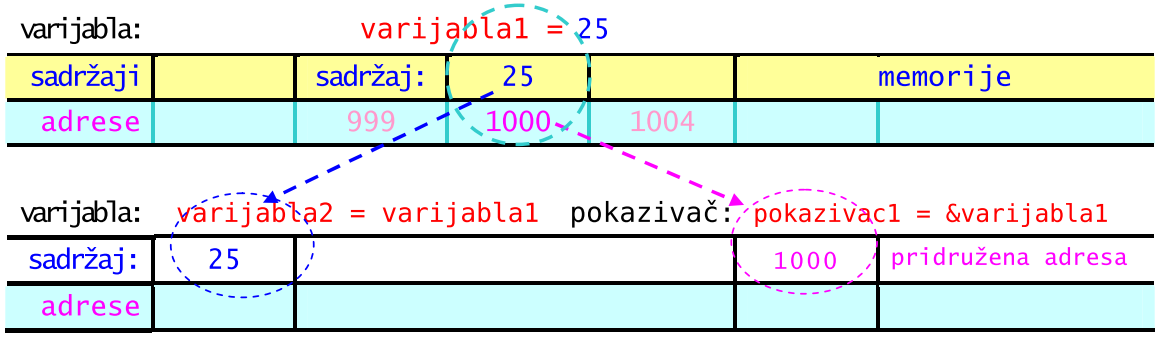
gdje će se pokazivaču `pokazivac1` pridružiti adresa varijable `varijabla1`. Na ovaj način pokazivaču `pokazivac1`, koji je smješten na nekoj drugoj adresi, kao sadržaj upisuje se:

- adresa varijable `varijabla1`, a ne
- vrijednost (sadržaj) varijable `varijabla1`.

Ako pretpostavimo da je `varijabla1` spremljena na adresi 1000 u memoriji računala te napišemo sljedeći kod:

```
1 varijabla1 = 25;
2 varijabla2 = varijabla1;       /* varijabli varijabla2 pridružuje se 25 */
3 pokazivac1 = &varijabla1;      /* pokazivaču pokazivac1 pridružuje se 1000 */
```

Odnos tih vrijednosti prikazan je na slici 7.2.



Slika 7.2: Adresni operator i sadržaj pokazivača

U navedenome primjeru varijabli `varijabla2` pridružili smo vrijednost varijable `varijabla1`, dok smo pokazivaču `pokazivac1` pridružili adresu varijable `varijabla1`. Varijable u koje spremamo adrese

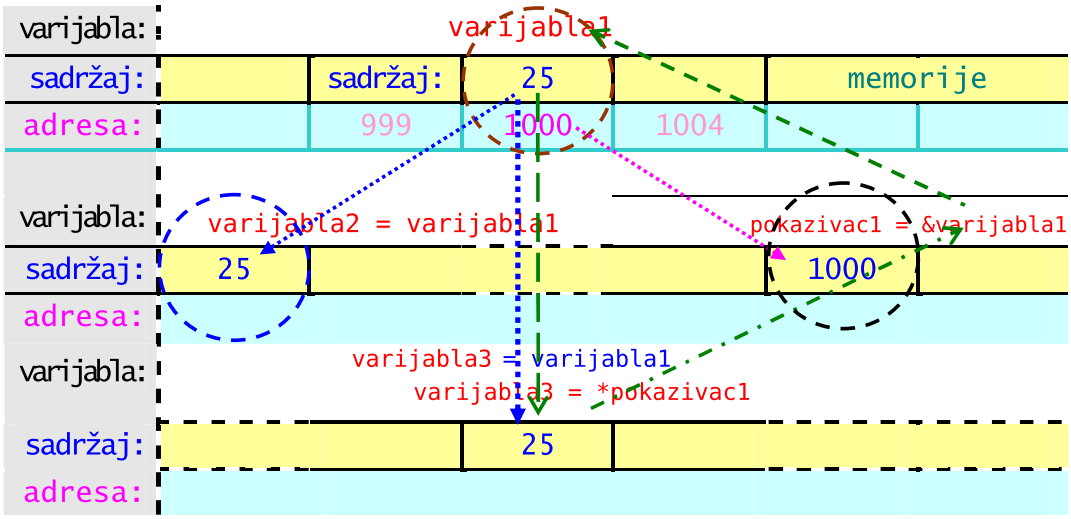
drugih objekata nazivamo pokazivačima (*pointers*). Pokazivač mora biti deklariran prije uporabe, primjerice `int* pokazivac1`.

7.2 Operator dereferenciranja (*)

Primjenom pokazivača, a pomoću operatora dereferenciranja *, možemo izravno pristupiti vrijednosti varijable na koju pokazivač pokazuje. Operator dereferenciranja ima značenje "*vrijednost na koju pokazuje*" (*value pointed by*):

```
varijabla3 = *pokazivac1;       // varijabla3 = 25
```

Pokazivač `pokazivac1` sadržava adresu varijable `varijabla1`. Dereferenciranjem pokazivača pristupa se vrijednosti na toj adresi, pa se vrijednost 25 pridružuje varijabli `varijabla3`. Taj je odnos prikazan na slici 7.3.



Slika 7.3: Operator dereferenciranja i pristup vrijednosti objekta

Pokazivač `pokazivac1` i sam je objekt te ima vlastitu adresu, ali njegova je vrijednost adresa objekta na koji pokazuje. Izraz `*pokazivac1` ne označuje pokazivač, nego objekt na koji pokazivač pokazuje. Zato se preko toga izraza vrijednost može i pročitati i promijeniti:

```
varijabla3 = *pokazivac1;       /* čita vrijednost varijable1 */
varijabla2 = varijabla1;        // izravno kopira vrijednost varijable1
*pokazivac1 = 30;               /* mijenja varijablu1 preko pokazivača */
```

Ako sve do sada sažmemo na sljedećem primjeru:

```
varijabla1 = 25;
pokazivac1 = &varijabla1;       // pretpostavljena adresa 1000
varijabla3 = *pokazivac1;       // varijabla3 = 25
```

onda nam mora biti jasno da je:

```
varijabla1 == 25;
pokazivac1 == &varijabla1;
*pokazivac1 == 25;
varijabla3 == *pokazivac1;
```

Uz pretpostavljenu adresu 1000, i `pokazivac1` i izraz `&varijabla1` imaju vrijednost 1000.

```
#include <iostream>
using namespace std;
int main()
{
    int varijabla1 = 25;
    int varijabla2;
    int varijabla3;
    int* pokazivac1 = &varijabla1;

    varijabla2 = varijabla1;
    varijabla3 = *pokazivac1;

    cout << "Vrijednost varijable1: " << varijabla1 << '\n';
    cout << "Adresa varijable1: " << &varijabla1 << '\n';
    cout << "Vrijednost pokazivača1: " << pokazivac1 << '\n';
    cout << "Adresa pokazivača1: " << &pokazivac1 << '\n';
    cout << "Izravno kopirana vrijednost: " << varijabla2 << '\n';
    cout << "Vrijednost dohvaćena pokazivačem: " << varijabla3 << '\n';

    return 0;
}
```

Output

```
Vrijednost varijable1: 25
Adresa varijable1: 0xA...
Vrijednost pokazivača1: 0xA...
Adresa pokazivača1: 0xB...
Izravno kopirana vrijednost: 25
Vrijednost dohvaćena pokazivačem: 25
```

Adrese su u prikazu skraćene. Jednake oznake `0xA...` pokazuju da pokazivač pohranjuje adresu varijable `varijabla1`, dok se sam pokazivač nalazi na drugoj adresi.

7.3 Deklaracija pokazivača

Pokazivač deklariramo na sljedeći način:

```
tip* ime_pokazivaca;
```

gdje `tip` određuje tip objekta na koji pokazivač smije pokazivati:

```
int* cijeliBroj;       /* pokazivač na int */
char* znak;            /* pokazivač na char */
float* realniBroj;     /* pokazivač na float */
```

Pokazivaču se prilikom deklaracije može odmah pridružiti adresa objekta:

```
int broj;
int* pokazivac = &broj;
```

Isti se odnos može uspostaviti i naknadnim pridruživanjem:

```
int broj;
int* pokazivac;
pokazivac = &broj;
```

U oba slučaja pokazivač `pokazivac` pohranjuje adresu varijable `broj`. Vrijednost varijable dohvaća se tek dereferenciranjem pokazivača. Stoga sljedeće pridruživanje nije ispravno:

```
int broj;
int* pokazivac;
pokazivac = broj;       /* pogrešno: broj nije adresa */
```

Pokazivač koji još ne treba pokazivati ni na jedan objekt treba inicijalizirati vrijednošću `nullptr`. Neinicijalizirani pokazivač ima neodređenu vrijednost i ne smije se dereferencirati.

Vrijednost se preko pokazivača može promijeniti ovako:

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

int main()
{
    int varijabla1 = 5;
    int varijabla2 = 15;
    int* pokazivac1 = &varijabla1;

    cout << "Početna vrijednost varijable1: " << *pokazivac1 << '\n';
    *pokazivac1 = 10;

    pokazivac1 = &varijabla2;
    cout << "Početna vrijednost varijable2: " << *pokazivac1 << '\n';
    *pokazivac1 = 20;

    cout << "varijabla1 = " << varijabla1 << '\n';
    cout << "varijabla2 = " << varijabla2 << '\n';

```

```
    return 0;
}
```

Output

```
Početna vrijednost varijable1: 5
Početna vrijednost varijable2: 15
varijabla1 = 10
varijabla2 = 20
```

Pokazivač ima vlastitu adresu, koja se razlikuje od adrese koju pohranjuje:

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

int main()
{
    int varijabla1 = 25;
    int* pokazivac1 = &varijabla1;

    cout << "Vrijednost varijable1: " << varijabla1 << '\n';
    cout << "Adresa varijable1: " << &varijabla1 << '\n';
    cout << "Vrijednost pokazivača1: " << pokazivac1 << '\n';
    cout << "Adresa pokazivača1: " << &pokazivac1 << '\n';

    return 0;
}
```

Ispisane adrese ovise o pojedinome pokretanju programa i ne treba ih smatrati stalnim vrijednostima.

Moguće je deklarirati više pokazivača u jednoj naredbi, ali se znak * mora navesti uz svaki deklarator. U ovoj ćemo skripti radi preglednosti svaki pokazivač deklarirati zasebno:

```
int* pokazivac1;
int* pokazivac2;
```

Odvojene deklaracije preglednije pokazuju da je svaka varijabla pokazivač.

7.4 Pokazivači i polja

Koncepti polja i pokazivača blisko su povezani. U većini izraza ime polja pretvara se u pokazivač na njegov prvi element. Polje ipak nije pokazivač: ima vlastitu pohranu i poznat broj elemenata, a njegovu se imenu ne može pridružiti druga adresa.

```
int brojevi[20];
int* p;
```

tako da je sljedeći izraz ispravan:

```
p = brojevi;
```

U navedenom se izrazu ime polja `brojevi` pretvara u pokazivač na prvi element polja, odnosno u izraz `&brojevi[0]`. To ne znači da je polje pokazivač. Pokazivaču `p` može se pridružiti druga adresa, dok polje nije objekt kojemu se može pridruživati vrijednost. Stoga sljedeće pridruživanje nije dopušteno:

```
brojevi = p;       /* pogrešno: polju se ne može pridruživati */
```

Elementima polja preko pokazivača pristupa se ovako:

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

int main()
{
    int brojevi[5]{};
    int* p = brojevi;

    for (int n = 0; n < 5; ++n)
    {
        *(p + n) = 10 * (n + 1);
    }

    for (int broj : brojevi)
    {
        cout << broj << ' ';
    }
    cout << '\n';

    return 0;
}
```

Output

```
10 20 30 40 50
```

Do sada smo elementima polja pristupali operatorom indeksiranja `[]`. Izraz `a[n]` ekvivalentan je izrazu `*(a + n)`:

```
a[5] = 0;
*(a + 5) = 0;
```

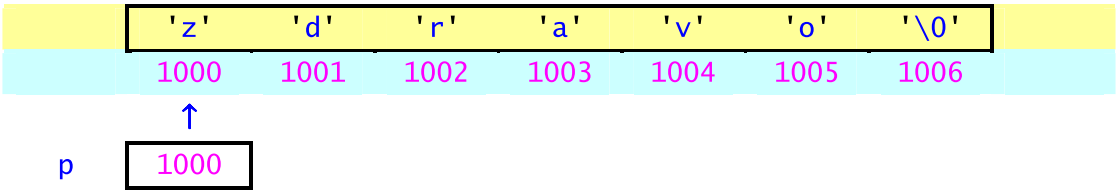
Oba izraza pristupaju šestomu elementu polja jer indeksiranje počinje od nule.

7.5 Pokazivači i znakovna polja

Ako deklariramo pokazivač na znakovni tip, tada možemo pisati:

```
1 char tekst[] = "zdravo";
2 char* p = tekst;
```

što je prikazano na slici 7.4:



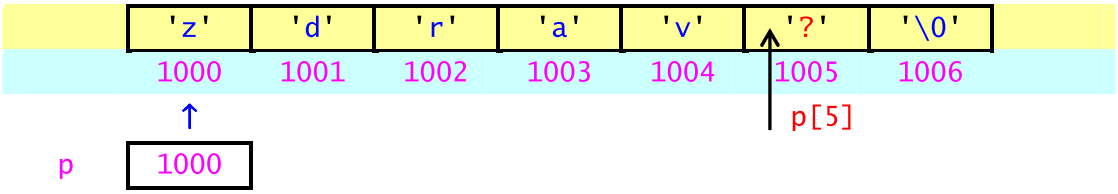
Slika 7.4: Pokazivač na prvi element znakovnoga polja

Pokazivač `p` pokazuje na prvi element promjenjivoga polja `tekst`. Stringovni literal ne smije se mijenjati; zato bi za izravan pokazivač na literal trebalo pisati `const char*`.

Ako sada želimo promijeniti neki znak u znakovnome polju, to možemo učiniti preko pokazivača, primjerice slovo 'o' zamijeniti znakom '?'. Sljedeće dvije naredbe alternativni su zapisi iste operacije; u programu je dovoljno upotrijebiti jednu od njih:

```
1 p[5] = '?';         /* prvi čnain */
2 // *(p + 5) = '?'; /* drugi, istovrijedan način */
```

Rezultat promjene prikazan je na slici 7.5.



Slika 7.5: Promjena elementa znakovnoga polja preko pokazivača

7.6 Aritmetika pokazivača

U jeziku C++ granice dopuštene aritmetike pokazivača strogo su definirane standardom. Pokazivač se smije pomicati samo unutar istoga polja i do položaja neposredno iza njegova posljednjeg elementa. Pokazivač na pojedinačni objekt u tu se svrhu promatra kao pokazivač na polje od jednoga elementa. Izlazak iz dopuštenih granica uzrokuje nedefinirano ponašanje (*undefined behavior* – UB).

Pokazivaču se može pribrojiti ili od njega oduzeti cijeli broj, a dva pokazivača na elemente istoga polja mogu se oduzeti. Pomak ovisi o tipu objekta na koji pokazivač pokazuje:

```
char* znak;
short* p1;
long* p2;
```

Ako pokazivači pokazuju na elemente odgovarajućih polja i napišemo:

```
znak++;
p1++;
p2++;
```

svaki će pokazivač prijeći na sljedeći element svojega polja. Broj bajtova odgovara vrijednostima `sizeof(char)`, `sizeof(short)` i `sizeof(long)` na konkretnoj implementaciji; njihove se veličine ne smiju unaprijed pretpostaviti. Izrazi su ekvivalentni izrazima:

```
znak = znak + 1;
p1 = p1 + 1;
p2 = p2 + 1;
```

Postfiksni operatori ++ i -- imaju viši prioritet od operatora dereferenciranja *. Zato se izrazi

```
*p++;
*p++ = *q++;
```

grupiraju kao `*(p++)` i `*(p++) = *(q++)`. U prvome se izrazu dereferencira trenutačna vrijednost pokazivača `p`, a zatim se `p` pomiče na sljedeći element. Drugi je izraz po učinku ekvivalentan sljedećim naredbama:

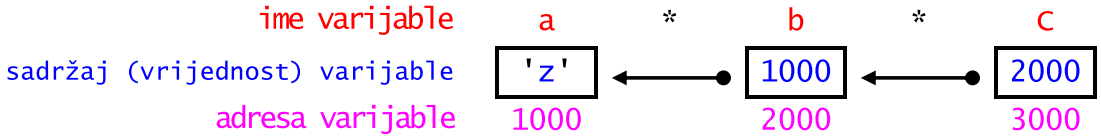
```
*p = *q;
p++;
q++;
```

Kada namjeravana skupina operacija nije odmah očita, zagradama treba izričito pokazati redoslijed primjene operatora.

7.7 Pokazivači na pokazivače

C++ dopušta primjenu pokazivača koji pokazuju na druge pokazivače, što se označava dodatnom zvjezdicom u deklaraciji. Promotrimo to na sljedećim primjerima:

```
char a;
char* b;
char** c;
a = 'z';
b = &a;
c = &b;
```



Slika 7.6: Pokazivač na pokazivač

Varijabla c je pokazivač na pokazivač. O ovoj varijabli može se govoriti na tri različita načina koji odgovaraju različitim vrijednostima:

```
c       // izraz tipa char**, vrijednost 2000
*c      // izraz tipa char*, vrijednost 1000
**c     // izraz tipa char, vrijednost 'z'
```

7.8 Pokazivači tipa void*

Pokazivač tipa `void*` može sadržavati adresu objekta bilo kojega objektnog tipa. Ne može se izravno dereferencirati niti se nad njim smije izvoditi aritmetika pokazivača. Prije dereferenciranja mora se pretvoriti u pokazivač na stvarni tip objekta.

Pokazivač `void*` pretvara se u pokazivač na odgovarajući tip ovako:

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

void povecaj_int(void* podatak)
{
    int* broj = static_cast<int*>(podatak);
    ++(*broj);
}

int main()
{
    int broj = 12;
    povecaj_int(&broj);
    cout << broj << '\n';

    return 0;
}
```

Output

```
13
```

Funkcija mora znati stvarni tip objekta. Određivanje tipa samo prema vrijednosti operatora `sizeof` nije pouzdano jer različiti tipovi mogu imati jednaku veličinu. `sizeof` je operator, a ne funkcija.

Kratki sažetak

U ovome poglavlju upoznali smo pokazivače kao objekte čija vrijednost može biti adresa drugoga objekta ili nul-vrijednost. Objašnjeni su adresni i operator dereferenciranja, inicijalizacija i promjena vrijednosti pokazivača te odnos pokazivača i polja. Prikazana je aritmetika pokazivača i njezina ograničenja, pokazivači na pokazivače te generički pokazivač `void*`. Posebno je naglašeno da se pokazivač smije dereferencirati samo dok pokazuje na postojeći objekt odgovarajućega tipa.

Zapamtite

Nul-pokazivač u suvremenome C++-u zapisuje se ključnom riječi `nullptr`. Pokazivač se smije dereferencirati samo dok pokazuje na postojeći objekt odgovarajućega tipa, a aritmetika pokazivača dopuštena je samo unutar istoga polja (i do položaja neposredno iza njegova kraja).

Pitanja za samoprovjeru

1. Koja je razlika između operatora & i *?
2. Što označava nullptr?
3. U kojem je području dopuštena aritmetika pokazivača?
4. Zašto se pokazivač ne smije dereferencirati nakon isteka života objekta?

8. Reference

Što ćete naučiti?

Nakon ovoga poglavlja moći ćete:

- objasniti što je referenca te kako se deklarira i inicijalizira;
- razlikovati prijenos argumenta po vrijednosti od prijenosa argumenta po referenci;
- primijeniti konstantnu referencu za pristup objektu bez kopiranja i promjene;
- objasniti kada funkcija može vratiti referencu;
- razlikovati osnovna svojstva pokazivača i referenci.

U prethodnim poglavljima upoznali smo varijable, pokazivače i funkcije. Varijable služe za pohranu podataka, pokazivači omogućuju rad s memorijskim adresama, a funkcije olakšavaju organizaciju programa u manje i preglednije cjeline. Pri radu s funkcijama često se javlja potreba da funkcija promijeni vrijednost argumenta koji joj je proslijedio pozivatelj. Jedan od načina ostvarivanja takve komunikacije jest uporaba pokazivača. Međutim, rad s pokazivačima zahtijeva izravan rad s adresama, što početnicima često predstavlja dodatnu poteškoću.

Programski jezik C++ uvodi reference kao način pristupa postojećemu objektu pod drugim imenom. Referenca nije zaseban objekt koji pohranjuje adresu na način pokazivača, premda implementacija vezu može ostvariti pomoću adrese. U ovome poglavlju obrađuju se lvalue-reference, zapisane znakom &.

Najvažnije svojstvo reference jest da nakon povezivanja s varijablom obje oznake predstavljaju isti objekt u memoriji. Promjena vrijednosti preko reference odmah mijenja i vrijednost izvorne varijable, jer se ne radi o kopiji podatka nego o istom objektu kojemu pristupamo različitim imenima.

Reference se vrlo često koriste pri prijenosu argumenata funkcijama jer omogućuju rad s izvornim objektima bez njihova kopiranja. Izbjegavanje kopiranja može povećati učinkovitost programa, osobito kada funkcija obrađuje velike objekte ili složenije strukture podataka.

8.1 Deklaracija i inicijalizacija reference

Referenca se deklarira znakom & uz tip reference. Za razliku od pokazivača, referenca se mora inicijalizirati prilikom deklaracije. Nakon uspostavljanja veze nije ju moguće povezati s drugim objektom.

```
1 int broj = 25;
2 int &r = broj;
```

U prethodnom primjeru varijabla `broj` i referenca `r` predstavljaju isti objekt u memoriji. Promjena vrijednosti preko jednog naziva odmah je vidljiva i preko drugoga.

```
1 cout << broj << endl;
```

```
cout << r << endl;

r = 50;

cout << broj << endl;
cout << r << endl;
```

Nakon izvršavanja programa obje će naredbe za ispis prikazati vrijednost 50. Razlog tome jest što referenca ne sadrži vlastitu kopiju podatka nego omogućuje pristup istoj memorijskoj lokaciji.

Napomena

Za razliku od pokazivača, referenca mora biti inicijalizirana prilikom deklaracije. Sljedeća deklaracija nije dopuštena:

```
int &r;      // pogrešno
```

Referenca se nakon inicijalizacije više ne može povezati s drugom varijablom.

8.2 Reference kao parametri funkcija

Jedna od najvažnijih primjena referenci jest prijenos argumenata funkcijama. Kada se argument prenosi po vrijednosti, funkcija dobiva kopiju podatka. Sve promjene obavljene unutar funkcije odnose se samo na kopiju i ne utječu na izvornu varijablu.

Ako se argument prenosi po referenci, funkcija izravno pristupa izvornoj varijabli. Zbog toga svaka promjena napravljena unutar funkcije ostaje vidljiva i nakon završetka njezina izvođenja.

```
void povecaj(int &x)
{
    x++;
}

int broj = 10;

povecaj(broj);

cout << broj;
```

Program će ispisati vrijednost 11 jer funkcija mijenja izvornu varijablu, a ne njezinu kopiju.

Prijenos argumenta po referenci posebno je koristan kada funkcije obrađuju velike objekte. Budući da se objekt ne kopira, izbjegavaju se vrijeme i memorija potrebni za izradu kopije.

Zapamtite

- Referenca predstavlja drugo ime postojećega objekta.
- Referenca ne stvara kopiju podatka.
- Referenca se mora inicijalizirati prilikom deklaracije.
- Referencu nije moguće naknadno povezati s drugim objektom.
- Referenca ne može imati vrijednost `nullptr`.
- Promjena vrijednosti preko reference mijenja izvorni objekt.

8.2.1 Konstantne reference

U prethodnim primjerima referenca je omogućavala promjenu vrijednosti izvorne varijable. Međutim, postoje situacije u kojima funkcija treba samo pristupiti podatku, ali ga ne smije mijenjati. U takvim se slučajevima koristi konstantna referenca (**`const`** reference).

Dodavanjem ključne riječi **`const`** ispred tipa podataka zabranjuje se promjena vrijednosti objekta preko reference.

```
1 void ispisi(const string &tekst)
2 {
3     cout << tekst << endl;
4 }
```

Funkcija prima referencu na objekt tipa `string`. Budući da je referenca deklarirana kao konstantna, sadržaj objekta nije moguće mijenjati.

Na primjer, sljedeća naredba nije dopuštena:

```
1 void ispisi(const string &tekst)
2 {
3     tekst = "Novi tekst";      // pogrešno
4 }
```

Konstantne reference imaju dvije važne prednosti. Budući da se objekt ne kopira, izvođenje programa je učinkovitije, a istodobno je onemogućena slučajna promjena izvornih podataka. Zbog toga se konstantne reference često koriste pri prijenosu velikih objekata funkcijama.

8.2.2 Funkcije koje vraćaju reference

Osim što se reference koriste kao parametri funkcija, funkcija može kao rezultat vratiti i referencu. Time se pozivatelju omogućuje izravan pristup objektu koji funkcija vraća.

Sljedeći primjer vraća referencu na varijablu koja sadrži veću od dvije zadane vrijednosti.

```
1 int& maksimum(int &a, int &b)
```

```
{
    if (a > b)
    {
        return a;
    }
    else
    {
        return b;
    }
}
```

Funkcija ne vraća kopiju vrijednosti nego referencu na jednu od dvije varijable. Zbog toga se rezultat funkcije može koristiti kao obična varijabla.

```
int x = 10;
int y = 20;

maksimum(x, y) = 100;

cout << x << " " << y;
```

Nakon izvođenja programa vrijednosti varijabli iznose:

Output

```
10 100
```

Budući da je funkcija vratila referencu na varijablu `y`, naredba

```
maksimum(x, y) = 100;
```

ekvivalentna je naredbi

```
y = 100;
```

Na taj način funkcija omogućuje izravnu promjenu objekta na koji vraća referencu.

Napomena

Funkcija ne smije vratiti referencu na lokalnu varijablu.

```
int& funkcija()
{
    int x = 10;
    return x;          // pogrešno
}
```

Lokalna varijabla prestaje postojati nakon povratka iz funkcije. Vraćena referenca tada je viseća; svaka njezina uporaba uzrokuje nedefinirano ponašanje.

Zapamtite

- Konstantna referenca omogućuje pristup objektu bez njegove promjene.
- Konstantne reference izbjegavaju nepotrebno kopiranje velikih objekata.
- Funkcija može vratiti referencu na postojeći objekt.
- Nikada ne vraćajte referencu na lokalnu varijablu funkcije.

Tablica 8.1: Usporedba pokazivača i referenci

Svojstvo	**Pokazivač**	**Referenca**
Definicija	Pokazivač pohranjuje memorijsku adresu drugoga objekta.	Referenca predstavlja drugo ime postojećega objekta.
Deklaracija	U deklaraciji se zapisuje znakom *.	U deklaraciji se zapisuje znakom &.
Inicijalizacija	Može se deklarirati bez inicijalizacije.	Mora se inicijalizirati prilikom deklaracije.
Nul-vrijednost	Može imati vrijednost `nullptr`.	Ne može imati nul-vrijednost.
Promjena povezivanja	Može se naknadno usmjeriti na drugi objekt.	Nakon inicijalizacije ne može se povezati s drugim objektom.
Memorijska adresa	Pokazivač zauzima vlastitu memorijsku lokaciju.	Referenca predstavlja isti objekt kao i izvorni naziv.
Pristup vrijednosti	Za pristup vrijednosti potrebno je dereferenciranje operatorom *.	Vrijednosti se pristupa izravno, bez dereferenciranja.
Primjena	Koristi se pri radu s dinamičkom memorijom, poljima i složenim strukturama podataka.	Najčešće se koristi za prijenos argumenata funkcijama i preopterećenje operatora.
Sigurnost	Zahtijeva pažljivo rukovanje jer može dovesti do pogrešaka (npr. viseći pokazivači).	Ne može biti nul-referenca prema pravilima jezika, ali može postati viseća ako objekt na koji se odnosi prestane postojati.

Kratki sažetak

U ovome poglavlju upoznali smo reference kao mehanizam koji omogućuje pristup postojećemu objektu pod drugim imenom. Objašnjen je način deklariranja i inicijalizacije referenci te njihova primjena pri prijenosu argumenata funkcijama. Prikazane su i konstantne reference, koje omogućuju pristup objektu bez kopiranja i bez mogućnosti njegove promjene preko reference. Na kraju je prikazano kako funkcija može vratiti referencu na postojeći objekt te je naglašeno da vraćanje reference na lokalnu varijablu dovodi do viseće reference i nedefiniranoga ponašanja.

Pitanja za samoprovjeru

1. Što je referenca u programskom jeziku C++?
2. Po čemu se referenca razlikuje od obične varijable?
3. Zašto se referenca mora inicijalizirati prilikom deklaracije?
4. Što se događa kada se preko reference promijeni vrijednost objekta?
5. Koja je osnovna razlika između reference i pokazivača?
6. Koje su prednosti prijenosa argumenta po referenci?
7. Čemu služe konstantne reference?
8. Zašto funkcija koja prima argument kao konstantnu referencu ne može mijenjati objekt preko te reference?
9. Kada je korisno primijeniti konstantnu referencu umjesto prijenosa vrijednošću?
10. Što znači da funkcija vraća referencu?
11. Zašto funkcija ne smije vratiti referencu na lokalnu varijablu?
12. Objasnite na primjeru kada je prikladnije upotrijebiti referencu, a kada pokazivač.

9. Strukture

Što ćete naučiti?

Nakon ovoga poglavlja moći ćete definirati strukture, pristupati njihovim članovima, koristiti pokazivače na strukture, ugniježđene strukture, unije i pobrojene tipove.

9.1 Strukture podataka

Struktura je korisnički definirani tip podataka koji objedinjuje više imenovanih članova. Članovi strukture mogu biti različitih tipova i veličina, a zajedno opisuju jednu cjelinu.

Opći oblik definicije strukture i deklaracije objekta jest:

```
struct ime_strukture
{
    tip1 clan1;
    tip2 clan2;
};

ime_strukture ime_objekta;
```

Ime `ime_strukture` označuje novi tip podataka, a `ime_objekta` objekt toga tipa. Deklaracije unutar vitičastih zagrada određuju članove strukture.

Primjerice, struktura `Proizvod` može objediniti naziv i cijenu proizvoda:

```
struct Proizvod
{
    char ime[30];
    float cijena;
};

Proizvod jabuka;
Proizvod naranca, dinja;
```

Jednom definirana struktura postaje valjan tip podataka, pa se njezini objekti deklariraju na isti način kao objekti osnovnih tipova. Objekti se mogu deklarirati i neposredno uz definiciju strukture:

```
struct Proizvod
{
    char ime[30];
    float cijena;
} jabuka, naranca, dinja;
```

Ime tipa može se izostaviti samo ako poslije ne namjeravamo deklarirati nove objekte te anonimne strukture. U pravilu je preglednije imenovati tip, a objekte deklarirati odvojeno.

Važno je razlikovati tip strukture od objekta strukture. Tip opisuje raspored i tipove članova, a objekt je konkretna cjelina koja sadržava vlastite vrijednosti tih članova. Iz jednoga se tipa može deklarirati proizvoljan broj objekata.

Članu objekta pristupa se operatorom točka (`.`):

```
jabuka.ime
jabuka.cijena
naranca.ime
naranca.cijena
dinja.ime
dinja.cijena
```

Izrazi `jabuka.ime`, `naranca.ime` i `dinja.ime` označuju znakovna polja od 30 elemenata, a izrazi `jabuka.cijena`, `naranca.cijena` i `dinja.cijena` objekte tipa `float`.

Sljedeći program prikazuje uporabu strukture za pohranu podataka o filmu:

```
#define _CRT_SECURE_NO_WARNINGS
#include <cstdlib>
#include <cstring>
#include <iostream>

using namespace std;

struct Film
{
    char naslov[50];
    int godina;
};

void ispisiFilm(const Film& film);

int main()
{
    Film mojFilm, tvojFilm;
    char unos[50];

    strcpy(mojFilm.naslov, "2001: Odiseja u svemiru");
    mojFilm.godina = 1968;

    cout << "Unesi naslov: ";
    cin.getline(tvojFilm.naslov, 50);
    cout << "Unesi godinu: ";
    cin.getline(unos, 50);
    tvojFilm.godina = atoi(unos);

    cout << "Moj omiljeni film je:\n";
    ispisiFilm(mojFilm);
```

```
    cout << "A tvoj:\n";
    ispisiFilm(tvojFilm);

    return 0;
}

void ispisiFilm(const Film& film)
{
    cout << film.naslov << " (" << film.godina << ")\n";
}
```

Output

```
Unesi naslov: Alien
Unesi godinu: 1979
Moj omiljeni film je:
2001: Odiseja u svemiru (1968)
A tvoj:
Alien (1979)
```

Članovi strukture ponašaju se kao i drugi objekti svojega tipa. `tvojFilm.godina` objekt je tipa `int`, a `mojFilm.naslov` znakovno je polje od 50 elemenata. Objekti `mojFilm` i `tvojFilm` objekti su tipa `Film`; funkciji `ispisiFilm` prosljeđuju se konstantnom referencom kako bi se izbjeglo kopiranje cijele strukture.

Strukture se mogu organizirati u polja. Sljedeći program pohranjuje podatke o pet filmova:

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

using namespace std;

constexpr int BROJ_FILMOVA = 5;

struct Film
{
    char naslov[50];
    int godina;
};

void ispisiFilm(const Film& film);

int main()
{
    Film filmovi[BROJ_FILMOVA];
    char unos[50];

    for (int i = 0; i < BROJ_FILMOVA; i++)
    {
        cout << "Unesi naslov: ";
```

```
        cin.getline(filmovi[i].naslov, 50);
        cout << "Unesi godinu: ";
        cin.getline(unos, 50);
        filmovi[i].godina = atoi(unos);
    }

    cout << "\nUnijeli ste ove filmove:\n";
    for (int i = 0; i < BROJ_FILMOVA; i++)
    {
        ispisiFilm(filmovi[i]);
    }

    return 0;
}

void ispisiFilm(const Film& film)
{
    cout << film.naslov << " (" << film.godina << ")\n";
}
```

Output

```
Unesi naslov: Alien
Unesi godinu: 1979
Unesi naslov: Blade Runner
Unesi godinu: 1982
Unesi naslov: Matrix
Unesi godinu: 1999
Unesi naslov: Rear Window
Unesi godinu: 1954
Unesi naslov: Taxi Driver
Unesi godinu: 1975

Unijeli ste ove filmove:
Alien (1979)
Blade Runner (1982)
Matrix (1999)
Rear Window (1954)
Taxi Driver (1975)
```

9.2 Pokazivači na strukturu

Pokazivač na strukturu mora biti deklariran kao pokazivač na odgovarajući tip:

```
struct Film
{
    char naslov[50];
    int godina;
```

```
};

Film film;
Film* pokazivacNaFilm = &film;
```

Objektu na koji pokazivač pokazuje može se pristupiti dereferenciranjem, a zatim se njegovu članu pristupa operatorom točka. Operator strelica (`->`) kraći je zapis iste operacije:

```
pokazivacNaFilm->naslov
(*pokazivacNaFilm).naslov
```

Ta su dva izraza ekvivalentna. Zagrade u drugome izrazu obvezne su jer operator točka ima viši prioritet od operatora dereferenciranja. Izraz `*pokazivacNaFilm.naslov` stoga nije valjan: pokušao bi pristupiti članu `naslov` samoga pokazivača.

Sljedeći program podatke upisuje preko pokazivača na strukturu:

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

using namespace std;

struct Film
{
    char naslov[50];
    int godina;
};

int main()
{
    Film film;
    Film* pokazivacNaFilm = &film;
    char unos[50];

    cout << "Unesi naslov: ";
    cin.getline(pokazivacNaFilm->naslov, 50);
    cout << "Unesi godinu: ";
    cin.getline(unos, 50);
    pokazivacNaFilm->godina = atoi(unos);

    cout << "Upisali ste:\n";
    cout << pokazivacNaFilm->naslov;
    cout << " (" << pokazivacNaFilm->godina << ")\n";

    return 0;
}
```

Output

```
Unesi naslov: Matrix
Unesi godinu: 1999
Upisali ste:
Matrix (1999)
```

Tablica 9.1: Pristup članovima strukture

Izraz	Ekvivalentan izraz	Opis
`film.naslov`		član `naslov` objekta `film`
`pokazivacNaFilm->naslov`	`(*pokazivacNaFilm).naslov`	član `naslov` objekta na koji pokazivač pokazuje

9.3 Ugniježđene strukture

Struktura može biti član druge strukture. Tada se članovima ugniježđenoga objekta pristupa nizom operatora točka i strelica:

```
struct Film
{
    char naslov[50];
    int godina;
};

struct Prijatelj
{
    char ime[50];
    char email[50];
    Film omiljeniFilm;
};

Prijatelj prijatelj1, prijatelj2;
Prijatelj* pokazivacNaPrijatelja = &prijatelj1;
```

Nakon tih deklaracija valjani su, primjerice, sljedeći izrazi:

```
prijatelj1.ime
prijatelj2.omiljeniFilm.naslov
prijatelj1.omiljeniFilm.godina
pokazivacNaPrijatelja->omiljeniFilm.godina
```

Strukture u C++-u mogu sadržavati podatkovne i funkcijske članove te imati prava pristupa, kao i klase. Temeljna je razlika u zadanim pravima pristupa: članovi i bazne klase strukture prema zadanim su postavkama javni (`public`), a članovi i bazne klase klase privatni (`private`). Klase će biti obrađene u zasebnome poglavlju.

9.4 Korisnički definirana imena tipova

Postojećemu tipu može se dodijeliti drugo ime. U suvremenome C++-u za to se najčešće koristi deklaracija **using**:

```
using Znak = char;
using Rijec = char*;
using Ime = char[50];

Znak znak1, znak2;
Rijec pokazivacNaZnak;
Ime ime;
```

Stariji, još uvijek valjan zapis koristi ključnu riječ **typedef**:

```
typedef char Znak;
typedef char* Rijec;
typedef char Ime[50];
```

Oba zapisa uvode novo ime za postojeći tip; ne stvaraju nov, zaseban tip.

9.5 Unije

Unija (**union**) omogućuje da više članova dijeli isti memorijski prostor. Deklaracija unije nalikuje deklaraciji strukture, ali se vrijednost u pravilu pohranjuje samo u jednome članu:

```
union MojTip
{
    char znak;
    int cijeliBroj;
    float realniBroj;
};

MojTip vrijednost;
```

Članovima se pristupa na isti način kao članovima strukture:

```
vrijednost.znak
vrijednost.cijeliBroj
vrijednost.realniBroj
```

Svi članovi unije započinju na istoj memorijskoj adresi. Veličina unije dovoljna je za njezin najveći član, uz moguću dodatnu pohranu zbog poravnanja. U svakome je trenutku u pravilu aktivan član u koji je posljednji put zapisana vrijednost. Čitanje drugoga, neaktivnog člana najčešće uzrokuje nedefinirano ponašanje i ne treba ga koristiti za pretvorbu prikaza podatka.

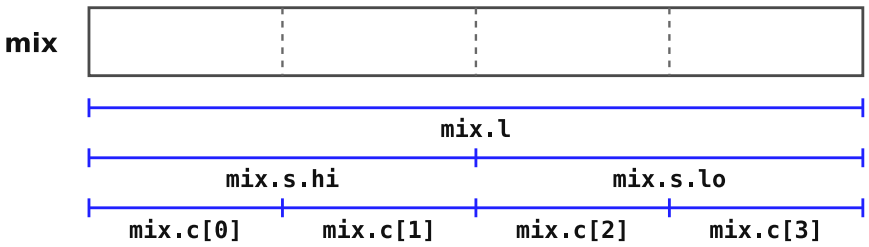
Sljedeći primjer ilustrira preklapanje članova unije:

```
struct dva_kratka_t
{
    short hi;
    short lo;
};

union mix_t
{
    long l;
    dva_kratka_t s;
    char c[sizeof(long)];
};

mix_t mix;
```

Raspored bajtova, veličina tipova i redoslijed članova `mix.s.hi` i `mix.s.lo` ovise o implementaciji. Slika 9.1 zato prikazuje samo jednu moguću izvedbu preklapanja memorije.



Slika 9.1: Primjer preklapanja članova unije

9.6 Anonimne unije

Unija može biti imenovani ili anonimni član strukture.

U prvome primjeru unija ima ime `cijena`, pa se njezinim članovima pristupa preko toga imena:

```
struct Knjiga
{
    char naslov[50];
    char autor[50];
    union
    {
        float euri;
        int centi;
    } cijena;
};

Knjiga knjiga;
knjiga.cijena.euri = 19.99F;
```

U drugome je primjeru unija anonimna, pa se njezinim članovima pristupa izravno kao članovima strukture:

```
struct Knjiga
{
    char naslov[50];
    char autor[50];
    union
    {
        float euri;
        int centi;
    };
};

Knjiga knjiga;
knjiga.euri = 19.99F;
```

U oba primjera istodobno treba upotrebljavati samo onaj član unije koji je aktivan.

9.7 Pobrojeni tipovi podataka

Kada varijabla predstavlja jedan element ograničenoga pojmovnog skupa, tim se vrijednostima mogu pridijeliti lako pamtljiva imena. Za to se koriste pobrojeni tipovi (*enumerations*):

```
enum Boja
{
    crna,
    plava,
    zelena,
    cijan,
    crvena,
    ljubicasta,
    zuta,
    bijela
};
```

Ime `Boja` označuje novi tip, a imena unutar vitičastih zagrada njegove pobrojene vrijednosti. Objekt toga tipa može se inicijalizirati i usporediti s jednom od navedenih vrijednosti:

```
Boja mojaBoja = plava;

if (mojaBoja == plava)
{
    cout << "Moja boja je plava!";
}
```

Ako vrijednosti nisu izričito zadane, prvoj se pobrojenoj vrijednosti pridružuje broj 0, sljedećoj 1 i tako dalje. Početna se vrijednost može zadati u deklaraciji:

```
enum Mjesec
{
    sijecanj = 1,
    veljaca,
    ozujak,
    travanj,
    svibanj,
    lipanj,
    srpanj,
    kolovoz,
    rujan,
    listopad,
    studeni,
    prosinac
};

Mjesec mjesec = sijecanj;
```

Tada vrijednosti od `sijecanj` do `prosinac` odgovaraju cijelim brojevima od 1 do 12.

Napomena

U suvremenome C++-u prednost se daje pobrojenomu tipu **`enum class`** kada implicitna pretvorba u cijeli broj nije potrebna. Njegove se vrijednosti pišu kvalificirano, primjerice `Boja::plava`, čime se smanjuje mogućnost sukoba imena i slučajnih brojčanih usporedbi.

Kratki sažetak

Struktura objedinjuje imenovane članove koji mogu biti različitih tipova, a svaki objekt strukture sadržava vlastite vrijednosti tih članova. Članovima se pristupa operatorom točka ili, preko pokazivača, operatorom strelica. Strukture mogu biti ugniježđene. Unija svojim članovima daje zajednički memorijski prostor, dok pobrojeni tip definira zatvoren skup imenovanih vrijednosti.

Pitanja za samoprovjeru

1. Koja je razlika između tipa strukture i objekta strukture?
2. Kako se članu strukture pristupa preko pokazivača?
3. Kako se pristupa članu ugniježđene strukture?
4. Što deklaracija **`using`** uvodi u program?
5. Po čemu se unija razlikuje od strukture?
6. Koju prednost ima **`enum class`**?

10. Organizacija i dinamičko upravljanje memorijom

Što ćete naučiti?

Nakon ovoga poglavlja moći ćete razlikovati opseg imena, životni vijek objekta i trajanje pohrane, objasniti tipičnu ulogu *stacka*, područja statičkih podataka i *heapa*, pratiti nastanak i nestanak *stack framea* funkcije, razlikovati pokazivač od dinamičkoga objekta na koji pokazuje, pravilno primijeniti izraze `new T`, `new T[n]`, `delete p` i `delete[] p` te prepoznati curenje memorije, viseći pokazivač, dvostruko oslobađanje i pristup izvan granica polja. Moći ćete i objasniti kako objekti klasa `string` i `vector` automatski upravljaju dodatnom memorijom.

Problem koji rješavamo

U programu se istodobno pojavljuju globalne i lokalne varijable, parametri funkcija, polja, pokazivači, objekti klasa i dinamički stvoreni objekti. Za svaki od njih programer mora znati odgovoriti na četiri pitanja:

1. Što je tom naredbom stvoreno?
2. Gdje se objekt uobičajeno pohranjuje?
3. Koliko dugo objekt postoji?
4. Tko i kada objekt uništava odnosno oslobađa njegovu memoriju?

Odgovori na ta pitanja povezuju gradivo o varijablama, funkcijama, poljima, pokazivačima, klasama i standardnim spremnicima.

U dosadašnjim programima količina potrebne memorije uglavnom je bila poznata pri prevođenju. Međutim, broj objekata ili njihov potreban životni vijek katkad se mogu odrediti tek tijekom izvođenja, primjerice nakon korisničkoga unosa, učitavanja podataka iz datoteke ili rezultata nekoga izračuna. Tada se može primijeniti dinamička dodjela memorije. Pritom nije dovoljno znati samo kako pribaviti memoriju: mora biti jasno i tko je odgovoran za objekt, koliko dugo on postoji te kada i na koji način njegova pohrana mora biti oslobođena.

10.1 Objekt, memorija i životni vijek

Objekt u C++-u zauzima određeni dio memorije, ima tip i postoji tijekom određenoga vremenskog razdoblja. Objekt ne mora imati ime. Primjerice, izraz `new int(25)` stvara objekt tipa `int` bez vlastitoga imena, a pristupa mu se preko pokazivača.

Pri razmatranju memorije važno je razlikovati tri povezana, ali različita pojma:

- **opseg imena** određuje u kojem se dijelu izvornoga koda neko ime može upotrijebiti;
- **životni vijek objekta** razdoblje je od početka njegova postojanja do njegova uništenja;
- **trajanje pohrane** opisuje koliko dugo memorija namijenjena objektu ostaje raspoloživa.

Mjesto deklaracije ne određuje uvijek i mjesto pohrane. Lokalna varijabla deklarirana ključnom riječju `static` ima lokalni opseg imena, ali statičko trajanje pohrane. Njezino je ime dostupno samo unutar funkcije, dok sam objekt postoji do završetka programa.

Napomena

C++ standard ne zahtijeva da implementacija ima područja memorije koja se doslovno zovu *stack* i *heap*. Standard propisuje ponašanje programa, trajanje pohrane i životni vijek objekata. Stack, programski kod, područja statičkih podataka i heap uobičajen su način na koji operacijski sustav, prevoditelj i izvršno okruženje ostvaruju ta pravila. Zato ćemo ih upotrebljavati kao praktičan model, ali ćemo uvijek razlikovati pravilo jezika od tipične implementacije.

10.2 Četiri vrste trajanja pohrane

C++ razlikuje četiri osnovne vrste trajanja pohrane. Za razumijevanje početničkih programa najvažnije su automatsko, statičko i dinamičko trajanje.

Tablica 10.1: Vrste trajanja pohrane u C++-u

Trajanje	Tipični primjeri	Početak i završetak
automatsko	lokalne varijable, parametri funkcije, lokalna polja i lokalni objekti klasa bez `static`	pohrana se pribavlja pri ulasku u blok ili pozivu funkcije, a objekt se automatski uništava pri izlasku iz bloka
statičko	globalni objekti, objekti namespacea, lokalni `static` objekti i statički podatkovni članovi	pohrana postoji tijekom cijeloga izvođenja programa; lokalni `static` objekt inicijalizira se pri prvome prolasku kroz njegovu deklaraciju
dinamičko	objekti stvoreni izrazima `new` i `new[]`	objekt postoji od uspješnoga stvaranja do odgovarajućega `delete` ili `delete[]`
nitno	objekti deklarirani s `thread_local`	svaka nit ima svoj objekt, koji postoji tijekom izvođenja te niti

Objekt s automatskim trajanjem pohrane ne mora programer ručno osloboditi. Njegov se destruktor, ako ga tip ima, poziva automatski pri izlasku iz bloka. Dinamički stvoren objekt nema takvu vezu s blokom u kojem je stvoren. Ako se njime upravlja sirovim pokazivačem, programer mora osigurati njegovo pravodobno i točno jedno oslobađanje.

10.3 Tipična područja memorije programa

Pojednostavljeni prikaz na slici 10.1 pomaže povezati vrste trajanja pohrane s uobičajenom organizacijom procesa.



Slika 10.1: Tipična organizacija memorije procesa

- **Područje programskoga koda** sadržava prevedene strojne naredbe. U pravilu nije namijenjeno promjeni tijekom izvođenja.
- **Područje statičkih podataka** sadržava objekte sa statičkim trajanjem pohrane. Implementacije često odvojeno pohranjuju inicijalizirane podatke, podatke inicijalizirane nulom i sadržaj koji je samo za čitanje.
- **Stack** uobičajeno sadržava *stack frameove* trenutačno pozvanih funkcija. U njima se često nalaze parametri, lokalni objekti automatskoga trajanja i podatci potrebni za povratak iz funkcije.
- **Heap** područje je iz kojega se uobičajeno pribavlja pohrana za dinamičke objekte. C++ standard za pohranu kojom upravljaju izrazi `new` i `delete` rabi naziv *free store*, iako se pojmovi *free store* i *heap* ne moraju u svakoj implementaciji potpuno podudarati.

Na suvremenim operacijskim sustavima adrese koje program vidi najčešće su virtualne adrese. Operacijski sustav preslikava ih na fizičku memoriju i po potrebi upravlja stranicama memorije. Za ispravno pisanje početničkih programa nije potrebno poznavati pojedinosti toga preslikavanja, ali je važno znati da adresa ispisana programom nije jednostavan redni broj ćelije fizičkoga RAM-a.

10.4 Što se gdje smješta?

Sljedeći program sadržava nekoliko tipičnih objekata različitoga trajanja pohrane:

```
#include <iostream>

using namespace std;

int globalnaVrijednost = 10;

void funkcija(int parametar)
{
    static int brojPoziva = 0;
    int lokalnaVrijednost = 20;
    int lokalnoPolje[3]{1, 2, 3};

    ++brojPoziva;
    cout << parametar + lokalnaVrijednost
         + lokalnoPolje[0] + brojPoziva << '\n';
}

int main()
{
    int broj = 5;
    funkcija(broj);

    return 0;
}
```

Objekt `globalnaVrijednost` ima statičko trajanje pohrane. Parametar `parametar`, objekti `lokalnaVrijednost` i `lokalnoPolje` te objekt `broj` imaju automatsko trajanje pohrane i u uobičajenoj se implementaciji nalaze u odgovarajućim *stack frameovima*.

Objekt `brojPoziva` poseban je slučaj. Njegovo je ime lokalno funkciji `funkcija`, ali objekt ima statičko trajanje pohrane. Inicijalizira se samo jednom i zadržava vrijednost između poziva funkcije. Zato se uobičajeno ne nalazi u *stack frameu* te funkcije.

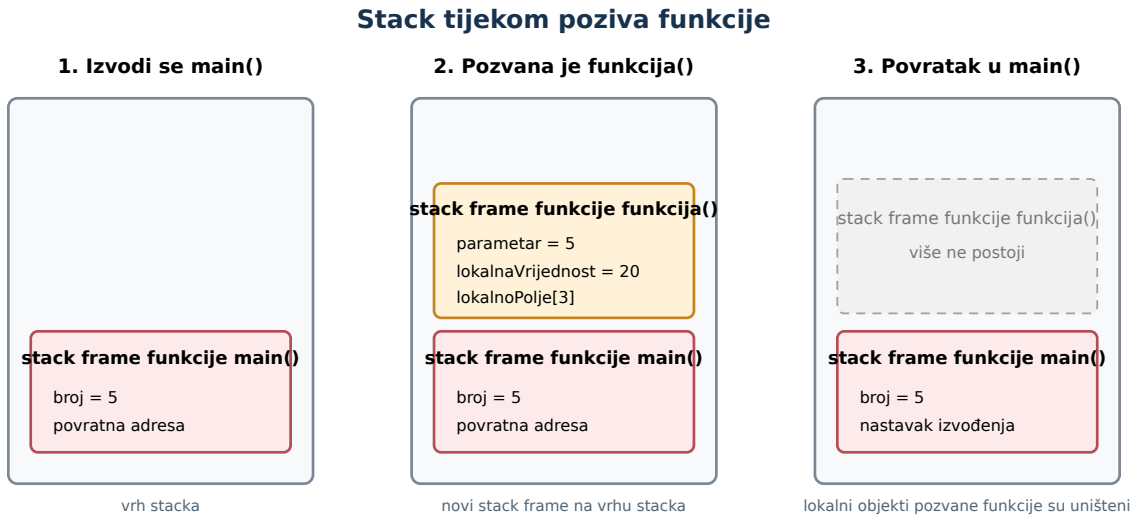
Tablica 10.2: Tipično mjesto i način uništavanja različitih objekata

Primjer	Trajanje	Uobičajeno mjesto	Uništavanje
globalna varijabla	statičko	statički podatci	automatski pri završetku programa
lokalna varijabla	automatsko	*stack frame*	automatski pri izlasku iz bloka
parametar funkcije	automatsko	*stack frame*	automatski pri povratku iz funkcije
lokalno polje fiksne veličine	automatsko	*stack frame*	automatski pri izlasku iz bloka
lokalna varijabla `static`	statičko	statički podatci	automatski pri završetku programa
lokalni pokazivač	automatsko	*stack frame*	pokazivač se uništava pri izlasku iz bloka; objekt na koji pokazuje time se ne mora uništiti
objekt stvoren s `new`	dinamičko	*free store* (uobičajeno na *heapu*)	odgovarajući `delete`
objekt stvoren s `new[]`	dinamičko	*free store* (uobičajeno na *heapu*)	odgovarajući `delete[]`
lokalni objekt klase	automatsko	*stack frame*	automatski poziv destruktora pri izlasku iz bloka
globalni objekt klase	statičko	statički podatci	automatski poziv destruktora pri završetku programa

10.5 Stack i pozivi funkcija

Pri pozivu funkcije uobičajeno se stvara njezin **stack frame**. On može sadržavati parametre, lokalne objekte, adresu na koju se izvođenje treba vratiti te druge podatke potrebne implementaciji. Točan raspored ovisi o procesoru, prevoditelju, optimizaciji i pozivnoj konvenciji.

Slika 10.2 prikazuje pojednostavljeni tijek poziva funkcije iz prethodnoga programa.



Slika 10.2: Nastanak i nestanak stack framea funkcije

Nakon povratka iz funkcije njezini lokalni objekti automatskoga trajanja više ne postoje. Zato funkcija ne smije vratiti pokazivač ili referencu na takav lokalni objekt:

```
int* stvoriBroj()
{
    int broj = 25;
    return &broj;     // pogrešno: broj uskoro prestaje postojati
}
```

Povratna vrijednost sadržavala bi adresu na kojoj je objekt nekada postojao. Nakon završetka funkcije taj je pokazivač **viseći pokazivač**, a njegovo dereferenciranje uzrokuje nedefinirano ponašanje.

10.5.1 Rekurzija i stack overflow

Svaki rekurzivni poziv funkcije uobičajeno dobiva novi *stack frame*. Ako rekurzija nema ispravan uvjet završetka ili je dubina poziva prevelika, *stack* može ostati bez raspoloživa prostora. Posljedica je **stack overflow**. Veliki lokalni objekti, primjerice vrlo velika polja, također mogu iscrpiti ograničeni prostor *stacka*.

Napomena

Optimizirajući prevoditelj može neke lokalne vrijednosti držati u registrima, potpuno ukloniti objekt koji nije potreban ili ugraditi poziv funkcije. Zato prikaz *stacka* opisuje uobičajeni konceptualni model, a ne obećava da će svaki objekt u strojnom programu imati zasebnu ćeliju upravo na *stacku*.

10.6 Free store, heap i izraz new

Dinamičko trajanje pohrane potrebno je kada se veličina ili životni vijek objekta ne mogu prikladno vezati uz jedan blok programa. Izraz `new` pribavlja dovoljno prostora, stvara objekt zadanoga tipa i vraća pokazivač na njega. Opći oblici za stvaranje jednoga objekta i polja objekata mogu se zapisati ovako:

```
T* pokazivac = new T;
T* polje = new T[brojElemenata];
```

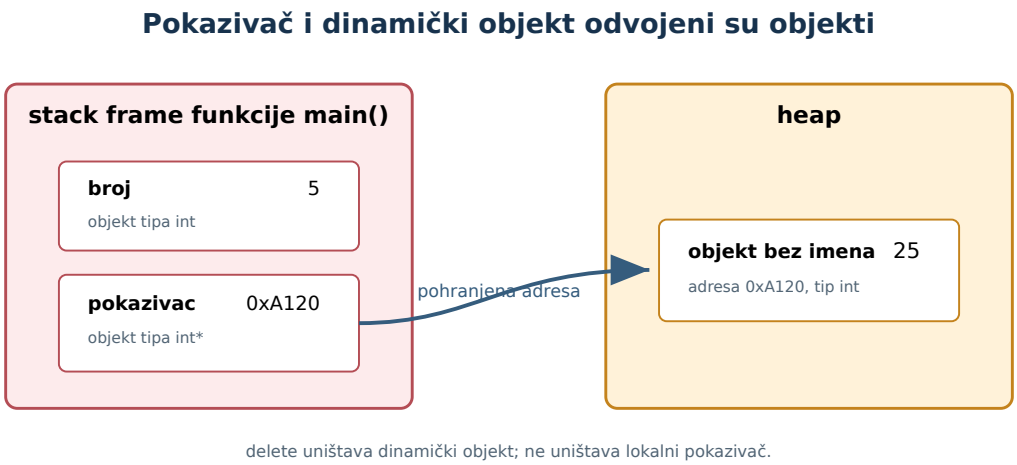
U prvome se slučaju stvara jedan dinamički objekt tipa `T`, a u drugome dinamičko polje od `brojElemenata` uzastopnih objekata istoga tipa. Konkretan primjer jednoga dinamičkog objekta glasi:

```
int* pokazivac = new int(25);
```

Tom naredbom nastaju dva odvojena objekta:

- lokalni objekt `pokazivac` tipa `int*`, koji pohranjuje adresu;
- dinamički objekt tipa `int` s početnom vrijednošću 25.

Njihov odnos prikazan je na slici 10.3.



Slika 10.3: Lokalni pokazivač i dinamički objekt na koji pokazuje

Preciznije, *new-izraz* najprije poziva odgovarajuću alokacijsku funkciju, primjerice `operator new`, kako bi pribavio pohranu, a zatim u toj pohrani inicijalizira objekt. Za objekt klase to znači poziv konstruktora. Slično tomu, *delete-izraz* najprije uništava objekt, odnosno poziva njegov destruktor, a zatim poziva odgovarajuću dealokacijsku funkciju. U početničkom se tekstu izrazi `new` i `delete` često sažeto nazivaju operatorima, ali ove dvije faze važne su za potpuno razumijevanje postupka.

Izlaskom iz funkcije uništava se lokalni pokazivač, ali se time ne uništava dinamički objekt na koji je

pokazivao. Ako je to bio jedini pokazivač koji je sadržavao njegovu adresu, objekt više nije dostupan, a njegova memorija ostaje zauzeta. To je curenje memorije.

10.6.1 Inicijalizacija dinamičkoga objekta

Različiti zapisi ne daju uvijek iste početne vrijednosti:

```
int* prvi = new int;        // vrijednost tipa int nije inicijalizirana
int* drugi = new int();     // inicijaliziran nulom
int* treci = new int{};     // inicijaliziran nulom
int* cetvrti = new int(25); // inicijaliziran vrijednošću 25
```

Čitanje neinicijalizirane vrijednosti objekta `*prvi` nije dopušteno. Za korisnički definirane tipove izraz `new T(...)` poziva odgovarajući konstruktor.

10.6.2 Neuspjela dodjela

Uobičajeni oblik operatora `new` pri neuspjeloj dodjeli baca iznimku `std::bad_alloc`. Ne vraća automatski `nullptr`. Poseban oblik s `std::nothrow` pri neuspjehu vraća nul-pokazivač:

```
#include <new>

int* polje = new (std::nothrow) int[vrloVelikBroj];

if (polje == nullptr)
{
    // dodjela nije uspjela
}
```

U suvremenome C++-u uobičajenije je dopustiti da se iznimka obradi na razini programa koja zna kako oporaviti postupak, nego posvuda primjenjivati oblik `std::nothrow`.

10.7 Delete-izrazi `delete` i `delete[]`

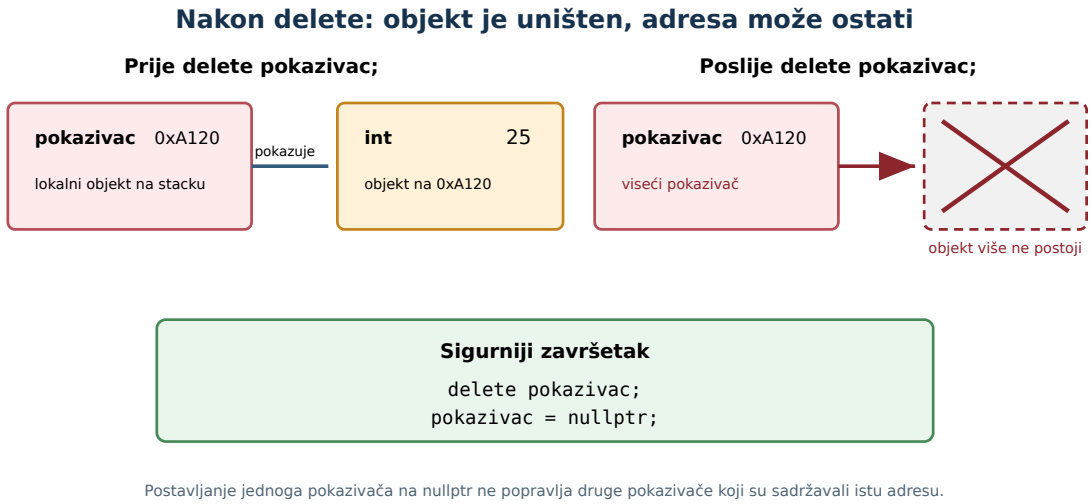
Dinamički objekt stvoren izrazom `new T` uništava se delete-izrazom `delete p`. Dinamičko polje stvoreno izrazom `new T[n]` uništava se delete-izrazom `delete[] p`:

```
delete pokazivac;
pokazivac = nullptr;

delete[] polje;
polje = nullptr;
```

Oblik oslobađanja mora odgovarati obliku stvaranja. Miješanje `new` i `delete[]` ili `new[]` i `delete` uzrokuje nedefinirano ponašanje.

Izraz **delete** najprije završava životni vijek objekta, što za objekt klase znači poziv destruktora, a zatim oslobađa njegovu pohranu. Sam objekt pokazivača pritom se ne uništava. U njemu zato može ostati stara, više nevaljana adresa, kao na slici 10.4.



Slika 10.4: Nastanak visećega pokazivača nakon oslobađanja objekta

Postavljanje pokazivača na `nullptr` nakon oslobađanja dobra je obrambena mjera, ali ne rješava sve probleme. Ako dva pokazivača sadržavaju adresu istoga objekta, promjena jednoga ne mijenja drugi:

```
int* prvi = new int(25);
int* drugi = prvi;

delete prvi;
prvi = nullptr;

// drugi i dalje sadržava staru adresu i ne smije se dereferencirati
```

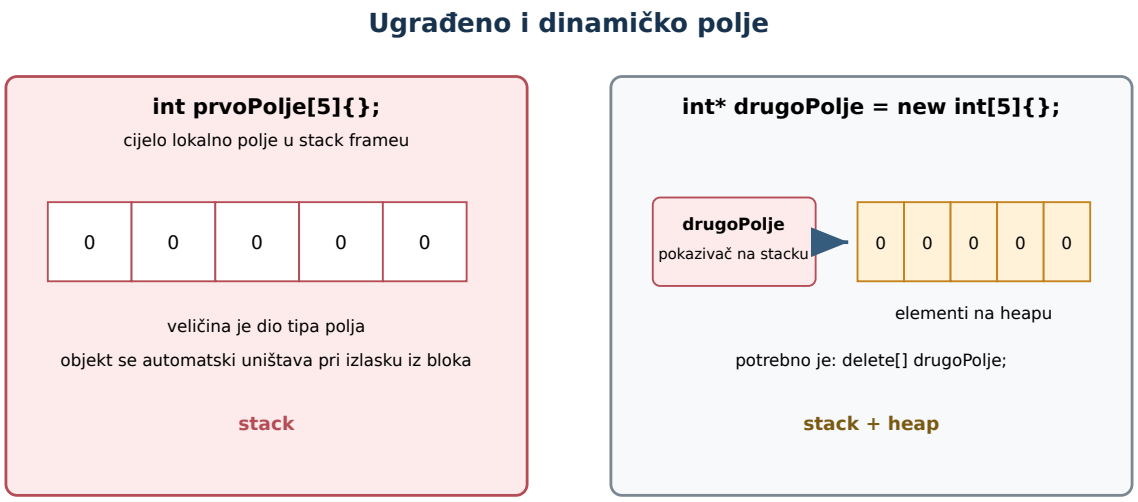
Poziv **delete** `nullptr;` dopušten je i nema učinka. Zato prije operatora **delete** nije potrebna posebna provjera pokazivača.

10.8 Ugrađena i dinamička polja

Ugrađeno lokalno polje i dinamičko polje mogu imati iste elemente, ali se razlikuju po trajanju pohrane i odgovornosti za oslobađanje:

```
int prvoPolje[5]{};
int* drugoPolje = new int[5]{};
```

Prvo je cijeli objekt polja automatskoga trajanja. Druga naredba stvara lokalni pokazivač automatskoga trajanja i odvojeno dinamičko polje od pet elemenata, kao na slici 10.5.



Slika 10.5: Usporedba lokalnoga ugrađenog polja i dinamičkoga polja

Veličina ugrađenoga polja u standardnom C++-u mora biti konstantan izraz poznat pri prevođenju. Veličina dinamičkoga polja može se odrediti tijekom izvođenja:

```
#include <iostream>

using namespace std;

int main()
{
    int brojElemenata;

    cout << "Koliko brojeva zelite unijeti? ";
    cin >> brojElemenata;

    if (brojElemenata <= 0)
    {
        cout << "Broj elemenata mora biti pozitivan.\n";
        return 1;
    }

    long* brojevi = new long[brojElemenata]{};

    for (int indeks = 0; indeks < brojElemenata; ++indeks)
    {
        cout << "Unesite broj: ";
        cin >> brojevi[indeks];
    }

    cout << "Unijeli ste: ";
    for (int indeks = 0; indeks < brojElemenata; ++indeks)
```

```
    {
        cout << brojevi[indeks];
        if (indeks < brojElemenata - 1)
        {
            cout << ", ";
        }
    }
    cout << '\n';
    cout << "Prvi element je brojevi[0] = " << brojevi[0] << '\n';

    delete[] brojevi;
    brojevi = nullptr;

    return 0;
}
```

Za unos pet vrijednosti jedan mogući izlaz programa jest:

Output

```
Koliko brojeva zelite unijeti? 5
Unesite broj: 1234
Unesite broj: 2345
Unesite broj: 3456
Unesite broj: 4567
Unesite broj: 5678
Unijeli ste: 1234, 2345, 3456, 4567, 5678
Prvi element je brojevi[0] = 1234
```

Broj elemenata određuje se tijekom izvođenja, ali nije neograničen. Ograničen je raspoloživom memorijom, najvećom veličinom objekta koju implementacija može podržati te rasponom tipa kojim program prikazuje broj elemenata. Dodjela može ne uspjeti čak i kada je uneseni broj pozitivan.

Napomena

Dinamičko polje stvoreno izrazom `new T[n]` ne zna se samo povećati. Za promjenjiv broj elemenata u suvremenom je C++-u u pravilu bolji izbor `vector`. Ručna dinamička polja ipak treba razumjeti jer jasno pokazuju odnos adrese, životnoga vijeka i odgovornosti za memoriju.

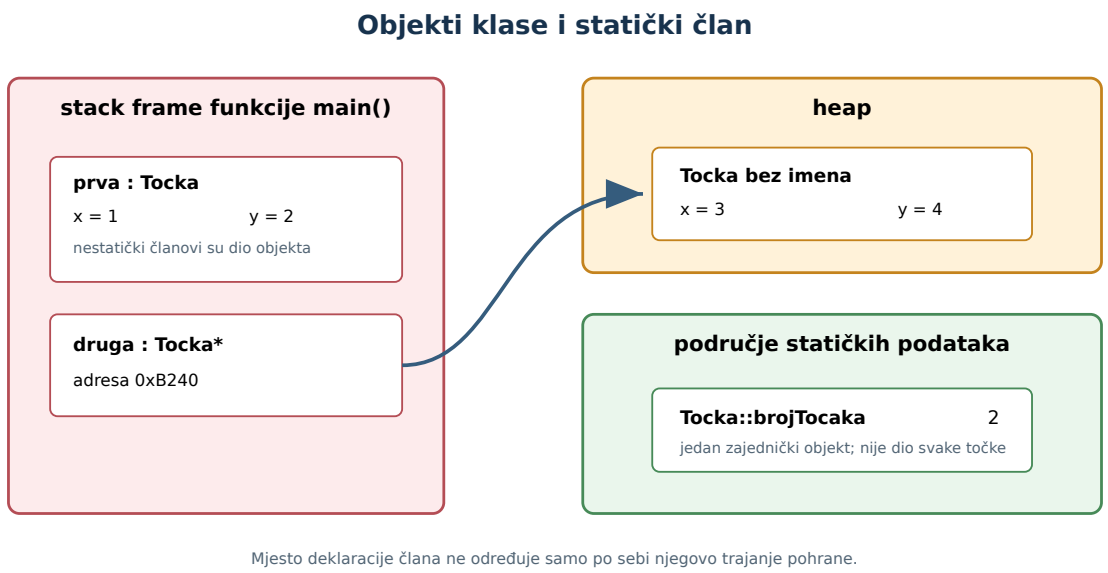
10.9 Objekti klasa u memoriji

Mjesto pohrane objekta klase određuje se na isti način kao i za ugrađene tipove. Lokalni objekt bez ključne riječi `static` ima automatsko trajanje, a objekt stvoren izrazom `new` dinamičko trajanje:

```
class Tocka
{
public:
```

```
4       Tocka(int noviX, int noviY) : x(noviX), y(noviY)
5       {
6           ++brojTocaka;
7       }
8
9       ~Tocka()
10      {
11          --brojTocaka;
12      }
13
14      static int brojTocaka;
15
16  private:
17      int x;
18      int y;
19  };
20
21  int Tocka::brojTocaka = 0;
22
23  int main()
24  {
25      Tocka prva(1, 2);
26      Tocka* druga = new Tocka(3, 4);
27
28      delete druga;
29      druga = nullptr;
30
31      return 0;
32  }
```

Nestatički članovi `x` i `y` dio su svakoga pojedinog objekta klase `Tocka`. Statički član `brojTocaka` nije dio svakoga objekta. Postoji jedan zajednički objekt sa statičkim trajanjem pohrane, prikazan na slici 10.6.



Slika 10.6: Lokalni i dinamički objekt klase te statički podatkovni član

Pri izlasku iz funkcije destruktor objekta `prva` poziva se automatski. Destruktor dinamičkoga objekta poziva se izvođenjem **`delete`** `druga`. Ako se taj operator izostavi, ne izostaje samo oslobađanje memorije: izostaje i uredan poziv destruktora objekta.

10.9.1 Veličina, poravnanje i padding

Veličina objekta klase nije nužno jednaka zbroju veličina njegovih članova. Procesor može zahtijevati da pojedini podatci počinju na prikladno poravnatoj adresi. Prevoditelj zato između članova ili na kraju objekta može umetnuti neimenovane bajtove za popunu (engl. *padding*). Operator **`sizeof`** vraća stvarnu veličinu objekta u bajtovima, uključujući takvu popunu.

Statički podatkovni član ne ulazi u veličinu svakoga objekta klase jer nije njegov sastavni dio. Funkcijski članovi također se ne kopiraju u svaki objekt; prevedeni kod funkcija pohranjuje se odvojeno. Kod klasa s virtualnim funkcijama implementacija obično dodaje skriveni pokazivač, pa i on može utjecati na veličinu objekta.

10.10 string i vector: objekt nije isto što i njegovi elementi

Razmotrimo lokalne objekte standardnih klasa:

```
#include <string>
#include <vector>

using namespace std;

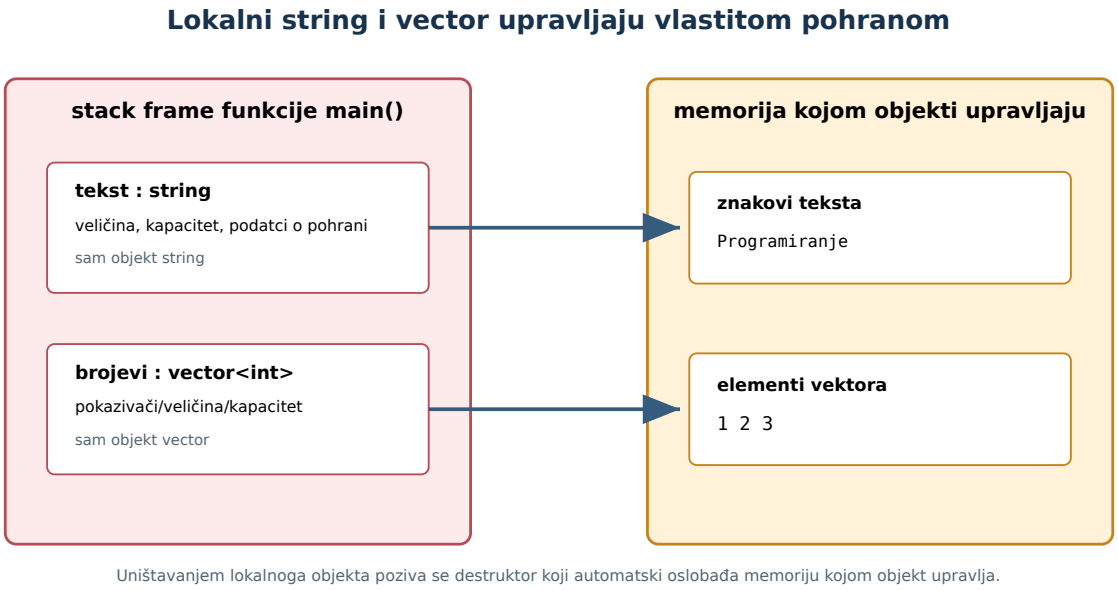
int main()
```

```
{
    string tekst = "Programiranje";
    vector<int> brojevi{1, 2, 3};

    return 0;
}
```

Objekti `tekst` i `brojevi` lokalni su objekti automatskoga trajanja. Uobičajeno se nalaze u *stack frameu* funkcije `main`. Svaki od njih sadržava podatke potrebne za upravljanje svojim elementima, primjerice veličinu, kapacitet i podatak o pohrani.

Elementi `vectora` u pravilu se nalaze u odvojeno dinamički pribavljenoj memoriji. Objekt `string` također može pribaviti dinamičku memoriju za znakove, iako mnoge implementacije kraće tekstove pohranjuju izravno unutar samoga objekta. Ta se optimizacija naziva *small string optimization* i nije zajamčena standardom.



Slika 10.7: Lokalni objekti standardnih klasa i memorija kojom upravljaju

Pri završetku funkcije automatski se pozivaju destruktori lokalnih objekata. Destruktor `vectora` oslobađa pohranu njegovih elemenata, a destruktor `stringa` pohranu kojom string upravlja. Programer zato ne smije pozvati **`delete`** nad adresom njihovih elemenata.

To je primjer načela **RAII** (engl. *Resource Acquisition Is Initialization*): upravljanje resursom vezano je uz životni vijek objekta. Objekt pribavlja resurs pri stvaranju i oslobađa ga u destruktoru. Iznimka, prijevremeni **`return`** ili drugi izlazak iz bloka tada ne preskaču oslobađanje resursa.

Zapamtite

Lokalni `vector` može biti na *stacku*, a njegovi elementi na *heapu*. Rečenica „vector je na heapu" zato je neprecizna: treba posebno navesti sam objekt spremnika i pohranu njegovih elemenata.

10.11 Vlasništvo nad dinamičkim objektom

Pokazivač samo pohranjuje adresu. Iz samoga tipa `T*` nije vidljivo tko je odgovoran za uništavanje objekta. Tu odgovornost nazivamo **vlasništvom**.

Ako funkcija samo privremeno pristupa objektu koji pripada pozivatelju, pokazivač nije vlasnik i ne smije pozvati `delete`. Ako je funkcija preuzela vlasništvo nad dinamičkim objektom, mora biti jasno određeno kada će ga osloboditi ili komu će vlasništvo predati.

Sirovi pokazivači ne izražavaju tu razliku dovoljno jasno. U suvremenome C++-u isključivo se vlasništvo najčešće prikazuje klasom `unique_ptr`, a zajedničko vlasništvo, kada je doista potrebno, klasom `shared_ptr`. Pametni pokazivači obrađuju se u zasebnom poglavlju; ovdje je bitna temeljna poruka: svaki dinamički objekt mora imati jasan i siguran put do točno jednoga odgovarajućeg uništenja.

10.12 Najčešće pogreške pri radu s memorijom

Mnoge memorijske pogreške uzrokuju **nedefinirano ponašanje**. To ne znači da će program obvezno odmah javiti pogrešku. Može se prividno ispravno izvoditi, ispisati pogrešan rezultat, srušiti se tek poslije ili se drukčije ponašati nakon promjene prevoditelja i optimizacije.

10.12.1 Curenje memorije

Adresa dinamičkoga objekta izgubljena je prije njegova oslobađanja:

```
int* pokazivac = new int(25);
pokazivac = new int(50); // adresa prvoga objekta je izgubljena
```

Prvi objekt i dalje zauzima memoriju, ali je njegova adresa izgubljena pa mu program više ne može pristupiti.

10.12.2 Pristup nakon oslobađanja

```
int* pokazivac = new int(25);
delete pokazivac;

cout << *pokazivac; // pogrešno: objekt više ne postoji
```

Činjenica da pokazivač još sadržava neku adresu ne znači da na toj adresi i dalje postoji valjan objekt odgovarajućega tipa.

10.12.3 Dvostruko oslobađanje

```
int* pokazivac = new int(25);
delete pokazivac;
delete pokazivac; // pogrešno
```

Isti dinamički objekt smije se osloboditi samo jednom. Drugi poziv ne „provjerava" je li memorija već oslobođena, nego uzrokuje nedefinirano ponašanje.

10.12.4 Pogrešan oblik operatora delete

```
int* polje = new int[10];
delete polje; // pogrešno: potrebno je delete[]
```

Za dinamičko polje mora se primijeniti `delete[]`. To je posebno važno za polja objekata klasa jer se moraju pozvati destruktori svih elemenata.

10.12.5 Pristup izvan granica polja

```
int polje[5]{};
polje[5] = 10; // pogrešno: valjani su indeksi od 0 do 4
```

C++ pri uporabi operatora `[]` u pravilu ne provjerava granice. Pristup izvan polja može prebrisati druge podatke, oštetiti stanje programa ili stvoriti sigurnosni propust. Funkcijski član `vector::at()` obavlja provjeru granica i pri nevaljanom indeksu baca iznimku.

10.12.6 Neinicijalizirani i nevaljani pokazivači

```
int* prvi;            // neinicijaliziran pokazivač
int* drugi = nullptr; // ne pokazuje ni na jedan objekt

// *prvi i *drugi nisu dopušteni izrazi za pristup objektu
```

Nul-pokazivač ima poznato stanje i može se provjeriti. Neinicijalizirani pokazivač sadržava neodređenu vrijednost i ne smije se ni dereferencirati ni upotrebljavati kao da označuje valjanu adresu.

Najčešće pogreške

- Ne nazivajte svaku lokalnu varijablu „statičkom memorijom". Lokalni objekt bez `static` ima automatsko trajanje i uobičajeno se pohranjuje na *stacku*.

- Ne poistovjećujte pokazivač s objektom na koji pokazuje. To su dva odvojena objekta, često različitoga trajanja pohrane.
- Ne pretpostavljajte da završetak funkcije oslobađa objekte stvorene new-izrazom.
- Ne miješajte `new`/`delete` i `new[]`/`delete[]`.
- Ne upotrebljavajte pokazivač nakon uništenja objekta.
- Ne vraćajte adresu ni referencu na lokalni objekt automatskoga trajanja.
- Ne pozivajte `delete` nad objektom koji nije stvoren odgovarajućim izrazom `new`.
- Ne pretpostavljajte da uobičajeni new-izraz pri neuspjehu vraća `nullptr`; on baca iznimku `std::bad_alloc`.

10.13 Kako otkrivati memorijske pogreške

Upozorenja prevoditelja treba uključiti već tijekom učenja. Primjerice, za GCC i Clang korisne su mogućnosti `-Wall -Wextra -Wpedantic`. Mnoge pogreške životnoga vijeka i pristupa memoriji mogu se tijekom ispitivanja otkriti AddressSanitizerom i UndefinedBehaviorSanitizerom:

```
g++ -std=c++17 -Wall -Wextra -Wpedantic \
    -fsanitize=address,undefined program.cpp
```

Takvi alati nisu zamjena za razumijevanje vlasništva i životnoga vijeka, ali vrlo učinkovito otkrivaju pristup nakon oslobađanja, izlazak izvan granica, dvostruko oslobađanje i dio curenja memorije.

10.14 Dinamička memorija u jeziku C

U jeziku C dinamičkom se memorijom upravlja funkcijama iz zaglavlja `<stdlib.h>`:

```
void* malloc(size_t velicina);
void* calloc(size_t brojElemenata, size_t velicinaElementa);
void* realloc(void* pokazivac, size_t novaVelicina);
void free(void* pokazivac);
```

Funkcija `malloc` dodjeljuje zadani broj bajtova bez inicijalizacije. Funkcija `calloc` dodjeljuje prostor za zadani broj elemenata i sve bajtove postavlja na nulu. Funkcija `realloc` pokušava promijeniti veličinu postojećega bloka, a `free` oslobađa blok.

Sljedeći je isječak zapisan jezikom C, a ne jezikom C++. Prikazuje siguran osnovni obrazac za promjenu veličine bloka. Rezultat funkcije `realloc` najprije se sprema u privremeni pokazivač. Izravno pridruživanje izvornomu pokazivaču izgubilo bi adresu staroga bloka ako promjena veličine ne uspije:

```
#include <stdlib.h>

```

```c
int main(void)
{
    int* brojevi = malloc(5 * sizeof *brojevi);

    if (brojevi == NULL)
    {
        return EXIT_FAILURE;
    }

    int* noviBlok = realloc(brojevi, 10 * sizeof *brojevi);

    if (noviBlok != NULL)
    {
        brojevi = noviBlok;
    }

    // Ako realloc nije uspio, brojevi i dalje pokazuje na stari blok.
    free(brojevi);

    return EXIT_SUCCESS;
}
```

Za memoriju čiji svi bajtovi trebaju početno biti nula može se uporabiti `calloc`, primjerice `int* nule = calloc(5, sizeof *nule);`. I ta se memorija nakon uporabe oslobađa pozivom `free(nule);`.

U C++ kodu ne smiju se miješati obitelji `malloc`/`free` i `new`/`delete`. Funkcije jezika C same ne pozivaju konstruktore i destruktore C++ objekata.

10.15 Od ručnoga upravljanja prema RAII-ju

Ručno upravljanje memorijom važno je razumjeti, ali se u suvremenom C++ kodu ne primjenjuje kada standardna klasa već sigurno rješava isti problem:

Tablica 10.3: Uobičajena sigurnija zamjena za ručno upravljanje memorijom

Ručni oblik	Uobičajeni suvremeni izbor
dinamičko polje promjenjive veličine	`std::vector<T>`
dinamički znakovni niz	`std::string`
jedan dinamički objekt s jednim vlasnikom	`std::unique_ptr<T>` i `std::make_unique<T>()`
dinamičko polje bajtova ili objekata s posebnim pravilima	odgovarajući standardni spremnik ili vlastita RAII klasa

RAII ne znači da dinamička memorija više ne postoji. Ona i dalje postoji, ali je odgovornost za njezino oslobađanje smještena u objekt čiji se životni vijek može pouzdano pratiti.

Kako razmišljati kao programer?

Kada pročitate deklaraciju ili izraz koji stvara objekt, postavite ista četiri pitanja: što je stvoreno, gdje se uobičajeno nalazi, kada prestaje postojati i tko je odgovoran za njegovo uništenje. Ako na posljednje pitanje nema jasnoga odgovora, oblik upravljanja resursom treba preispitati.

Zapamtite

- C++ standard propisuje trajanje pohrane i životni vijek objekata; *stack* i *heap* opis su uobičajene implementacije.
- Lokalni objekt automatskoga trajanja uobičajeno je dio *stack framea* i automatski se uništava pri izlasku iz bloka.
- Lokalni `static` objekt ima lokalni opseg imena, ali statičko trajanje pohrane.
- Pokazivač i objekt na koji pokazuje dva su odvojena objekta.
- `new` se uparuje s `delete`, a `new[]` s `delete[]`.
- Nakon `delete` objekt više ne postoji, čak i ako pokazivač još sadržava njegovu staru adresu.
- `string`, `vector` i pametni pokazivači primjenjuju RAII te automatski oslobađaju memoriju kojom upravljaju.
- Svaki dinamički objekt mora imati jasno određeno vlasništvo i točno jedan siguran put do uništenja.

Kratki sažetak

Objekti u C++-u imaju opseg imena, životni vijek i trajanje pohrane. Lokalni objekti bez ključne riječi `static` najčešće imaju automatsko trajanje i uobičajeno se nalaze u *stack frameovima.* Globalni objekti, lokalni `static` objekti i statički članovi imaju statičko trajanje. Objekti stvoreni izrazima `new` i `new[]` imaju dinamičko trajanje i moraju se uništiti odgovarajućim delete-izrazom `delete p` ili `delete[] p`.

Sam pokazivač nije isto što i objekt na koji pokazuje. Lokalni pokazivač može prestati postojati, a dinamički objekt ostati neoslobođen; dinamički objekt može biti uništen, a pokazivač ostati s nevaljanom adresom. Upravo iz tih razlika nastaju curenje memorije, viseći pokazivači, dvostruko oslobađanje i pristup nakon uništenja. Standardni spremnici i pametni pokazivači vežu upravljanje memorijom uz životni vijek objekta i zato su u suvremenom C++-u sigurniji izbor od ručnoga upravljanja.

Što sada znamo?

Sada možemo za globalne i lokalne varijable, parametre funkcija, polja, pokazivače, objekte klasa, `string` i `vector` objasniti što nastaje, gdje se objekt uobičajeno pohranjuje, koliko dugo postoji i tko ga uništava. Možemo pratiti promjene *stacka* tijekom poziva funkcije, razlikovati

lokalni pokazivač od dinamičkoga objekta te prepoznati pogreške koje narušavaju životni vijek ili vlasništvo nad objektom.

Pitanja za samoprovjeru

1. Koja je razlika između opsega imena, životnoga vijeka objekta i trajanja pohrane?
2. Zašto nije potpuno precizno reći da C++ standard zahtijeva *stack* i heap?
3. Koji objekti uobičajeno čine *stack frame* funkcije?
4. Zašto lokalna varijabla deklarirana s `static` uobičajeno nije pohranjena u *stack frameu* funkcije?
5. Što se događa s lokalnim objektima automatskoga trajanja pri povratku iz funkcije?
6. Zašto funkcija ne smije vratiti adresu lokalne varijable automatskoga trajanja?
7. Koja dva odvojena objekta nastaju naredbom `int* p = new int(25);`?
8. Što se događa s dinamičkim objektom ako lokalni pokazivač koji je sadržavao njegovu adresu prestane postojati?
9. Koja je razlika između izraza `new int` i `new int{}`?
10. Kako uobičajeni new-izraz prijavljuje neuspjelu dodjelu?
11. Zašto se `new[]` mora upariti s `delete[]`?
12. Zašto pokazivač može ostati viseći nakon operatora `delete`?
13. Zašto postavljanje jednoga pokazivača na `nullptr` ne popravlja sve druge pokazivače s istom starom adresom?
14. Po čemu se u memoriji razlikuju lokalno ugrađeno polje i dinamičko polje?
15. Gdje se nalaze nestatički, a gdje statički podatkovni članovi klase?
16. Zašto veličina objekta klase može biti veća od zbroja veličina njegovih podatkovnih članova?
17. Gdje se uobičajeno nalazi lokalni objekt `vector`, a gdje njegovi elementi?
18. Kako načelo RAII sprječava curenje resursa pri prijevremenom izlasku iz funkcije?
19. Objasnite razliku između curenja memorije, visećega pokazivača i dvostrukoga oslobađanja.
20. Zašto se u C++ programu ne smiju miješati `malloc`/`free` i `new`/`delete`?

11. Klase i namespace

Što ćete naučiti?

Nakon obrade ovoga poglavlja moći ćete:

- objasniti razliku između klase i objekta;
- definirati podatkovne i funkcijske članove klase;
- primijeniti prava pristupa **`private`**, **`protected`** i **`public`**;
- koristiti konstruktore, destruktore i konstruktor kopije;
- raditi s pokazivačima na objekte;
- preopteretiti operatore;
- objasniti ulogu pokazivača **`this`**, statičkih članova i prijateljskih funkcija;
- primijeniti jednostruko i višestruko nasljeđivanje;
- razlikovati virtualne funkcije i apstraktne klase;
- objasniti ograničenu i posebnu namjenu kvalifikatora **`volatile`**;
- objasniti čemu služi scope operator `::` i prepoznati njegove najčešće uporabe;
- organizirati imena pomoću namespacea te razlikovati **`using`**-deklaraciju i **`using`**-direktivu.

Klasa ili razred predstavlja logički način organiziranja podataka i funkcija u jednu cjelinu. Može se promatrati kao skup varijabli istih ili različitih tipova i njima pridruženih funkcija. Klasa se deklarira ključnom riječju **`class`**. Slično se u programskom jeziku C struktura deklarira ključnom riječju **`struct`**, ali klasa u pravilu ne sadrži samo podatke, nego i funkcije koje tim podacima upravljaju.

Opći oblik deklaracije klase je:

```
class ime_klase {
pravo_pristupa_1:
    clan_1;
pravo_pristupa_2:
    clan_2;
    // ...
};
```

Unutar deklaracije klase upotrebljavaju se ključne riječi kojima se određuju prava pristupa podacima i funkcijama:

- **`private`** – članovima mogu pristupiti samo funkcijski članovi iste klase i njezini prijatelji;
- **`protected`** – pristup je sličan privatnom, ali je član dostupan i izvedenim klasama;
- **`public`** – članu se može pristupiti iz svakoga dijela programa u kojem je klasa vidljiva.

Zapamtite

Ako se pravo pristupa u klasi ne navede, podrazumijeva se `private`. U strukturi deklariranoj ključnom riječju `struct` podrazumijeva se `public`.

Analizirajmo jednostavnu deklaraciju klase pravokutnika:

```
class CPravokutnik {
    int x, y;

public:
    void postavi_vrijednost(int, int);
    int povrsina() const;
};

CPravokutnik pravokut;
```

Klasa `CPravokutnik` predstavlja novi, korisnički definirani tip podataka, a `pravokut` je objekt toga tipa. Klasa sadrži četiri člana:

- dvije varijable tipa `int`, `x` i `y`, s privatnim pravom pristupa;
- dvije javne funkcije, `postavi_vrijednost()` i `povrsina()`.

Odnos klase i objekta jednak je odnosu tipa i varijable. U izrazu

```
int a;
```

`int` je tip podataka, a `a` je varijabla toga tipa. Jednako tako, `CPravokutnik` je tip podataka, a `pravokut` objekt toga tipa.

Članovima objekta pristupa se operatorom točke:

```
pravokut.postavi_vrijednost(4, 5);
int povrsina_pravokuta = pravokut.povrsina();
```

Privatnim podatkovnim članovima `x` i `y` nije moguće pristupiti na isti način. Njima mogu pristupati samo funkcijski članovi iste klase i prijatelji klase.

Kako razmišljati kao programer?

Zašto podatke skrivamo? Zato što klasa treba sama nadzirati vlastito stanje. Umjesto da bilo koji dio programa izravno mijenja podatke, javne funkcije određuju koje su promjene dopuštene i na koji se način provode.

11.1 Definicija funkcijskih članova

Funkcijski član može biti definiran unutar deklaracije klase ili izvan nje. Ako je definiran izvan klase, u njegovu se imenu navodi scope operator `::`, kojim se određuje kojoj klasi funkcija pripada.

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

class CPravokutnik {
    int x, y;

public:
    void postavi_vrijednost(int, int);
    int povrsina() const { return x * y; }
};

void CPravokutnik::postavi_vrijednost(int a, int b) {
    x = a;
    y = b;
}

int main() {
    CPravokutnik pravokut;
    pravokut.postavi_vrijednost(4, 5);

    cout << "Povrsina: " << pravokut.povrsina() << endl;
    return 0;
}
```

Output

```
Povrsina: 20
```

U deklaraciji klase nalazi se samo prototip funkcije `postavi_vrijednost()`, dok je njezina definicija napisana izvan klase. Zbog toga se navodi puno ime funkcije:

```
void CPravokutnik::postavi_vrijednost(int a, int b)
```

Operator `::` određuje da funkcija pripada klasi `CPravokutnik`. Ona zato ima pristup privatnim podatkovnim članovima `x` i `y`.

Funkcija `povrsina()` definirana je unutar klase jer je vrlo kratka. Funkcije definirane unutar deklaracije klase implicitno su kandidati za umetanje na mjestu poziva (*inline*), iako konačnu odluku o tome donosi prevoditelj.

Napomena

U izvornim primjerima često se susreće `void main()`. Standardni C++ zahtijeva da funkcija `main()` vraća vrijednost tipa `int`, pa se u ovome izdanju upotrebljava oblik `int main()`.

Jedna od važnih prednosti klasa jest mogućnost stvaranja više objekata istoga tipa. Svaki objekt ima vlastite podatkovne članove.

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

class CPravokutnik {
    int x, y;

public:
    void postavi_vrijednost(int, int);
    int povrsina() const { return x * y; }
};

void CPravokutnik::postavi_vrijednost(int a, int b) {
    x = a;
    y = b;
}

int main() {
    CPravokutnik pravokut, pravokut_b;

    pravokut.postavi_vrijednost(4, 5);
    pravokut_b.postavi_vrijednost(10, 15);

    cout << "Povrsina prvoga je: "
         << pravokut.povrsina() << endl;
    cout << "Povrsina drugoga je: "
         << pravokut_b.povrsina() << endl;

    return 0;
}
```

Output

```
Povrsina prvoga je: 20
Povrsina drugoga je: 150
```

Funkcije `pravokut.povrsina()` i `pravokut_b.povrsina()` vraćaju različite rezultate jer svaki objekt ima vlastite vrijednosti podatkovnih članova `x` i `y`.

Što sada znamo?

- Klasa se može sastojati od varijabli, funkcija i drugih klasa.
- Varijable unutar klase nazivaju se podatkovnim članovima.
- Funkcije unutar klase nazivaju se funkcijskim članovima ili metodama.
- Funkcijski članovi određuju što objekt može učiniti.
- Objekt nastao prema definiciji klase naziva se instancom klase.

11.2 Konstruktori i destruktori

Objekt pri stvaranju treba dobiti valjano početno stanje. Ako bi se u prethodnome primjeru funkcija `povrsina()` pozvala prije funkcije `postavi_vrijednost()`, podatkovni članovi ne bi bili inicijalizirani, pa bi rezultat bio neodređen.

Klasa zato može imati poseban funkcijski član koji se naziva konstruktor. Konstruktor:

- ima isto ime kao klasa;
- nema povratni tip, čak ni `void`;
- može primati parametre;
- poziva se automatski pri stvaranju objekta.

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

class CPravokutnik {
    int sirina, visina;

public:
    CPravokutnik(int, int);
    int povrsina() const { return sirina * visina; }
};

CPravokutnik::CPravokutnik(int a, int b) {
    sirina = a;
    visina = b;
}

int main() {
    CPravokutnik pravokut(4, 5);
    CPravokutnik pravokut_b(10, 15);

    cout << "Povrsina prvoga je: "
         << pravokut.povrsina() << endl;
    cout << "Povrsina drugoga je: "
```

```
24          << pravokut_b.povrsina() << endl;
25
26      return 0;
27  }
```

Output

```
Povrsina prvoga je: 20
Povrsina drugoga je: 150
```

Parametri se konstruktoru predaju u trenutku stvaranja objekta:

```
1  CPravokutnik pravokut(4, 5);
2  CPravokutnik pravokut_b(10, 15);
```

Destruktor obavlja završne radnje pri prestanku životnoga vijeka objekta. Ima isto ime kao klasa, ali s tildom ~ kao prefiksom. Ne prima parametre i nema povratni tip. Posebno je važan kada objekt upravlja resursima, primjerice dinamički alociranom memorijom.

```
1  #include <iostream>
2  using namespace std;
3
4  class CPravokutnik {
5      int* sirina;
6      int* visina;
7
8  public:
9      CPravokutnik(int, int);
10     ~CPravokutnik();
11     int povrsina() const { return (*sirina) * (*visina); }
12 };
13
14 CPravokutnik::CPravokutnik(int a, int b) {
15     sirina = new int;
16     visina = new int;
17     *sirina = a;
18     *visina = b;
19 }
20
21 CPravokutnik::~CPravokutnik() {
22     delete sirina;
23     delete visina;
24 }
25
26 int main() {
27     CPravokutnik pravokut(4, 5);
28     CPravokutnik pravokut_b(10, 15);
29
30     cout << "Povrsina prvoga je: "
```

```
31          << pravokut.povrsina() << endl;
32      cout << "Povrsina drugoga je: "
33          << pravokut_b.povrsina() << endl;
34
35      return 0;
36  }
```

Output

```
Povrsina prvoga je: 20
Povrsina drugoga je: 150
```

Napomena

Ovaj primjer služi samo za prikaz uloge destruktora. U stvarnom C++ programu dva cijela broja pohranjivala bi se izravno kao članovi tipa `int`, bez dinamičke dodjele memorije.

Najčešće pogreške

Ako klasa ručno upravlja dinamičkom memorijom, nije dovoljno definirati samo destruktor. Potrebno je pravilno urediti i kopiranje objekata. U protivnom se mogu pojaviti plitke kopije, višestruko oslobađanje iste memorije i pokazivači na nepostojeće podatke. U gornjem primjeru, automatski generirani konstruktor kopije i operator pridruživanja kopirali bi samo adrese pohranjene u pokazivačima. Posljedica su plitke kopije i višestruko oslobađanje iste memorije. O tome će biti riječi u odjeljku o operatoru pridruživanja.

11.2.1 Preopterećenje konstruktora

Konstruktori se, kao i druge funkcije, mogu preopteretiti. Više konstruktora može imati isto ime, ali različit broj parametara ili različite tipove parametara. Prevoditelj odabire konstruktor koji odgovara argumentima navedenima pri stvaranju objekta.

Podrazumijevani konstruktor (engl. *default constructor*) konstruktor je koji se može pozvati bez argumenata. Najčešće nema parametara, ali to može biti i konstruktor čiji svi parametri imaju zadane vrijednosti. Ako klasa nema nijedan korisnički deklariran konstruktor, prevoditelj implicitno deklarira podrazumijevani konstruktor.

Čim programer deklarira bilo koji konstruktor, primjerice konstruktor s parametrima, prevoditelj više ne deklarira implicitni podrazumijevani konstruktor. Zato se za klasu koja ima samo konstruktor `CPravokutnik(int, int)` objekt ne može deklarirati bez navođenja argumenata:

```
1 CPravokutnik pravokut;          // pogreška
2 CPravokutnik pravokut_b(4, 5);  // ispravno
```

Ako objekte želimo stvarati i bez argumenata i s argumentima, u klasi moramo deklarirati i definirati oba konstruktora.

Konstruktor kopije služi stvaranju novoga objekta prema već postojećem objektu iste klase. Njegov uobičajeni oblik je:

```
CPrimjer(const CPrimjer& drugi);
```

Ručno definiran konstruktor kopije može glasiti:

```
CPrimjer::CPrimjer(const CPrimjer& drugi) {
    a = drugi.a;
    b = drugi.b;
    c = drugi.c;
}
```

Napomena

Točna pravila o implicitnom stvaranju podrazumijevanoga konstruktora, konstruktora kopije i operatora pridruživanja ovise o tome koje je posebne funkcijske članove programer već deklarirao. Za početno razumijevanje važno je zapamtiti da se automatski generirani članovi ne smiju nekritički koristiti kada klasa upravlja resursima.

U sljedećem su programu izričito deklarirana oba preopterećena konstruktora: podrazumijevani konstruktor i konstruktor s dva parametra.

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

class CPravokutnik {
    int sirina, visina;

public:
    CPravokutnik();
    CPravokutnik(int, int);
    int povrsina() const { return sirina * visina; }
};

CPravokutnik::CPravokutnik() {
    sirina = 5;
    visina = 5;
}

CPravokutnik::CPravokutnik(int a, int b) {
    sirina = a;
    visina = b;
}

int main() {
    CPravokutnik pravokut(3, 4);
    CPravokutnik pravokut_b;

    cout << "Pravokut -- povrsina: "
```

```
28          << pravokut.povrsina() << endl;
29      cout << "Pravokut_b -- povrsina: "
30           << pravokut_b.povrsina() << endl;
31
32      return 0;
33  }
```

Output

```
Pravokut -- povrsina: 12
Pravokut_b -- povrsina: 25
```

Objekt `pravokut_b` stvoren je bez argumenata, pa je pozvan konstruktor bez parametara. Kada se objekt deklarira bez argumenata, iza njegova se imena ne stavljaju prazne zagrade:

```
1  CPravokutnik pravokut_b;    // objekt
2  CPravokutnik pravokut_b();  // deklaracija funkcije
```

11.3 Pokazivači na klase

Nakon deklariranja klase njezino se ime može upotrebljavati kao valjani tip podataka. Zato se može deklarirati i pokazivač na objekt klase:

```
1  CPravokutnik* pok;
```

Članu objekta pristupa se operatorom `.`, a članu objekta na koji pokazuje pokazivač operatorom `->`.

```
1  #include <iostream>
2  using namespace std;
3
4  class CPravokutnik {
5      int sirina, visina;
6
7  public:
8      void postavi_vrijednost(int, int);
9      int povrsina() const { return sirina * visina; }
10 };
11
12 void CPravokutnik::postavi_vrijednost(int a, int b) {
13     sirina = a;
14     visina = b;
15 }
16
17 int main() {
18     CPravokutnik a;
19     CPravokutnik* b = new CPravokutnik;
```

```
    CPravokutnik* c = &a;
    CPravokutnik* d = new CPravokutnik[2];

    a.postavi_vrijednost(1, 2);
    b->postavi_vrijednost(3, 4);
    d[0].postavi_vrijednost(5, 6);
    d[1].postavi_vrijednost(7, 8);

    cout << "a povrsina: " << a.povrsina() << endl;
    cout << "*b povrsina: " << b->povrsina() << endl;
    cout << "*c povrsina: " << c->povrsina() << endl;
    cout << "d[0] povrsina: " << d[0].povrsina() << endl;
    cout << "d[1] povrsina: " << d[1].povrsina() << endl;

    delete b;
    delete[] d;
    return 0;
}
```

Output

```
a povrsina: 2
*b povrsina: 12
*c povrsina: 2
d[0] povrsina: 30
d[1] povrsina: 56
```

Izraz	Značenje
`*x`	objekt ili vrijednost na koju pokazuje `x`
`&x`	adresa objekta `x`
`x.y`	član `y` objekta `x`
`(*x).y`	član `y` objekta na koji pokazuje `x`
`x->y`	isto što i `(*x).y`
`x[0]`	prvi objekt u polju na koje pokazuje `x`
`x[1]`	drugi objekt u polju na koje pokazuje `x`
`x[n]`	objekt s indeksom `n`

11.3.1 Klase definirane ključnom riječju `struct`

U C++-u struktura može imati iste vrste članova kao i klasa: podatkovne članove, funkcijske članove, konstruktore, nasljeđivanje i druga svojstva. Razlika je u podrazumijevanom pravu pristupa:

- u klasi je podrazumijevani pristup **`private`**;
- u strukturi je podrazumijevani pristup **`public`**.

U praksi se `struct` često upotrebljava za jednostavne skupove podataka, a `class` za tipove koji skrivaju podatke i nude javno sučelje.

11.4 Preopterećenje operatora

C++ sadrži operatore za ugrađene tipove podataka, primjerice +, -, *, == i =. Budući da klasa predstavlja korisnički definirani tip, programer može odrediti značenje postojećega operatora za objekte te klase. Taj se postupak naziva preopterećenje operatora.

Nije moguće stvarati nove operatorske simbole. Mogu se preopteretiti samo operatori koje C++ već poznaje.

11.4.1 Preopterećenje operatora +

Za ugrađene tipove značenje zbrajanja je unaprijed određeno:

```
int a = 3;
int b = 5;
int c = a + b;
```

Rezultat je 8. Za zbrajanje dvaju objekata klase treba odrediti koji se članovi zbrajaju i kakav se objekt vraća kao rezultat.

Promotrimo klasu dvodimenzijskoga vektora:

```
class CVektor {
    double dx;
    double dy;

public:
    CVektor() : dx(0), dy(0) {}
    CVektor(double x, double y) : dx(x), dy(y) {}

    void unesi_x(double x) { dx = x; }
    void unesi_y(double y) { dy = y; }
    double daj_x() const { return dx; }
    double daj_y() const { return dy; }
};
```

Pretpostavimo da su stvorena tri objekta:

```
CVektor a(2, 3), b(2, 2), c;
```

Prvi je način uporaba obične funkcije:

```
CVektor zbroji_vektore(const CVektor& vektor_1,
                       const CVektor& vektor_2) {
    return CVektor(vektor_1.daj_x() + vektor_2.daj_x(),
```

```
                    vektor_1.daj_y() + vektor_2.daj_y());
}

c = zbroji_vektore(a, b);
```

Drugi je način funkcijski član klase:

```
CVektor zbroji(const CVektor& v) const {
    return CVektor(dx + v.daj_x(), dy + v.daj_y());
}

c = a.zbroji(b);
```

Najčitljiviji je oblik:

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

Da bi takav zapis bio moguć, operator + treba preopteretiti.

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

class CVektor {
    double dx;
    double dy;

public:
    CVektor() : dx(0), dy(0) {}
    CVektor(double x, double y) : dx(x), dy(y) {}

    double daj_x() const { return dx; }
    double daj_y() const { return dy; }

    CVektor operator+(const CVektor& v) const;
};

CVektor CVektor::operator+(const CVektor& v) const {
    return CVektor(dx + v.daj_x(), dy + v.daj_y());
}

int main() {
    CVektor a(2, 3), b(2, 2);
    CVektor c = a + b;

    cout << c.daj_x() << "i + " << c.daj_y() << "j" << endl;
    return 0;
}
```

Output

```
4i + 5j
```

Izraz

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

može se shvatiti kao čitljiviji zapis poziva:

```
c = a.operator+(b);
```

Objekt lijevo od operatora jest objekt nad kojim se poziva funkcijski član, dok se objekt desno od operatora prosljeđuje kao argument.

11.4.2 Preopterećenje operatora ++ i --

Operatori inkrementa i dekrementa mogu se pojaviti u prefiksnom i sufiksnom obliku. Za razlikovanje sufiksnoga oblika, operatorska funkcija prima neupotrijebljeni parametar tipa `int`.

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

class CBrojac {
    int broji;

public:
    CBrojac() : broji(0) {}

    int daj_stanje() const { return broji; }
    CBrojac& operator++()                    // prefiksni oblik
    {
        ++broji;
        return *this;
    }

    CBrojac operator++(int)                  // sufiksni oblik
    {
        CBrojac stari = *this;
        ++(*this);
        return stari;
    }
};

int main() {
    CBrojac brojac;

    cout << brojac++.daj_stanje() << endl;
    cout << (++brojac).daj_stanje() << endl;
```

```
30
31      return 0;
32  }
```

Output

```
0
2
```

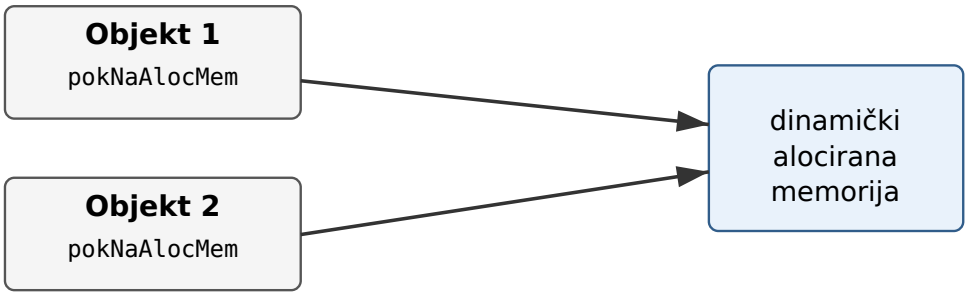
Kod sufiksnoga operatora prvo se vraća prethodna vrijednost, a zatim se stanje povećava. Kod prefiksnoga operatora stanje se najprije povećava, a zatim se vraća nova vrijednost. Ista se načela primjenjuju na operator dekrementa `--`.

11.4.3 Preopterećenje operatora pridruživanja =

Operator pridruživanja poziva se kada se postojeći objekt pridružuje drugom postojećem objektu:

```
1  CVektor a;
2  CVektor b(2, 3);
3  a = b;
```

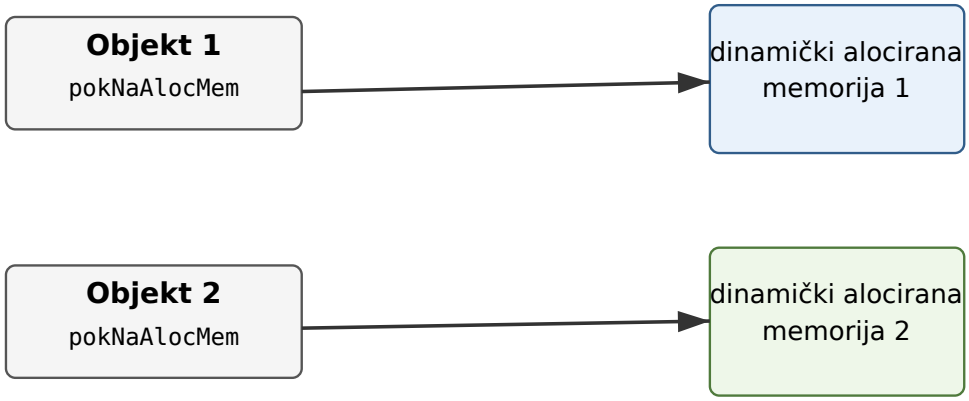
Ako klasa sadrži samo vrijednosne podatkovne članove, implicitni operator pridruživanja kopira svaki član. Problem nastaje kada klasa sadrži pokazivač na dinamički alociranu memoriju. Obično kopiranje tada kopira samo adresu, pa oba objekta pokazuju na isto memorijsko područje. Takva se kopija naziva plitka kopija.



Slika 11.1: Plitka kopija: dva objekta pokazuju na isto memorijsko područje

Ako jedan objekt oslobodi zajedničku memoriju, pokazivač u drugome objektu više nije valjan. Pokušaj pristupa takvoj memoriji uzrokuje nedefinirano ponašanje. Odnos je prikazan na slici 11.1.

Duboka kopija stvara novo memorijsko područje i u njega kopira sadržaj. Svaki objekt tada ima vlastiti resurs, kao na slici 11.2.



Slika 11.2: Duboka kopija: svaki objekt posjeduje vlastito memorijsko područje

Pojednostavljeni oblik operatora pridruživanja koji izvodi duboku kopiju može izgledati ovako:

```
CPrimjer& CPrimjer::operator=(const CPrimjer& drugi) {
    if (this != &drugi) {
        int* novi = new int(*drugi.podatak);
        delete podatak;
        podatak = novi;
    }
    return *this;
}
```

Provjera `this != &drugi` štiti od samopridruživanja, a vraćanje `*this` omogućuje lančano pridruživanje.

Zapamtite

Ako klasa ručno upravlja resursom, potreba za korisnički definiranim destruktorom u pravilu znači da treba razmotriti i konstruktor kopije te operator pridruživanja kopiranjem. To se naziva pravilom trojke. U suvremenom C++-u poželjno je vlasništvo povjeriti tipovima koji sami upravljaju resursom, primjerice klasama `string`, `vector` i pametnim pokazivačima. Tada klasa najčešće ne mora ručno definirati nijednu od tih operacija.

Moguće je preopteretiti i druge operatore. Primjerice, operator == može odrediti da su dva vektora jednaka samo ako su im jednake obje komponente.

11.5 Ključna riječ `this`

Pri pozivu nestatičkoga funkcijskog člana prevoditelj funkciji prosljeđuje skriveni pokazivač na objekt nad kojim je funkcija pozvana. Tom pokazivaču pristupa se ključnom riječju **`this`**.

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

class CPrimjer {
public:
    bool isti_objekt(const CPrimjer& drugi);
};

bool CPrimjer::isti_objekt(const CPrimjer& drugi) {
    return &drugi == this;
}

int main() {
    CPrimjer a;
    CPrimjer* b = &a;

    if (b->isti_objekt(a))
        cout << "Da, &a je b" << endl;

    return 0;
}
```

Output

```
Da, &a je b
```

Funkcija `isti_objekt()` uspoređuje adresu argumenta s pokazivačem **`this`**. Budući da oba pokazuju na isti objekt, uvjet je istinit.

11.6 Statički članovi

Klasa može sadržavati statičke podatkovne i funkcijske članove. Statički podatkovni član ne pripada pojedinom objektu, nego klasi kao cjelini. Postoji samo jedna njegova vrijednost koju dijele svi objekti te klase.

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

class CPrimjer {
public:
    static int broj_objekata;

    CPrimjer() { broj_objekata++; }
    ~CPrimjer() { broj_objekata--; }
};

int CPrimjer::broj_objekata = 0;
```

```
13
14 int main() {
15     CPrimjer a;
16     cout << a.broj_objekata << endl;
17
18     CPrimjer b;
19     cout << a.broj_objekata << endl;
20
21     CPrimjer* c = new CPrimjer;
22     cout << CPrimjer::broj_objekata << endl;
23
24     delete c;
25     cout << CPrimjer::broj_objekata << endl;
26
27     return 0;
28 }
```

Output

```
1
2
3
2
```

Statičkom članu može se pristupiti preko objekta, ali je jasnije koristiti ime klase i scope operator:

```
1 CPrimjer::broj_objekata
```

11.7 Prijateljska funkcija

Privatnim i zaštićenim članovima u pravilu se ne može pristupiti izvan klase. Iznimka je funkcija koja je u deklaraciji klase označena ključnom riječju **friend**. Takva funkcija nije funkcijski član klase, ali ima pristup njezinim privatnim i zaštićenim članovima.

```
1  #include <iostream>
2  using namespace std;
3
4  class CPravokutnik {
5      int sirina, visina;
6
7  public:
8      void postavi_vrijednost(int, int);
9      int povrsina() const { return sirina * visina; }
10
11     friend CPravokutnik dupliciraj(const CPravokutnik&);
12 };
13
```

```
14 void CPravokutnik::postavi_vrijednost(int a, int b) {
15     sirina = a;
16     visina = b;
17 }
18
19 CPravokutnik dupliciraj(const CPravokutnik& ulaz) {
20     CPravokutnik rezultat;
21     rezultat.sirina = ulaz.sirina * 2;
22     rezultat.visina = ulaz.visina * 2;
23     return rezultat;
24 }
25
26 int main() {
27     CPravokutnik pravokut_a, pravokut_b;
28
29     pravokut_a.postavi_vrijednost(2, 3);
30     pravokut_b = dupliciraj(pravokut_a);
31
32     cout << pravokut_b.povrsina() << endl;
33     return 0;
34 }
```

Output

```
24
```

Funkcija `dupliciraj()` izravno pristupa privatnim članovima `sirina` i `visina`, jer je u klasi deklarirana kao prijateljska funkcija.

Napomena

Prijateljstvo treba koristiti promišljeno. Ono namjerno zaobilazi zaštitu podataka, pa prevelik broj prijateljskih funkcija i klasa može otežati održavanje programa.

11.7.1 Prijateljska klasa

Ponekad je potrebno omogućiti svim funkcijskim članovima jedne klase pristup privatnim i zaštićenim članovima druge klase. Tada se cijela klasa može proglasiti prijateljskom.

```
1 #include <iostream>
2 using namespace std;
3
4 class CKvadrat;
5
6 class CPravokutnik {
7     int sirina, visina;
8
9 public:
```

```
    int povrsina() const { return sirina * visina; }
    void pretvori(const CKvadrat&);
};

class CKvadrat {
    int stranica;

public:
    void postavi_stranicu(int a) { stranica = a; }
    friend class CPravokutnik;
};

void CPravokutnik::pretvori(const CKvadrat& kvadrat) {
    sirina = kvadrat.stranica;
    visina = kvadrat.stranica;
}

int main() {
    CKvadrat kvadrat;
    CPravokutnik pravokut;

    kvadrat.postavi_stranicu(4);
    pravokut.pretvori(kvadrat);

    cout << pravokut.povrsina() << endl;
    return 0;
}
```

Output

```
16
```

Na početku programa nalazi se unaprijedna deklaracija:

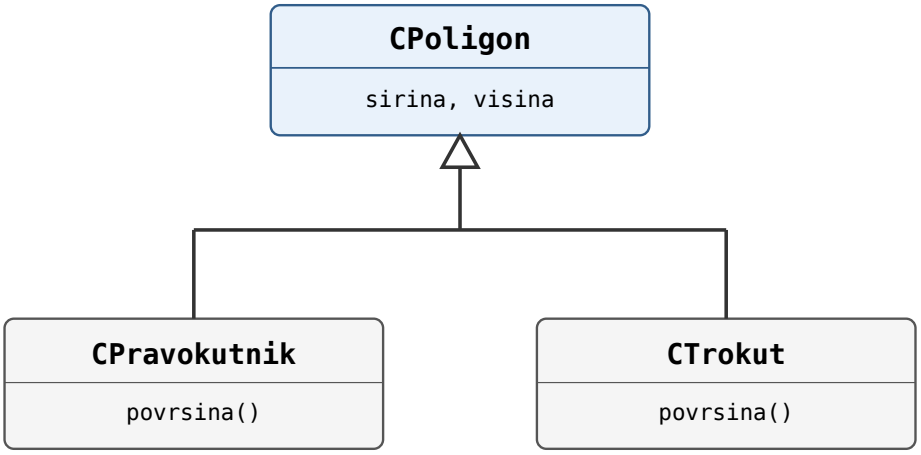
```
class CKvadrat;
```

Ona je potrebna zato što se ime klase `CKvadrat` pojavljuje u deklaraciji klase `CPravokutnik` prije nego što je `CKvadrat` potpuno definirana.

11.8 Nasljeđivanje između klasa

Nasljeđivanje omogućuje stvaranje nove klase na temelju postojeće klase. Izvedena klasa preuzima dostupne članove osnovne klase i može dodati vlastite članove ili redefinirati ponašanje.

Pretpostavimo da želimo opisati pravokutnik i trokut. Oba su poligoni i oba se mogu opisati širinom i visinom. Zajednički se članovi mogu izdvojiti u osnovnu klasu `CPoligon`, a posebna ponašanja smjestiti u izvedene klase `CPravokutnik` i `CTrokut`. Odnos klasa prikazan je na slici 11.3.



Slika 11.3: Primjer hijerarhije nasljeđivanja

Opći oblik javnoga nasljeđivanja je:

```
class IzvedenaKlasa : public OsnovnaKlasa {
    // clanovi izvedene klase
};
```

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

class CPoligon {
protected:
    int sirina, visina;

public:
    void postavi_vrijednosti(int a, int b) {
        sirina = a;
        visina = b;
    }
};

class CPravokutnik : public CPoligon {
public:
    int povrsina() const {
        return sirina * visina;
    }
};

class CTrokut : public CPoligon {
public:
    int povrsina() const {
        return sirina * visina / 2;
```

```
    }
};

int main() {
    CPravokutnik pravokut;
    CTrokut trokut;

    pravokut.postavi_vrijednosti(4, 5);
    trokut.postavi_vrijednosti(4, 5);

    cout << pravokut.povrsina() << endl;
    cout << trokut.povrsina() << endl;

    return 0;
}
```

Output

```
20
10
```

Objekti klasa `CPravokutnik` i `CTrokut` nasljeđuju članove `sirina`, `visina` i `postavi_vrijednosti()` iz klase `CPoligon`.

Zaštićeni pristup sličan je privatnom, ali izvedena klasa može pristupiti zaštićenim članovima osnovne klase. Privatni članovi osnovne klase postoje u objektu izvedene klase, ali im izvedena klasa ne može izravno pristupiti.

	`public`	`protected`	`private`
Funkcijski članovi iste klase	da	da	da
Funkcijski članovi izvedene klase	da	da	ne
Kod izvan klase	da	ne	ne

Kod javnoga nasljeđivanja javni članovi osnovne klase ostaju javni, a zaštićeni ostaju zaštićeni. Kod zaštićenoga nasljeđivanja javni i zaštićeni članovi osnovne klase postaju zaštićeni u izvedenoj klasi. Kod privatnoga nasljeđivanja javni i zaštićeni članovi osnovne klase postaju privatni u izvedenoj klasi.

Ako se pravo nasljeđivanja ne navede, za klasu deklariranu ključnom riječju `class` podrazumijeva se privatno nasljeđivanje, a za `struct` javno nasljeđivanje.

Zapamtite

Konstruktori i destruktori osnovne klase ne nasljeđuju se kao obični funkcijski članovi. Ipak, pri stvaranju objekta izvedene klase najprije se poziva konstruktor osnovne klase, a zatim konstruktor izvedene klase. Pri uništavanju se redoslijed obrće.

Ako osnovna klasa nema odgovarajući konstruktor bez parametara, konstruktor osnovne klase navodi se u inicijalizacijskoj listi konstruktora izvedene klase:

```
IzvedenaKlasa(parametri)
    : OsnovnaKlasa(argumenti) {
    // tijelo konstruktora izvedene klase
}
```

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

class CMajka {
public:
    CMajka() {
        cout << "Majka: nema parametara" << endl;
    }

    CMajka(int) {
        cout << "Majka: int parametar" << endl;
    }
};

class CKcer : public CMajka {
public:
    CKcer(int) {
        cout << "Kcer: int parametar" << endl;
    }
};

class CSin : public CMajka {
public:
    CSin(int a) : CMajka(a) {
        cout << "Sin: int parametar" << endl;
    }
};

int main() {
    CKcer ana(1);
    CSin ante(1);
    return 0;
}
```

Output

```
Majka: nema parametara
Kcer: int parametar
Majka: int parametar
Sin: int parametar
```

Pri stvaranju objekta `ana` nije izričito naveden konstruktor osnovne klase, pa se poziva konstruktor bez parametara. Pri stvaranju objekta `ante` konstruktor osnovne klase pozvan je argumentom u inicijalizacijskoj listi.

11.8.1 Višestruko nasljeđivanje

C++ dopušta da izvedena klasa naslijedi članove iz više osnovnih klasa. Osnovne se klase u deklaraciji odvajaju zarezima.

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

class CPoligon {
protected:
    int sirina, visina;

public:
    void postavi_vrijednosti(int a, int b) {
        sirina = a;
        visina = b;
    }
};

class CIzlaz {
public:
    void izlaz(int vrijednost) const {
        cout << vrijednost << endl;
    }
};

class CPravokutnik : public CPoligon, public CIzlaz {
public:
    int povrsina() const {
        return sirina * visina;
    }
};

class CTrokut : public CPoligon, public CIzlaz {
public:
    int povrsina() const {
        return sirina * visina / 2;
    }
};

int main() {
    CPravokutnik pravokut;
    CTrokut trokut;

    pravokut.postavi_vrijednosti(4, 5);
    trokut.postavi_vrijednosti(4, 5);
```

```
42
43      pravokut.izlaz(pravokut.povrsina());
44      trokut.izlaz(trokut.povrsina());
45
46      return 0;
47  }
```

Output

```
20
10
```

Klase `CPravokutnik` i `CTrokut` nasljeđuju geometrijske podatke iz klase `CPoligon`, a funkciju ispisa iz klase `CIzlaz`.

11.9 Pokazivači na osnovnu klasu

Pokazivaču na osnovnu klasu može se pridružiti adresa objekta izvedene klase. Takav pokazivač, bez virtualnih funkcija, može izravno pristupati samo članovima koji su dostupni preko sučelja osnovne klase.

```
1   #include <iostream>
2   using namespace std;
3
4   class CPoligon {
5   protected:
6       int sirina, visina;
7
8   public:
9       void postavi_vrijednosti(int a, int b) {
10          sirina = a;
11          visina = b;
12      }
13  };
14
15  class CPravokutnik : public CPoligon {
16  public:
17      int povrsina() const {
18          return sirina * visina;
19      }
20  };
21
22  class CTrokut : public CPoligon {
23  public:
24      int povrsina() const {
25          return sirina * visina / 2;
26      }
```

```
};

int main() {
    CPravokutnik pravokut;
    CTrokut trokut;

    CPoligon* poligon_1 = &pravokut;
    CPoligon* poligon_2 = &trokut;

    poligon_1->postavi_vrijednosti(4, 5);
    poligon_2->postavi_vrijednosti(4, 5);

    cout << pravokut.povrsina() << endl;
    cout << trokut.povrsina() << endl;

    return 0;
}
```

Output

```
20
10
```

Pokazivači `poligon_1` i `poligon_2` imaju tip `CPoligon*`, ali pokazuju na objekte izvedenih klasa. Preko njih je moguće pozvati `postavi_vrijednosti()`, jer je ta funkcija član osnovne klase. Funkcija `povrsina()` u ovom primjeru nije član osnovne klase, pa joj se preko tih pokazivača ne može pristupiti.

11.10 Virtualni funkcijski članovi

Ako osnovna klasa deklarira funkciju koju izvedene klase trebaju redefinirati, funkcija se može označiti ključnom riječju **`virtual`**. Tada poziv preko pokazivača ili reference na osnovnu klasu odabire implementaciju prema stvarnom tipu objekta.

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

class CPoligon {
protected:
    int sirina, visina;

public:
    void postavi_vrijednosti(int a, int b) {
        sirina = a;
        visina = b;
    }

    virtual int povrsina() const {
```

```
        return 0;
    }
};

class CPravokutnik : public CPoligon {
public:
    int povrsina() const override {
        return sirina * visina;
    }
};

class CTrokut : public CPoligon {
public:
    int povrsina() const override {
        return sirina * visina / 2;
    }
};

int main() {
    CPravokutnik pravokut;
    CTrokut trokut;
    CPoligon poligon;

    CPoligon* poligon_1 = &pravokut;
    CPoligon* poligon_2 = &trokut;
    CPoligon* poligon_3 = &poligon;

    poligon_1->postavi_vrijednosti(4, 5);
    poligon_2->postavi_vrijednosti(4, 5);
    poligon_3->postavi_vrijednosti(4, 5);

    cout << poligon_1->povrsina() << endl;
    cout << poligon_2->povrsina() << endl;
    cout << poligon_3->povrsina() << endl;

    return 0;
}
```

Output

```
20
10
0
```

Ključna riječ `override` nije obvezna, ali je korisna jer prevoditelju nalaže da provjeri postoji li u osnovnoj klasi odgovarajuća virtualna funkcija.

Zapamtite

Ako se objekti izvedenih klasa namjeravaju uništavati preko pokazivača na osnovnu klasu, destruktor osnovne klase treba biti virtualan.

11.11 Apstraktne klase

U hijerarhiji klasa osnovna klasa ponekad predstavlja samo zajedničko sučelje. Ne mora imati smisla stvarati njezine objekte. Takva klasa može sadržavati čistu virtualnu funkciju, koja se deklarira dodavanjem `= 0`:

```
virtual int povrsina() const = 0;
```

Klasa koja sadrži barem jednu čistu virtualnu funkciju naziva se apstraktnom klasom. Nije dopušteno stvarati njezine objekte, ali je dopušteno stvarati pokazivače i reference na nju.

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

class CPoligon {
protected:
    int sirina, visina;

public:
    virtual ~CPoligon() = default;

    void postavi_vrijednosti(int a, int b) {
        sirina = a;
        visina = b;
    }

    virtual int povrsina() const = 0;
};

class CPravokutnik : public CPoligon {
public:
    int povrsina() const override {
        return sirina * visina;
    }
};

class CTrokut : public CPoligon {
public:
    int povrsina() const override {
        return sirina * visina / 2;
    }
};

int main() {
```

```
    CPravokutnik pravokut;
    CTrokut trokut;

    CPoligon* poligon_1 = &pravokut;
    CPoligon* poligon_2 = &trokut;

    poligon_1->postavi_vrijednosti(4, 5);
    poligon_2->postavi_vrijednosti(4, 5);

    cout << poligon_1->povrsina() << endl;
    cout << poligon_2->povrsina() << endl;

    return 0;
}
```

Output

```
20
10
```

Pokazivači istoga tipa, `CPoligon*`, mogu pokazivati na objekte različitih izvedenih klasa. Poziv virtualne funkcije odabire odgovarajuću izvedenu implementaciju.

Osnovna klasa može sadržavati i običnu funkciju koja poziva čistu virtualnu funkciju. Time se zajedničko ponašanje definira samo jednom.

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

class CPoligon {
protected:
    int sirina, visina;

public:
    virtual ~CPoligon() = default;

    void postavi_vrijednosti(int a, int b) {
        sirina = a;
        visina = b;
    }

    virtual int povrsina() const = 0;

    void ispisi_povrsinu() const {
        cout << this->povrsina() << endl;
    }
};

class CPravokutnik : public CPoligon {
public:
```

```
    int povrsina() const override {
        return sirina * visina;
    }
};

class CTrokut : public CPoligon {
public:
    int povrsina() const override {
        return sirina * visina / 2;
    }
};

int main() {
    CPravokutnik pravokut;
    CTrokut trokut;

    CPoligon* poligon_1 = &pravokut;
    CPoligon* poligon_2 = &trokut;

    poligon_1->postavi_vrijednosti(4, 5);
    poligon_2->postavi_vrijednosti(4, 5);

    poligon_1->ispisi_povrsinu();
    poligon_2->ispisi_povrsinu();

    return 0;
}
```

Output

```
20
10
```

Pokazivač `this` u funkciji `ispisi_povrsinu()` pokazuje na objekt čiji se kod trenutačno izvršava. Budući da je `povrsina()` virtualna, poziva se implementacija stvarnoga tipa objekta.

11.12 Scope resolution operator `::`

U C++-u isto ime može postojati u različitim dijelovima programa: u globalnom području, klasi, osnovnoj klasi ili namespaceu. Scope resolution operator `::` određuje u kojem se od tih područja traži određeno ime. Opći je oblik:

```
podrucje::ime
```

Naziv s lijeve strane operatora određuje područje kojem ime s desne strane pripada. Ako s lijeve strane nema naziva, zapis počinje operatorom `::` i odnosi se na globalno područje.

11.12.1 Pristup imenu iz globalnoga područja

Ako lokalna varijabla ima isto ime kao globalna, lokalna varijabla skriva globalnu. Operator `::` omogućuje izričit pristup globalnoj varijabli:

```cpp
#include <iostream>

int broj = 3;

int main() {
    int broj = 10;

    std::cout << broj << '\n';   // lokalna varijabla
    std::cout << ::broj << '\n'; // globalna varijabla
    return 0;
}
```

Output

```
10
3
```

11.12.2 Određivanje pripadnosti klasi

Scope operator već smo upotrijebili pri definiranju funkcijskoga člana izvan deklaracije klase:

```cpp
void CPravokutnik::postavi_vrijednost(int a, int b)
```

Zapis `CPravokutnik::postavi_vrijednost` govori da je funkcija `postavi_vrijednost()` član klase `CPravokutnik`. Isti se operator upotrebljava za pristup statičkim članovima preko imena klase:

```cpp
CPrimjer::broj_objekata
```

11.12.3 Pristup članu osnovne klase

Ako izvedena klasa definira funkciju istoga imena kao osnovna klasa, funkcija osnovne klase može se izričito odabrati njezinim punim imenom:

```cpp
#include <iostream>

class COsnovna {
public:
    void ispisi() const {
        std::cout << "Osnovna klasa\n";
    }
};
```

```
9
10 class CIzvedena : public COsnovna {
11 public:
12     void ispisi() const {
13         std::cout << "Izvedena klasa\n";
14     }
15
16     void ispisi_osnovnu() const {
17         COsnovna::ispisi();
18     }
19 };
```

U funkciji `ispisi_osnovnu()` zapis `COsnovna::ispisi()` izričito odabire funkciju osnovne klase, premda izvedena klasa ima funkciju jednakoga imena.

11.12.4 Pristup članu namespacea

Scope operator određuje i kojem namespaceu pripada neko ime. Primjerice, u zapisu

```
1 std::cout
```

`std` je namespace, a `cout` ime definirano u njemu. Tu ćemo uporabu podrobnije objasniti u odjeljku o namespaceu.

Zapamtite

Scope operator `::` ne pristupa članu određenoga objekta. Članovima objekta pristupa se operatorom točke `.` ili operatorom strelice `->`, dok `::` određuje područje kojem ime pripada.

11.13 Kvalifikator `volatile`

Kvalifikator **`volatile`** stavlja se uz varijablu čiju vrijednost može promijeniti nešto izvan redovitoga izvođenja programa, primjerice sklopovlje. Time se prevoditelju poručuje da pri čitanju ne smije pretpostaviti kako je prethodno pročitana vrijednost ostala nepromijenjena.

Opći oblik deklaracije je:

```
1 volatile tip ime;
```

Primjerice, varijabla povezana sa stanjem nekoga uređaja može biti deklarirana ovako:

```
1 volatile int stanje_uredjaja;
```

Vrijednost varijable `stanje_uredjaja` može se promijeniti neovisno o naredbama koje je program neposredno izvršio, pa se njezina trenutačna vrijednost mora provjeriti pri svakom pristupu.

11.13.1 Funkcijski članovi kvalificirani s `volatile`

Kvalifikator **`volatile`** može se primijeniti i na nestatički funkcijski član klase. Tada se piše iza zagrada s parametrima:

```
povratni_tip ime_funkcije(parametri) volatile;
```

Takav se funkcijski član može pozvati za objekt deklariran kao **`volatile`**. Ako funkcija samo čita objekt i ne mijenja njegove članove, uz **`volatile`** navodi se i **`const`**:

```
class CRegistar {
private:
    int vrijednost;

public:
    int procitaj() const volatile;
    void postavi(int nova_vrijednost) volatile;
};

int CRegistar::procitaj() const volatile {
    return vrijednost;
}

void CRegistar::postavi(int nova_vrijednost) volatile {
    vrijednost = nova_vrijednost;
}

int main() {
    volatile CRegistar registar;
    registar.postavi(5);
    int broj = registar.procitaj();
    return broj == 5 ? 0 : 1;
}
```

U deklaraciji `procitaj() const volatile` kvalifikator **`const`** znači da funkcija ne mijenja objekt, a **`volatile`** omogućuje njezino pozivanje za **`volatile`** objekt. Funkcija `postavi()` mijenja objekt pa nema kvalifikator **`const`**, ali mora imati **`volatile`** kako bi se mogla pozvati za objekt `registar`.

Vrijede sljedeća pravila:

- kvalifikator se piše iza zagrada s parametrima;
- mora se navesti i u deklaraciji i u definiciji funkcijskoga člana izvan klase;
- **`volatile`** objekt može pozvati samo funkcijske članove označene s **`volatile`** ili **`const volatile`**;
- statički funkcijski članovi ne mogu biti kvalificirani s **`volatile`** jer se ne pozivaju za pojedini objekt;
- konstruktor i destruktor ne kvalificiraju se s **`volatile`**;
- funkcijski član označen s **`volatile`** može se pozvati i za običan objekt; ako postoje obična i

`volatile` inačica iste funkcije, prevoditelj odabire onu koja odgovara objektu za koji se funkcija poziva.

Najčešće pogreške

Kvalifikator `volatile` ne štiti varijablu kada joj istodobno pristupa više threadova. Za takve zajedničke podatke upotrebljavaju se alati namijenjeni usklađivanju rada threadova, primjerice `std::atomic` i `std::mutex`.

11.14 Namespace

U većem programu različite biblioteke ili dijelovi programa mogu definirati klase, funkcije i varijable jednakih imena. `namespace` (hrv. imenski prostor) mehanizam je za organiziranje imena i izbjegavanje sukoba među jednakim nazivima u programu. U nastavku ćemo zadržati izvorni C++ naziv *namespace*.

Opći oblik definicije namespacea je:

```
namespace ime {
    // deklaracije i definicije
}
```

Članu namespacea pristupa se kvalificiranim imenom, odnosno scope operatorom `::`:

```
ime::clan
```

```
#include <iostream>

namespace prvi {
    int vrijednost = 5;
}

namespace drugi {
    double vrijednost = 3.1416;
}

int main() {
    std::cout << prvi::vrijednost << '\n';
    std::cout << drugi::vrijednost << '\n';
    return 0;
}
```

Output

```
5
3.1416
```

Obje varijable imaju ime `vrijednost`, ali njihova su puna imena `prvi::vrijednost` i `drugi::vrijednost`, pa nema pogreške višestruke definicije istoga imena u istom području dosega.

11.14.1 Ključna riječ `using`

`using`-deklaracija uvodi jedno određeno ime iz namespacea u trenutačno područje dosega:

```
using prvi::x;
```

`using`-direktiva čini sva imena iz odabranoga namespacea dostupnima za nekvalificirano traženje:

```
using namespace prvi;
```

Razliku prikazuje sljedeći primjer.

```
#include <iostream>

namespace prvi {
    int x = 5;
    int y = 10;
}

namespace drugi {
    double x = 3.1416;
    double y = 2.7183;
}

int main() {
    using prvi::x;          // using-deklaracija
    using namespace drugi; // using-direktiva

    std::cout << x << '\n';
    std::cout << y << '\n';
    std::cout << drugi::x << '\n';
    return 0;
}
```

Output

```
5
2.7183
3.1416
```

Nekvalificirano ime `x` odnosi se na izričito uvedeno ime `prvi::x`. Ime `y` pronalazi se u namespaceu `drugi` zahvaljujući **`using`**-direktivi. Puna kvalifikacija `drugi::x` i dalje se može upotrijebiti.

Doseg **using**-deklaracije i **using**-direktive počinje na mjestu na kojem su napisane i traje do kraja pripadajućega područja dosega. Zbog toga se njihovo djelovanje može ograničiti blokom:

```
{
    using namespace prvi;
    std::cout << x << '\n';
}

{
    using namespace drugi;
    std::cout << x << '\n';
}
```

Zapamtite

U zaglavnim datotekama ne treba pisati **using namespace** `std;` jer bi se sva imena iz namespacea `std` učinila dostupnima u svakoj datoteci koja uključi takvo zaglavlje. U većim programima preporučuje se upotreba kvalificiranih imena, primjerice `std::cout` i `std::string`, ili ciljane **using**-deklaracije u ograničenom području dosega.

11.14.2 Ugniježđeni namespacei i zamjenska imena

Namespacei mogu biti ugniježđeni. Od standarda C++17 mogu se zapisati sažeto:

```
namespace aplikacija::grafika {
    class Prozor {
        // ...
    };
}

aplikacija::grafika::Prozor glavni_prozor;
```

Dugom ili često korištenom namespaceu može se pridružiti kraće zamjensko ime:

```
namespace grafika = aplikacija::grafika;

grafika::Prozor pomocni_prozor;
```

Zamjensko ime ne stvara novi namespace, nego samo uvodi drugi naziv za postojeći.

Kratki sažetak

Klasa povezuje podatke i funkcije u jedinstven korisnički definirani tip. Zaštita članova omogućuje skrivanje implementacije, konstruktori uspostavljaju početno stanje, a destruktori završavaju životni vijek objekta. Preopterećenje operatora omogućuje prirodan rad s objektima, dok nasljeđivanje i virtualne funkcije omogućuju izgradnju hijerarhija i polimorfno ponašanje. Apstraktna klasa definira

zajedničko sučelje, ali se njezini objekti ne mogu izravno stvarati. Scope operator `::` određuje područje kojem ime pripada. Kvalifikator **`volatile`** upotrebljava se za podatke koji se mogu promijeniti izvan redovitoga izvođenja programa, a **`volatile`** objekt može pozivati odgovarajuće kvalificirane funkcijske članove. Namespacei organiziraju imena i sprječavaju njihove sukobe.

Pitanja za samoprovjeru

1. Koja je razlika između klase i objekta te koja su podrazumijevana prava pristupa u klasi i strukturi?
2. Kada se pozivaju konstruktor i destruktor?
3. Zašto je plitka kopija opasna kada klasa posjeduje dinamički dodijeljenu memoriju i što nalaže pravilo trojke?
4. Koja je razlika između **`private`** i **`protected`** pristupa pri nasljeđivanju?
5. Zašto se funkcija u osnovnoj klasi deklarira kao virtualna i što apstraktnu klasu čini apstraktnom?
6. U kojim se posebnim okolnostima koristi **`volatile`** i koje funkcijske članove može pozvati **`volatile`** objekt?
7. Čemu služi scope operator `::`? Objasnite njegovu uporabu uz globalno ime, član klase, član osnovne klase i član namespacea.
8. Koji problem rješavaju namespacei?
9. Koja je razlika između **`using`**-deklaracije i **`using`**-direktive te zašto se **`using namespace`** `std;` ne preporučuje u zaglavnim datotekama?

12. Predlošci (Templates)

Jedan postupak ne bi trebalo ponovno pisati samo zato što se promijenio tip podataka.

Problem koji rješavamo

Kako napisati jednu funkciju ili klasu koja može raditi s različitim tipovima podataka, a pritom zadržati provjeru tipova koju pruža C++?

Što ćete naučiti?

Nakon obrade ovoga poglavlja moći ćete:

- objasniti zašto se upotrebljavaju templates;
- napisati i pozvati jednostavan funkcijski template;
- objasniti kako prevoditelj određuje konkretan tip template parametra;
- upotrijebiti više template parametara;
- definirati klasni template i stvarati njegove objekte;
- definirati funkcijski član klasnoga templatea izvan klase;
- prepoznati vezu između templatesa i standardne biblioteke;
- objasniti zašto se definicije templatesa najčešće pišu u zaglavne datoteke.

12.1 Zašto su nam potrebni templates?

Pretpostavimo da želimo funkciju koja vraća veću od dvije vrijednosti. Ako radimo samo s cijelim brojevima, možemo napisati:

```
int veci(int a, int b) {
    return a > b ? a : b;
}
```

Za realne brojeve potrebna nam je gotovo jednaka funkcija:

```
double veci(double a, double b) {
    return a > b ? a : b;
}
```

Mogli bismo dodati i treću inačicu za znakove:

```
char veci(char a, char b) {
    return a > b ? a : b;
}
```

Sve tri funkcije provode jednak postupak. Razlikuje se samo tip parametara i povratne vrijednosti. Takvo ponavljanje otežava održavanje programa: promjenu postupka morali bismo unijeti u svaku inačicu.

`Template` (hrv. predložak) omogućuje da opći postupak napišemo jednom, a konkretan tip odredimo pri uporabi. U nastavku rabimo naziv *template* jer se upravo taj naziv pojavljuje u sintaksi programskoga jezika, dokumentaciji i porukama prevoditelja.

Kako razmišljati kao programer?

Kod koji se razlikuje samo po tipu podataka ne treba odmah kopirati. Najprije treba provjeriti može li se zajednički postupak izraziti templateom. Time se ne uklanja provjera tipova: prevoditelj i dalje stvara i provjerava funkciju za svaki konkretni tip koji se upotrijebi.

Pisanje funkcija i klasa koje rade s različitim tipovima naziva se *generičko programiranje*. Riječ *generičko* ovdje ne znači da program prihvaća bilo što bez provjere, nego da je isti postupak opisan neovisno o jednome unaprijed zadanom tipu.

12.2 Funkcijski template

Funkcijski template započinje retkom `template`. Funkciju `veci()` možemo zapisati ovako:

```
template <typename T>
T veci(T a, T b) {
    return a > b ? a : b;
}
```

U toj definiciji slovo `T` privremeni je naziv tipa. Ono ne znači `int`, `double` ili neki drugi određeni tip. Konkretan tip bit će poznat tek kada se template upotrijebi.

Tablica 12.1: Dijelovi definicije funkcijskoga templatea

Dio zapisa	Značenje
`template`	Započinje definiciju templatea.
`typename`	Označuje da parametar predstavlja tip podatka.
`T`	Naziv template parametra; uobičajen je, ali nije obvezan.
`T a, T b`	Parametri funkcije imat će tip koji bude pridružen oznaci `T`.
`T` ispred imena funkcije	Povratna vrijednost imat će isti tip `T`.

Naziv template parametra može biti i opisniji:

```
template <typename Tip>
Tip veci(Tip a, Tip b) {
    return a > b ? a : b;
}
```

U kratkim se primjerima najčešće upotrebljava `T`, dok opisniji naziv može biti jasniji kada template ima više parametara.

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

template <typename T>
T veci(T a, T b) {
    return a > b ? a : b;
}

int main() {
    cout << veci(4, 7) << '\n';
    cout << veci(3.5, 2.1) << '\n';
    cout << veci('a', 'z') << '\n';
    return 0;
}
```

Output

```
7
3.5
z
```

U programu postoji samo jedna definicija templatea, ali se on upotrebljava s trima različitim tipovima.

12.3 Kako prevoditelj određuje tip?

Pri pozivu funkcijskoga templatea prevoditelj promatra argumente. U pozivu

```
veci(4, 7)
```

oba su argumenta tipa **`int`**, pa prevoditelj zaključuje da `T` predstavlja **`int`**. Na temelju templatea stvara odgovarajuću funkciju, koju možemo zamisliti ovako:

```
int veci(int a, int b) {
    return a > b ? a : b;
}
```

Za poziv `veci(3.5, 2.1)` stvara se inačica za **`double`**, a za poziv `veci('a', 'z')` inačica za **`char`**. Taj se postupak naziva *instanciranje templatea*, a nastala funkcija *instanca templatea*.

Template, dakle, nije gotova funkcija za neki određeni tip. On je uputa prema kojoj prevoditelj stvara potrebnu funkciju kada naiđe na njezinu uporabu.

```
  veci(4, 7)     ⟶      veci<int>(int, int)
veci(3.5, 2.1)   ⟶  veci<double>(double, double)
veci('a', 'z')   ⟶     veci<char>(char, char)
```

Zapamtite

Prevoditelj stvara samo one konkretne inačice funkcije koje program doista upotrebljava. Ako se `veci()` nikada ne pozove s tipom **`char`**, inačica za taj tip nije potrebna.

12.3.1 Argumenti različitih tipova

U templateu `veci()` oba parametra imaju isti tip `T`. Zato je sljedeći poziv problematičan:

```
1 veci(4, 7.5);  // prvi je argument int, a drugi double
```

Iz prvoga argumenta prevoditelj zaključuje da bi `T` trebao biti **`int`**, a iz drugoga da bi trebao biti **`double`**. Budući da isti `T` ne može istodobno predstavljati dva tipa, tip se ne može jednoznačno odrediti.

Jedno je rješenje uskladiti tipove argumenata:

```
1 veci(4.0, 7.5);
```

Drugo je rješenje izričito zadati tip template parametra.

12.4 Izričito zadavanje tipa

Konkretan tip može se navesti unutar znakova < i >:

```
1 cout << veci<double>(4, 7.5) << '\n';
```

Ovdje smo izričito odredili da je `T` tip **`double`**. Cijeli broj 4 zato se prije poziva pretvara u vrijednost tipa **`double`**.

Izričito zadavanje dopušteno je i kada bi prevoditelj mogao sam odrediti tip:

```
1 veci<int>(4, 7);
```

Takav je zapis najčešće nepotrebno opširan. Tip izričito navodimo kada ga nije moguće odrediti iz argumenata ili kada namjerno želimo određenu pretvorbu.

Najčešće pogreške

Poziv `veci(4, 7.5)` ne znači automatski da će prevoditelj odabrati **`double`**. Pri određivanju template parametra ne traži se proizvoljno "najpogodniji zajednički tip". Ako oba funkcijska

parametra ovise o istome `T`, zaključci dobiveni iz argumenata moraju se slagati.

12.5 Više template parametara

Template može imati više parametara. Svaki od njih može predstavljati drugi tip:

```
template <typename T, typename U>
void ispisi_par(T prvi, U drugi) {
    cout << prvi << " - " << drugi << '\n';
}
```

Pri pozivu

```
ispisi_par("Godina", 2026);
```

parametri `T` i `U` ne moraju predstavljati isti tip. Prevoditelj ih određuje neovisno jedan o drugome.

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

template <typename T, typename U>
void ispisi_par(T prvi, U drugi) {
    cout << prvi << ": " << drugi << '\n';
}

int main() {
    ispisi_par("Broj bodova", 87);
    ispisi_par(string("Temperatura"), 23.5);
    ispisi_par('A', true);
    return 0;
}
```

Output

```
Broj bodova: 87
Temperatura: 23.5
A: 1
```

U posljednjem pozivu vrijednost `true` ispisuje se kao `1` jer nije uključena opcija `boolalpha`.

12.5.1 `typename` i `class`

U popisu template parametara zapisi

```
template <typename T>
```

i

```
template <class T>
```

imaju jednako značenje. Ključna riječ `class` na tom mjestu ne znači da `T` mora biti klasa. `T` može biti i ugrađeni tip, primjerice `int`.

U ovoj ćemo knjizi dosljedno rabiti `typename` jer jasnije pokazuje da parametar predstavlja tip.

12.6 Template ne radi nužno sa svakim tipom

Definicija funkcije `veci()` sadržava izraz `a > b`. Zato se taj template može uspješno upotrijebiti samo s tipom za koji je takva usporedba dopuštena.

Jednako vrijedi i za druge operacije. Sljedeći template zahtijeva da se dvije vrijednosti tipa `T` mogu zbrojiti:

```
template <typename T>
T zbroji(T a, T b) {
    return a + b;
}
```

Za tipove `int` i `double` operator + provodi aritmetičko zbrajanje. Za objekte klase `string` isti operator spaja tekstove. Za korisnički definiranu klasu poziv će biti valjan samo ako ona podržava potrebni operator.

```
cout << zbroji(3, 5) << '\n';
cout << zbroji(string("pro"), string("gram")) << '\n';
```

Output

```
8
program
```

Napomena

Template ne obećava da će svaki tip podržavati svaku naredbu u njegovoj definiciji. Pogreška se javlja kada pokušamo stvoriti konkretnu inačicu s tipom koji ne podržava potrebne operacije. Poruka prevoditelja pritom može biti dulja od poruke za običnu funkciju jer prikazuje i mjesto uporabe i postupak instanciranja templatea.

12.7 Klasni template

Templateom se može opisati i klasa. Klasni template koristan je kada klasa čuva ili obrađuje podatke čiji tip ne želimo unaprijed ograničiti.

Promotrimo klasu koja predstavlja točku u ravnini. Koordinate ponekad želimo pohraniti kao cijele, a ponekad kao realne brojeve. Umjesto dviju gotovo jednakih klasa možemo napisati jedan klasni template:

```
template <typename T>
class Tocka {
private:
    T x;
    T y;

public:
    Tocka(T x, T y) : x(x), y(y) {
    }

    void ispisi() const {
        cout << '(' << x << ", " << y << ')' << '\n';
    }
};
```

Oznaka T upotrebljava se za tip obiju koordinata, parametre konstruktora i odgovarajuće podatkovne članove. Za razliku od poziva funkcijskoga templatea, pri stvaranju objekta u osnovnom ćemo zapisu izričito navesti konkretan tip:

```
Tocka<int> a(3, 5);
Tocka<double> b(2.5, 4.8);
```

`Tocka<int>` i `Tocka<double>` dva su različita konkretna tipa nastala iz istoga klasnog templatea. Objekt jednoga tipa nije automatski objekt drugoga tipa.

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

template <typename T>
class Tocka {
private:
    T x;
    T y;

public:
    Tocka(T x, T y) : x(x), y(y) {
    }

    void ispisi() const {
        cout << '(' << x << ", " << y << ')' << '\n';
    }
```

```
};

int main() {
    Tocka<int> a(3, 5);
    Tocka<double> b(2.5, 4.8);

    a.ispisi();
    b.ispisi();
    return 0;
}
```

Output

```
(3, 5)
(2.5, 4.8)
```

Zapamtite

Naziv samoga templatea nije potpuni tip objekta. Potrebno je navesti i konkretne template argumente: `Tocka<int>`, `Tocka<double>` i slično.

12.7.1 Određivanje template argumenata klase u C++17

Od standarda C++17 prevoditelj u nekim slučajevima može odrediti template argumente klasnoga templatea iz argumenata konstruktora. Zato je za prethodnu klasu dopušten i zapis:

```
Tocka a(3, 5);      // prevoditelj određuje Tocka<int>
Tocka b(2.5, 4.8);  // prevoditelj određuje Tocka<double>
```

Ta se mogućnost naziva *class template argument deduction* (CTAD). U početnim ćemo primjerima ipak navoditi `Tocka<int>` i `Tocka<double>` jer se tada jasno vidi koji konkretan tip stvaramo, a određivanje tipa nije moguće u svakoj situaciji.

12.8 Funkcijski članovi klasnoga templatea

Kao i kod obične klase, funkcijski član može biti definiran unutar deklaracije klase ili izvan nje. Definicija izvan klasnoga templatea zahtijeva nešto dulji zapis.

Najprije u klasi ostavljamo deklaraciju funkcije:

```
template <typename T>
class Tocka {
private:
    T x;
    T y;
```

```
6
7  public:
8      Tocka(T x, T y);
9      void ispisi() const;
10 };
```

Zatim funkcijske članove definiramo ovako:

```
1  template <typename T>
2  Tocka<T>::Tocka(T x, T y) : x(x), y(y) {
3  }
4
5  template <typename T>
6  void Tocka<T>::ispisi() const {
7      cout << '(' << x << ", " << y << ')' << '\n';
8  }
```

U svakoj se definiciji ponavlja redak `template <typename T>` jer funkcijski član i dalje pripada templateu. Zapis `Tocka<T>` označuje klasu za opći tip `T`, a scope operator `::` određuje da konstruktor i funkcija `ispisi()` pripadaju toj klasi.

Tablica 12.2: Dijelovi definicije funkcijskoga člana izvan klasnoga templatea

Dio zapisa	Značenje
`template <typename T>`	Funkcija pripada templateu s parametrom `T`.
`Tocka<T>`	Označuje klasni template s njegovim template argumentom.
`::`	Scope operator povezuje funkciju s klasom `Tocka<T>`.
`ispisi() const`	Funkcijski član i njegovi kvalifikatori; moraju odgovarati deklaraciji u klasi.

Najčešće pogreške

Pri definiranju funkcijskoga člana izvan klasnoga templatea često se izostavi `<T>` pa se napiše `Tocka::ispisi()`. `Tocka` je naziv templatea, a `Tocka<T>` tip kojem funkcijski član pripada. Ispravan je zapis `Tocka<T>::ispisi()`.

12.9 Klasni template s više tipova

Klasni template također može imati više template parametara. Sljedeća klasa čuva dvije povezane vrijednosti koje ne moraju biti istoga tipa:

```
1  template <typename T, typename U>
2  class Par {
3  private:
4      T prvi;
```

```
    U drugi;

public:
    Par(T prvi, U drugi) : prvi(prvi), drugi(drugi) {
    }

    void ispisi() const {
        cout << prvi << ": " << drugi << '\n';
    }
};
```

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

template <typename T, typename U>
class Par {
private:
    T prvi;
    U drugi;

public:
    Par(T prvi, U drugi) : prvi(prvi), drugi(drugi) {
    }

    void ispisi() const {
        cout << prvi << ": " << drugi << '\n';
    }
};

int main() {
    Par<string, int> student("Ana", 24);
    Par<string, double> mjerenje("Temperatura", 23.5);

    student.ispisi();
    mjerenje.ispisi();
    return 0;
}
```

Output

```
Ana: 24
Temperatura: 23.5
```

U tipu `Par<string, int>` parametar `T` predstavlja `string`, a `U` predstavlja `int`. Redoslijed template argumenata odgovara redoslijedu template parametara.

Takav zapis priprema nas za tipove iz standardne biblioteke koji također imaju više template argumenata. Primjerice, spremnik `map<string, int>` povezuje ključeve tipa `string` s vrijednostima

tipa `int`.

12.10 Veza između templatesa i STL-a

Templates nisu samo način pisanja vlastitih funkcija i klasa. Na njima se temelji velik dio standardne biblioteke jezika C++. Spremnik `vector` klasni je template:

```
vector<int> brojevi;
vector<double> mjerenja;
vector<string> imena;
```

Ne postoje tri ručno napisane, nepovezane klase za cijele brojeve, realne brojeve i tekstove. Jedan klasni template `vector` upotrebljava se za stvaranje potrebnih konkretnih tipova.

I algoritmi standardne biblioteke najčešće su funkcijski templates. Funkcija `sort()` može sortirati različite spremnike i različite tipove elemenata ako su zadovoljeni uvjeti potrebni za uspoređivanje i pristup elementima:

```
sort(brojevi.begin(), brojevi.end());
sort(imena.begin(), imena.end());
```

U obama je pozivima naziv algoritma jednak. Konkretni tipovi određuju se iz predanih iteratora.

Prisjetimo se...

Zapis u šiljastim zagradama već smo vidjeli kod tipova standardne biblioteke. Nakon ovoga poglavlja možemo preciznije protumačiti `vector<int>`: `vector` je klasni template, a `int` je template argument prema kojem nastaje konkretan tip spremnika.

Razumijevanje templatesa zato je prirodan uvod u STL. Pri uporabi STL-a ne moramo znati sve pojedinosti njegova unutarnjeg koda, ali moramo razumjeti zašto se tipovi navode unutar znakova < i > te zašto neki algoritam za određeni tip radi, a za drugi ne radi.

12.11 Gdje se piše definicija templatea?

Kod obične funkcije u zaglavnu datoteku često pišemo samo deklaraciju, a definiciju smještamo u zasebnu `.cpp` datoteku. Template zahtijeva drukčiji uobičajeni raspored.

Prevoditelj pri instanciranju mora vidjeti cijelu definiciju templatea. Nije dovoljno da zna samo njegovu deklaraciju. Zbog toga se deklaracija i definicija templatea najčešće zajedno pišu u zaglavnu datoteku.

Datoteka `Tocka.h` može izgledati ovako:

```
#ifndef TOCKA_H
#define TOCKA_H

#include <iostream>
```

```
5
6  template <typename T>
7  class Tocka {
8  private:
9      T x;
10     T y;
11
12 public:
13     Tocka(T x, T y) : x(x), y(y) {
14     }
15
16     void ispisi() const {
17         std::cout << '(' << x << ", " << y << ')' << '\n';
18     }
19 };
20
21 #endif
```

Datoteka koja upotrebljava klasu uključuje to zaglavlje:

```
1  #include "Tocka.h"
2
3  int main() {
4      Tocka<double> polozaj(2.5, 4.8);
5      polozaj.ispisi();
6      return 0;
7  }
```

Napomena

U potpunim programima u ovoj knjizi rabimo `using namespace std;`. Zaglavna je datoteka iznimka: u nju tu direktivu ne stavljamo jer bi djelovala u svakoj izvornoj datoteci koja zaglavlje uključi. Zato je u primjeru `Tocka.h` upotrijebljeno kvalificirano ime `std::cout`.

Zapamtite

Praktično pravilo glasi: definicija templatea mora biti dostupna prevoditelju na mjestu njegove uporabe. Zato se templates najčešće u cijelosti definiraju u zaglavnim datotekama. Samo premještanje definicije u uobičajenu `.cpp` datoteku često završava pogreškom pri povezivanju programa.

Mogući su i napredniji načini odvajanja definicija te izričito instanciranje, ali oni nisu potrebni za osnovnu uporabu templatesa.

12.12 Najčešće pogreške

Najčešće pogreške

1. **Izostavljen redak `template <typename T>`.** Bez toga retka `T` nije poznat naziv tipa.
2. **Argumenti dovode do različitih zaključaka o istome tipu.** Poziv `veci(3, 4.5)` nije valjan za template u kojem oba parametra imaju tip `T`. Tipove treba uskladiti ili izričito zadati template argument.
3. **Izostavljen je zapis `<T>` uz naziv klasnoga templatea.** U definiciji funkcijskoga člana izvan klase pišemo `Tocka<T>::ispisi()`, a ne `Tocka::ispisi()`.
4. **Tip ne podržava potrebnu operaciju.** Template `veci()` zahtijeva usporedbu operatorom `>`, a `ispisi()` zahtijeva da se vrijednost može poslati u izlazni tok operatorom `<<`.
5. **Definicija je skrivena u zasebnoj `.cpp` datoteci.** Prevoditelj na mjestu uporabe tada može vidjeti deklaraciju, ali ne i kod potreban za stvaranje konkretne inačice.
6. **Pretpostavlja se da `T` mora biti ugrađeni tip.** Template parametar može predstavljati i klasu, primjerice `string` ili korisnički definiranu klasu, ako taj tip podržava operacije koje template zahtijeva.

Kada prevoditelj prijavi pogrešku povezanu s templateom, korisno je najprije provjeriti tri pitanja:

1. Koji je konkretan tip pridružen svakom template parametru?
2. Koje operacije definicija templatea pokušava provesti nad tim tipom?
3. Podržava li konkretan tip svaku od tih operacija?

Takav je postupak često jasniji od čitanja cijele, ponekad vrlo duge poruke prevoditelja od početka do kraja.

12.13 Kratki sažetak

Što sada znamo?

- Template omogućuje da funkciju ili klasu napišemo neovisno o jednome unaprijed zadanom tipu podataka.
- U zapisu `template <typename T>` oznaka `T` predstavlja tip koji će se odrediti pri uporabi.
- Prevoditelj iz funkcijskoga templatea stvara potrebne konkretne funkcije. Taj se postupak naziva instanciranje.
- Tip funkcijskoga templatea često se određuje iz argumenata, ali se može zadati i izričito, primjerice `veci<double>(4, 7.5)`.

- Template može imati više parametara, pa različiti dijelovi funkcije ili klase mogu upotrebljavati različite tipove.
- Konkretni tip klasnoga templatea zapisuje se zajedno s njegovim argumentima, primjerice `Tocka<int>`.
- Template radi samo s tipovima koji podržavaju operacije upotrijebljene u njegovoj definiciji.
- `vector` je klasni template, a mnogi su STL algoritmi funkcijski templates.
- Definicije templatesa najčešće se nalaze u zaglavnim datotekama kako bi bile dostupne prevoditelju na mjestu uporabe.

Pitanja za samoprovjeru

1. Koji problem rješava funkcijski template?
2. Što predstavlja `T` u zapisu `template <typename T>`?
3. Kako prevoditelj određuje konkretan tip funkcijskoga templatea?
4. Zašto poziv `veci(3, 4.5)` nije valjan ako oba parametra funkcije imaju tip `T`?
5. Kada se template argument zadaje izričito i kako izgleda takav poziv?
6. Jesu li `typename` i `class` istovrijedni u popisu template parametara?
7. Zašto template `veci()` ne mora raditi sa svakim mogućim tipom?
8. Po čemu se klasni template razlikuje od konkretne klase nastale iz njega?
9. Zašto se pri definiranju funkcijskoga člana izvan klasnoga templatea piše `Tocka<T>::ispisi()`?
10. Kakva je veza između klasnoga templatea i tipa `vector<int>`?
11. Zašto se definicije templatesa najčešće nalaze u zaglavnim datotekama?

13. Rad s datotekama

Što ćete naučiti?

Nakon ovoga poglavlja moći ćete otvoriti, provjeriti, čitati i zapisivati tekstne i binarne datoteke pomoću tokova, odabrati odgovarajući način otvaranja, protumačiti stanje toka te postaviti položaj za čitanje ili pisanje.

Za rad s datotekama C++ pruža sljedeće klase iz zaglavlja `<fstream>`:

- `std::ofstream` za zapisivanje u datoteku,
- `std::ifstream` za čitanje iz datoteke,
- `std::fstream` za čitanje iz datoteke i zapisivanje u nju.

13.1 Otvaranje i zatvaranje datoteke

Rad s datotekom obično obuhvaća tri faze:

1. otvaranje datoteke,
2. čitanje podataka ili zapisivanje podataka,
3. zatvaranje datoteke.

Datoteka se može otvoriti pri stvaranju objekta toka ili naknadno funkcijskim članom `open()`. Pojednostavnjeni oblik deklaracije toga funkcijskog člana jest:

```
1 void open(const char* imeDatoteke, ios_base::openmode nacin);
```

Parametar `imeDatoteke` određuje datoteku koja se otvara, a parametar `nacin` način na koji se datoteka otvara. Najčešći načini prikazani su u tablici 13.1.

Tablica 13.1: Načini otvaranja datoteke

Način	Značenje
`ios::in`	Otvaranje za čitanje.
`ios::out`	Otvaranje za zapisivanje.
`ios::ate`	Nakon otvaranja početni se položaj postavlja na kraj datoteke; poslije ga je moguće promijeniti.
`ios::app`	Svako se zapisivanje obavlja na kraju datoteke.
`ios::trunc`	Pri otvaranju se odbacuje postojeći sadržaj datoteke.
`ios::binary`	Datoteka se otvara u binarnome načinu rada.

Načini iz tablice 13.1 mogu se kombinirati bitovnim operatorom ILI |. Primjerice, datoteka `primjer.bin` otvara se za binarno zapisivanje na njezin kraj ovako:

```
ofstream datoteka;
datoteka.open("primjer.bin", ios::out | ios::app | ios::binary);
```

Zadani načini otvaranja pojedinih vrsta tokova prikazani su u tablici 13.2.

Tablica 13.2: Zadani načini otvaranja datotečnih tokova

Klasa	Zadani način otvaranja
`ofstream`	`ios::out`
`ifstream`	`ios::in`
`fstream`	`ios::in | ios::out`

Zadani se način primjenjuje kada se argument načina otvaranja izostavi. Iako je zadani argument klase `ofstream` samo `ios::out`, otvaranje postojeće datoteke samo za zapisivanje u pravilu odbacuje njezin dotadašnji sadržaj. Za dodavanje na kraj treba navesti `ios::app`.

Je li datoteka otvorena, može se provjeriti funkcijskim članom `is_open()`:

```
bool is_open() const;
```

Funkcija vraća **`true`** ako je tok povezan s otvorenom datotekom, a inače vraća **`false`**. Stanje toka nakon otvaranja ili nakon pojedine operacije može se provjeriti i izravno, primjerice uvjetom **`if`** `(!datoteka)`.

Datoteka se izričito zatvara funkcijskim članom `close()`:

```
void close();
```

Objekt datotečnoga toka automatski zatvara datoteku kada prestane postojati, odnosno pri izlasku iz svojega područja važenja. To je primjer načela RAII objašnjenoga u poglavlju o dinamičkoj memoriji.

13.2 Zapisivanje u datoteku

```
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
    ofstream datoteka("primjer.txt");

    if (!datoteka)
    {
        cout << "Pogreška pri otvaranju datoteke.\n";
        return 1;
    }
```

```
15
16      datoteka << "Ovo je prvi redak teksta.\n";
17      datoteka << "Ovo je drugi redak teksta.\n";
18
19      return 0;
20  }
```

Program ne ispisuje retke na zaslon, nego ih zapisuje u datoteku `primjer.txt`. Nakon njegova izvođenja datoteka sadržava:

Output

```
Ovo je prvi redak teksta.
Ovo je drugi redak teksta.
```

13.3 Čitanje iz datoteke

```
1   #include <fstream>
2   #include <iostream>
3   #include <string>
4
5   using namespace std;
6
7   int main()
8   {
9       ifstream datoteka("primjer.txt");
10
11      if (!datoteka)
12      {
13          cout << "Pogreška pri otvaranju datoteke.\n";
14          return 1;
15      }
16
17      string redak;
18      while (getline(datoteka, redak))
19      {
20          cout << redak << '\n';
21      }
22
23      return 0;
24  }
```

Output

```
Ovo je prvi redak teksta.
Ovo je drugi redak teksta.
```

U uvjetu petlje provjerava se uspjeh same operacije čitanja. Sljedeća se petlja često susreće, ali nije ispravna pa je ovdje navedena samo kao komentirani kod:

```
// while (!datoteka.eof())
// {
//     getline(datoteka, redak);
//     cout << redak << '\n';
// }
```

Funkcijski član `eof()` opisuje stanje toka nakon prethodne operacije; ne predviđa hoće li sljedeće čitanje uspjeti. Ako se stanje provjeri prije čitanja, petlja može ući u tijelo, a da čitanje unutar tijela zatim ne uspije. Tada se može ponovno obraditi prethodni redak ili neki drugi nevaljan sadržaj. Ispravno je provjeravati rezultat same operacije čitanja, kao u prethodnome primjeru.

13.4 Stanja toka i oporavak nakon pogreške

Datotečni tok pamti rezultat prethodnih operacija. Najvažniji funkcijski članovi za ispitivanje stanja prikazani su u tablici 13.3.

Tablica 13.3: Provjera stanja ulazno-izlaznoga toka

Funkcijski član	Značenje
`good()`	Nijedna zastavica pogreške nije postavljena.
`eof()`	Pri čitanju je dosegnut kraj datoteke.
`fail()`	Prethodna operacija nije uspjela, primjerice zbog nevaljanoga formata podatka.
`bad()`	Nastala je ozbiljna ulazno-izlazna pogreška.
`clear()`	Ponovno postavlja zastavice stanja kako bi se tok mogao dalje upotrebljavati.

Funkcija `clear()` uklanja zastavice pogreške, ali ne popravlja uzrok pogreške i ne uklanja nevaljani podatak iz ulaza. Nakon pogreške oblikovanja potrebno je ukloniti ili preskočiti sporni podatak. Sljedeći program čita cijele brojeve, prijavljuje nevaljane zapise i nastavlja s obradom:

```
#include <fstream>
#include <iostream>
#include <string>

using namespace std;

int main()
{
```

```
    ifstream datoteka("brojevi.txt");
    if (!datoteka)
    {
        cerr << "Datoteka se ne moze otvoriti.\n";
        return 1;
    }

    while (true)
    {
        int broj;
        if (datoteka >> broj)
        {
            cout << broj << '\n';
        }
        else if (datoteka.eof())
        {
            break;
        }
        else
        {
            datoteka.clear();
            string nevaljaniZapis;
            datoteka >> nevaljaniZapis;
            cerr << "Preskacem: " << nevaljaniZapis << '\n';
        }
    }

    return 0;
}
```

Ako je postavljena zastavica `badbit`, nastavak rada često nije moguć jer uzrok može biti kvar uređaja ili druga ozbiljna ulazno-izlazna pogreška. U takvom slučaju program treba prijaviti problem i prekinuti tu operaciju.

13.5 Pozicioniranje u datoteci

Ulazni tok ima položaj s kojega će se obaviti sljedeće čitanje, a izlazni tok položaj na kojem će se obaviti sljedeće pisanje. Funkcijski članovi `tellg()` i `tellp()` vraćaju trenutačni položaj, dok ga `seekg()` i `seekp()` mijenjaju. Slova `g` i `p` potječu od izraza *get position* i *put position*.

Položaj se može zadati u odnosu na početak (`ios::beg`), trenutačni položaj (`ios::cur`) ili kraj datoteke (`ios::end`). Sljedeći program u binarnome načinu određuje veličinu datoteke u bajtovima, a zatim se vraća na njezin početak:

```
#include <fstream>
#include <iostream>

using namespace std;

```

```
int main()
{
    ifstream datoteka("primjer.bin", ios::binary);
    if (!datoteka)
    {
        cerr << "Datoteka se ne moze otvoriti.\n";
        return 1;
    }

    datoteka.seekg(0, ios::end);
    streampos velicina = datoteka.tellg();
    datoteka.seekg(0, ios::beg);

    cout << "Velicina datoteke: " << velicina << " bajtova\n";
    return 0;
}
```

Prije pozicioniranja nakon neuspjeloga čitanja treba pozvati `clear()`, jer tok sa zadržanom zastavicom pogreške neće uredno izvršiti novu operaciju. Pri izmjeni čitanja i pisanja u istome objektu `fstream` treba između operacija postaviti položaj ili obaviti odgovarajuću sinkronizaciju toka.

13.6 Binarne datoteke i funkcije `read()` i `write()`

U tekstnoj datoteci vrijednosti se pretvaraju u znakove. U binarnoj se datoteci mogu zapisati njihove bajtovne reprezentacije. Funkcije `write()` i `read()` primaju adresu prvoga bajta i broj bajtova koji se zapisuju ili čitaju. Sljedeći program zapisuje i ponovno čita jednostavan zapis:

```
#include <fstream>
#include <iostream>
#include <type_traits>

using namespace std;

struct Mjerenje
{
    int oznaka;
    double vrijednost;
};

static_assert(is_trivially_copyable_v<Mjerenje>);

int main()
{
    Mjerenje izlazni{17, 23.5};

    ofstream izlaz("mjerenje.bin", ios::binary);
    if (!izlaz)
        return 1;

```

```
    izlaz.write(reinterpret_cast<const char*>(&izlazni),
                sizeof izlazni);
    if (!izlaz)
        return 1;
    izlaz.close();

    Mjerenje ulazni{};
    ifstream ulaz("mjerenje.bin", ios::binary);
    if (!ulaz)
        return 1;

    ulaz.read(reinterpret_cast<char*>(&ulazni), sizeof ulazni);
    if (!ulaz)
        return 1;

    cout << ulazni.oznaka << ' ' << ulazni.vrijednost << '\n';
    return 0;
}
```

Najčešće pogreške

Sirovo zapisivanje objekta nije općenito prenosiv format datoteke. Raspored bajtova, poravnanje i popunjavanje strukture te prikaz cjelobrojnih i realnih tipova mogu se razlikovati među platformama i prevoditeljima. Funkcije `read()` i `write()` na ovaj se način smiju primijeniti samo na prikladne, trivijalno kopirljive tipove. Za dugotrajnu razmjenu podataka treba definirati vlastiti format i svako polje zapisivati u dogovorenom obliku.

Kratki sažetak

Datotečni tok povezuje program s datotekom. Način otvaranja određuje čita li se datoteka, zapisuje li se u nju, dodaje li se sadržaj na njezin kraj ili se obrađuje u binarnome obliku. Uspjeh čitanja treba provjeravati samom operacijom čitanja. Stanje toka pokazuje zašto operacija nije uspjela, položaj toka može se mijenjati funkcijama `seekg()` i `seekp()`, a binarni se sadržaj čita i zapisuje funkcijama `read()` i `write()`.

Pitanja za samoprovjeru

1. Koje su tri osnovne faze rada s datotekom?
2. Koja je razlika između načina `ios::ate` i `ios::app`?
3. Što znače `ios::trunc` i `ios::binary`?
4. Zašto pri čitanju datoteke nije dobro upotrebljavati petlju `while (!datoteka.eof())`?
5. Koja je razlika između stanja `eof()`, `fail()` i `bad()`?
6. Zašto nakon funkcije `clear()` ponekad treba preskočiti nevaljani ulazni podatak?

7. Čemu služe funkcije `tellg()`, `seekg()`, `tellp()` i `seekp()`?
8. Zašto sirovi binarni zapis strukture nije nužno prenosiv između različitih sustava?

14. Osnove STL-a: vector i algoritmi

Ne učimo STL zato da bismo zapamtili nove klase. Učimo ga zato da bismo manje pisali, manje griješili i više razmišljali o rješavanju problema.

Problem koji rješavamo

Kako spremiti proizvoljan broj elemenata bez unaprijed poznate veličine polja te kako ih sortirati, pronaći ili prebrojati bez pisanja vlastitih algoritama?

Što ćete naučiti?

Nakon ovoga poglavlja moći ćete:

- stvoriti objekt tipa `std::vector`;
- dodavati i uklanjati elemente;
- pristupati elementima i prolaziti kroz spremnik;
- koristiti iteratore na osnovnoj razini;
- primijeniti osnovne STL algoritme;
- povezati lambda funkcije s algoritmima STL-a.

14.1 Zašto nam je potreban vector?

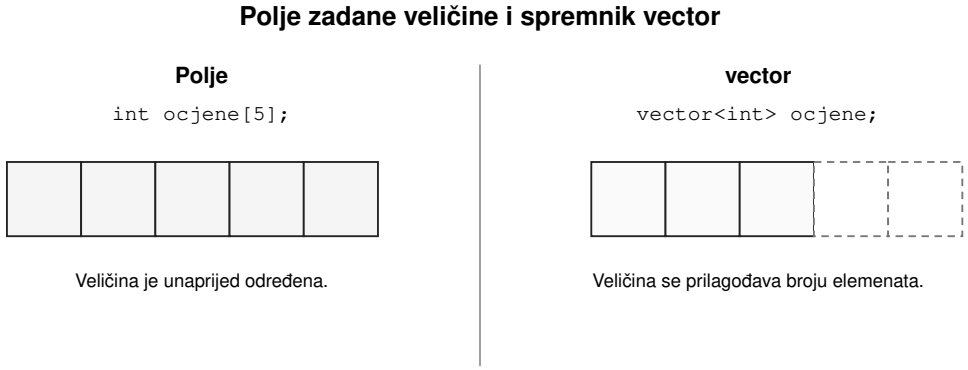
Prisjetimo se...

Do sada smo za spremanje više podataka koristili polja:

```
1 int ocjene[5];
```

Takvo polje može spremiti točno pet elemenata. Njegova veličina mora biti poznata unaprijed i tijekom izvođenja programa ne može se jednostavno povećati.

Što ako unaprijed ne znamo koliko će elemenata korisnik unijeti? U takvoj je situaciji prikladniji spremnik `std::vector`. On sam upravlja svojom veličinom i automatski se prilagođava dodavanju ili uklanjanju elemenata.



Slika 14.1: Razlika između polja zadane veličine i spremnika `vector`

Za rad sa spremnikom `vector` potrebno je uključiti zaglavlje:

```
#include <vector>
```

Najjednostavnija deklaracija izgleda ovako:

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

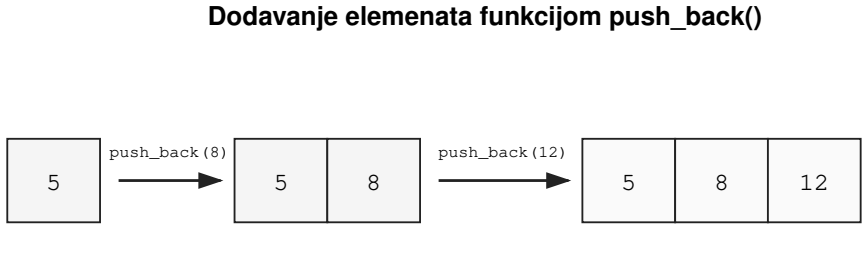
vector<int> brojevi;
```

14.2 Dodavanje i uklanjanje elemenata

Funkcija `push_back()` dodaje element na kraj spremnika:

```
vector<int> brojevi;

brojevi.push_back(5);
brojevi.push_back(8);
brojevi.push_back(12);
```



Slika 14.2: Postupno dodavanje elemenata u spremnik `vector`

Funkcija `pop_back()` uklanja posljednji element:

```
brojevi.pop_back();
```

Funkcije `pop_back()`, `front()` i `back()` smiju se pozvati samo kada spremnik nije prazan. Prije njihova poziva stanje se može provjeriti funkcijom `empty()`:

```
if (!brojevi.empty())
{
    cout << brojevi.back() << endl;
    brojevi.pop_back();
}
```

14.3 Pristup elementima

Elementima se može pristupiti operatorom `[]`:

```
cout << brojevi[0] << endl;
```

Može se koristiti i funkcija `at()`:

```
cout << brojevi.at(0) << endl;
```

Tablica 14.1: Usporedba operatora `[]` i funkcije `at()`

Način pristupa	Prednost	Provjera granica
`brojevi[i]`	Kraći zapis.	Ne provjerava je li indeks valjan.
`brojevi.at(i)`	Sigurniji pristup elementu.	Provjerava granice.

14.4 Veličina i stanje spremnika

Tablica 14.2: Osnovni članovi klase `vector`

Član klase	Opis
`push_back()`	Dodaje element na kraj spremnika.
`pop_back()`	Uklanja posljednji element.
`size()`	Vraća broj elemenata.
`capacity()`	Vraća broj elemenata za koje je trenutačno rezervirana pohrana.
`reserve(n)`	Rezervira pohranu za najmanje `n` elemenata bez promjene veličine.
`empty()`	Provjerava je li spremnik prazan.
`clear()`	Uklanja sve elemente.
`front()`	Vraća prvi element.
`back()`	Vraća posljednji element.
`begin()`	Vraća iterator na prvi element.
`end()`	Vraća iterator iza posljednjega elementa.

14.4.1 Veličina, kapacitet i rezerviranje pohrane

Veličina (`size`) broj je elemenata koji trenutačno postoje u spremniku, dok je kapacitet (`capacity`) broj elemenata za koje je već pribavljena pohrana. Kapacitet može biti veći od veličine kako se pri svakom dodavanju ne bi morala pribavljati nova pohrana.

```
vector<int> brojevi;
brojevi.reserve(100);

cout << brojevi.size() << '\n';      // 0
cout << brojevi.capacity() << '\n';  // najmanje 100
```

Funkcija `reserve()` ne stvara elemente i ne mijenja `size()`. Funkcija `resize()`, naprotiv, mijenja broj elemenata; pri povećanju stvara nove elemente, a pri smanjenju uklanja višak. Rezerviranje je korisno kada približno znamo koliko će se elemenata dodati jer može smanjiti broj realokacija i kopiranja ili premještanja elemenata.

14.5 Prolazak kroz elemente

```
for (int broj : brojevi)
{
    cout << broj << " ";
}
```

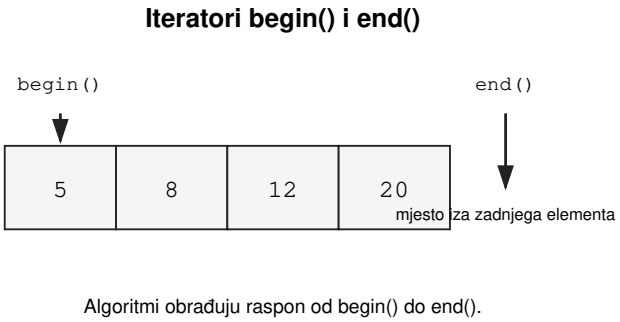
Ako elemente želimo mijenjati, koristimo referencu:

```
for (int& broj : brojevi)
{
    broj *= 2;
}
```

14.6 Iteratori

Iterator možemo zamisliti kao pokazivač koji prolazi kroz elemente spremnika.

```
for (auto it = brojevi.begin();
     it != brojevi.end();
     ++it)
{
    cout << *it << " ";
}
```



Slika 14.3: Iteratori `begin()` i `end()` u spremniku `vector`

Napomena

Za jednostavan prolazak kroz sve elemente početnicima je najčešće preglednija petlja `range-based for`. Iteratori su ipak važni jer ih koriste mnogi algoritmi standardne biblioteke.

Najčešće pogreške

Kada dodavanje elemenata zahtijeva realokaciju, svi postojeći iteratori, pokazivači i reference na elemente spremnika `vector` postaju nevaljani. Ako realokacije nema, `push_back()` ipak poništava iterator `end()`. Uklanjanje ili umetanje unutar spremnika u pravilu poništava iteratore i reference na mjestu promjene i iza njega. Zato nakon operacije koja mijenja spremnik ne treba nastaviti rabiti ranije spremljeni iterator bez provjere pravila te operacije. Poziv `reserve()` prije većega broja dodavanja može smanjiti broj realokacija, ali i sam poništava

sve iteratore i reference ako doista poveća kapacitet.

14.7 Osnovni STL algoritmi

Algoritmi se nalaze u zaglavlju:

```
#include <algorithm>
```

Tablica 14.3: Osnovni STL algoritmi

Algoritam	Namjena
`sort()`	Sortira elemente.
`find()`	Traži zadanu vrijednost.
`count()`	Prebrojava pojavljivanja zadane vrijednosti.
`max_element()`	Pronalazi najveći element.
`min_element()`	Pronalazi najmanji element.
`reverse()`	Obrće redoslijed elemenata.
`for_each()`	Primjenjuje operaciju na svaki element.

14.7.1 Sortiranje

```
sort(brojevi.begin(), brojevi.end());
```

14.7.2 Traženje

```
auto it = find(brojevi.begin(),
               brojevi.end(),
               8);

if (it != brojevi.end())
    cout << "Element je pronaden." << endl;
else
    cout << "Element nije pronaden." << endl;
```

14.7.3 Brojanje

```
int koliko = count(brojevi.begin(),
                   brojevi.end(),
```

```
                    5);
```

14.7.4 Najveći i najmanji element

```
auto najveci = max_element(brojevi.begin(), brojevi.end());
auto najmanji = min_element(brojevi.begin(), brojevi.end());

if (najveci != brojevi.end())
{
    cout << *najveci << endl;
    cout << *najmanji << endl;
}
```

Za prazan raspon obje funkcije vraćaju iterator jednak `brojevi.end()`. Prije dereferenciranja rezultata zato treba provjeriti da nije jednak `end()`.

14.7.5 Obrtanje redoslijeda

```
reverse(brojevi.begin(), brojevi.end());
```

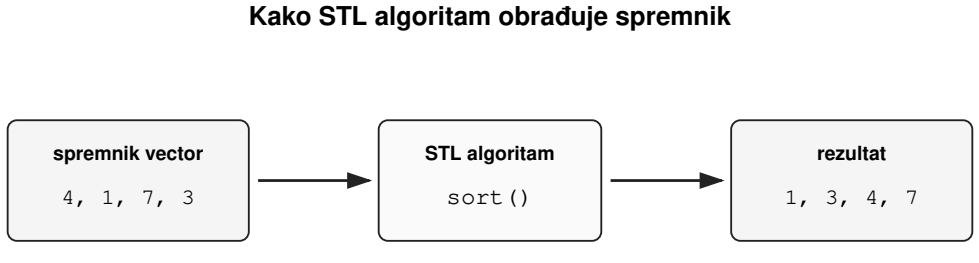
14.8 Lambda funkcije i STL

Silazno sortiranje:

```
sort(brojevi.begin(),
     brojevi.end(),
     [](int a, int b)
     {
         return a > b;
     });
```

Ispis svakoga elementa:

```
for_each(brojevi.begin(),
         brojevi.end(),
         [](int broj)
         {
             cout << broj << " ";
         });
```



Slika 14.4: Od spremnika preko algoritma do rezultata

14.9 STL u jednoj stranici

Tablica 14.4: Koji alat odabrati?

Želim...	**Koristim...**
Spremiti proizvoljan broj elemenata.	`vector`
Dodati element na kraj.	`push_back()`
Ukloniti posljednji element.	`pop_back()`
Saznati broj elemenata.	`size()`
Unaprijed rezervirati pohranu.	`reserve()`
Saznati trenutačni kapacitet.	`capacity()`
Proći kroz sve elemente.	`range-based for`
Sortirati elemente.	`sort()`
Pronaći element.	`find()`
Prebrojati elemente određene vrijednosti.	`count()`
Pronaći najveći ili najmanji element.	`max_element()`, `min_element()`
Obrnuti redoslijed.	`reverse()`
Primijeniti operaciju na svaki element.	`for_each()`

Najčešće pogreške

- Zaboravljeno uključivanje zaglavlja `<vector>` ili `<algorithm>`.
- Pristup elementu izvan granica spremnika.
- Pozivanje `pop_back()` nad praznim spremnikom.

- Pozivanje `front()` ili `back()` nad praznim spremnikom.
- Dereferenciranje rezultata funkcije `find()` bez provjere.
- Dereferenciranje rezultata `max_element()` ili `min_element()` za prazan raspon.
- Zaboravljanje operatora * pri pristupu elementu preko iteratora.
- Miješanje vrijednosti `size()` s najvećim valjanim indeksom.
- Uporaba iteratora ili reference koja je poništena realokacijom spremnika.

Zapamtite

- `vector` automatski upravlja svojom veličinom.
- Elementi se dodaju funkcijom `push_back()`.
- `size()` označuje broj elemenata, a `capacity()` raspoloživu pohranu.
- `reserve()` može smanjiti broj realokacija, ali ne stvara elemente.
- Funkcija `at()` provjerava granice, a operator `[]` ne.
- `begin()` označava prvi element, a `end()` mjesto iza zadnjega.
- Algoritmi STL-a rade nad rasponom elemenata zadanih iteratorima.
- Standardne algoritme treba koristiti umjesto ponovnoga pisanja već riješenih postupaka.

Kratki sažetak

U ovome poglavlju upoznali smo spremnik `std::vector`, koji omogućuje spremanje proizvoljnoga broja elemenata bez unaprijed zadane veličine. Prikazani su dodavanje, uklanjanje, pristup i prolazak kroz elemente, odnos veličine i kapaciteta te osnovna uloga i valjanost iteratora. Upoznali smo i najčešće algoritme standardne biblioteke: `sort()`, `find()`, `count()`, `max_element()`, `min_element()`, `reverse()` i `for_each()`.

Što sada znamo?

Nakon ovoga poglavlja znate:

- koristiti `std::vector`;
- dodavati i uklanjati elemente;
- razlikovati veličinu i kapacitet te primijeniti `reserve()`;
- pristupati elementima i prolaziti kroz spremnik;
- razumjeti osnovnu ulogu iteratora i prepoznati njihovo poništavanje;
- sortirati, pronaći i prebrojati elemente;

- pronaći najveći i najmanji element;
- povezati lambda funkcije s algoritmima STL-a.

Pitanja za samoprovjeru

1. Koji problem rješava spremnik `vector`?
2. Koje je zaglavlje potrebno uključiti?
3. Čemu služi funkcija `push_back()`?
4. Koja je razlika između operatora `[]` i funkcije `at()`?
5. Što vraća funkcija `size()`?
6. Koja je razlika između `size()`, `capacity()`, `reserve()` i `resize()`?
7. Što predstavljaju iteratori `begin()` i `end()`?
8. Kada dodavanje elementa može poništiti sve iteratore i reference na elemente?
9. Zašto se rezultat funkcije `find()` uspoređuje s `end()`?
10. Koji algoritam sortira elemente?
11. Kako se pronalazi najveći element?
12. Zašto rezultat funkcije `max_element()` treba provjeriti prije dereferenciranja?
13. Čemu služi algoritam `reverse()`?
14. Kako se lambda funkcija može primijeniti u algoritmu `sort()`?

15. Lambda funkcije

Što ćete naučiti?

Nakon ovoga poglavlja moći ćete objasniti sintaksu i način rada lambda funkcija, razlikovati hvatanje varijabli vrijednošću i referencom te primijeniti ključnu riječ `mutable`. Također ćete moći napisati generičku lambda funkciju i upotrijebiti lambde kao funkcijske objekte u algoritmima standardne biblioteke.

Lambda funkcije (*engl. lambda expressions*) uvedene su u standardu C++11 kako bi omogućile definiranje kratkih funkcija izravno na mjestu njihove uporabe. Prije njihova uvođenja bilo je potrebno napisati zasebnu funkciju ili funkcijski objekt čak i kada se određena funkcija koristila samo jedanput. Takav je pristup povećavao količinu programskoga koda i smanjivao njegovu preglednost.

Lambda funkcija nema vlastito ime u uobičajenom smislu. Preciznije, lambda izraz stvara neimenovani funkcijski objekt jedinstvenoga tipa. Taj se objekt može odmah pozvati, spremiti u varijablu ili proslijediti drugoj funkciji. Osobito je koristan pri radu s algoritmima standardne biblioteke, primjerice pri sortiranju, pretraživanju, brojanju i obradi elemenata spremnika podataka.

15.1 Sintaksa lambda funkcije

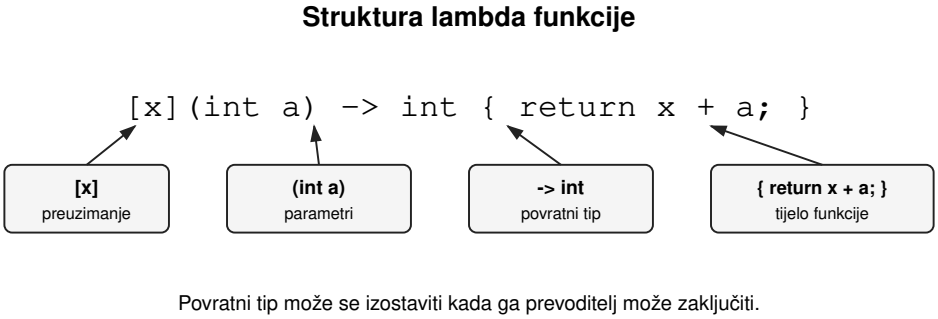
Opći oblik lambda funkcije je:

```
[hvatanje (capture)](parametri) -> povratni_tip
{
    // tijelo funkcije
};
```

Pojedini dijelovi imaju sljedeće značenje:

- uglate zagrade određuju koje se varijable iz okoline hvataju;
- oble zagrade sadrže parametre lambda funkcije;
- povratni tip određuje tip vrijednosti koju funkcija vraća;
- tijelo funkcije sadrži naredbe koje se izvršavaju pri njezinu pozivu.

Povratni se tip najčešće ne mora navesti jer ga prevoditelj može zaključiti iz izraza koji se vraća.



Slika 15.1: Osnovni dijelovi lambda funkcije

15.2 Prva lambda funkcija

```
#include <iostream>
using namespace std;
int main()
{
    auto pozdrav = []()
    {
        cout << "Pozdrav svijete!" << endl;
    };
    pozdrav();
    return 0;
}
```

Ključna riječ `auto` omogućuje prevoditelju da automatski odredi tip objekta u koji je lambda funkcija spremljena. Funkcija se poziva pomoću naziva objekta i oblih zagrada, jednako kao obična funkcija.

Lambda se može i odmah izvršiti:

```
[]()
{
    cout << "Lambda funkcija je odmah izvrsena." << endl;
}();
```

15.3 Lambda funkcije s parametrima

```
auto zbroj = [](int a, int b)
{
    return a + b;
};
cout << zbroj(8, 12) << endl;
```

Argumenti se mogu prenositi po vrijednosti, po referenci ili po konstantnoj referenci, jednako kao kod običnih funkcija.

```
auto povecaj = [](int &x)
{
    x++;
};
int broj = 10;
povecaj(broj);
cout << broj << endl;
```

15.4 Povratna vrijednost

```
auto kvadrat = [](int x)
{
    return x * x;
};
```

Povratni tip može se i izričito navesti iza operatora `->`:

```
auto podijeli = [](double a, double b) -> double
{
    return a / b;
};
```

15.5 Generičke lambda funkcije

Od standarda C++14 parametri lambda funkcije mogu biti deklarirani ključnom riječi `auto`. Takva se lambda funkcija naziva generičkom jer može primati različite tipove podataka.

```
auto kvadrat = [](auto x)
{
    return x * x;
};
cout << kvadrat(7) << endl;
cout << kvadrat(3.5) << endl;
```

15.6 Hvatanje varijabli iz okoline

Lambda funkcija može koristiti varijable deklarirane izvan svojega tijela. Način njihova hvatanja određuje se izrazom unutar uglatih zagrada.

Tablica 15.1: Načini hvatanja varijabli iz okoline

Zapis	Značenje
`[]`	Ne hvata nijednu varijablu iz okoline.
`[x]`	Hvata varijablu `x` vrijednošću.
`[&x]`	Hvata varijablu `x` referencom.
`[=]`	Potrebne lokalne varijable automatskoga trajanja zadano hvata vrijednošću.
`[&]`	Potrebne lokalne varijable automatskoga trajanja zadano hvata referencom.
`[x, &y]`	Varijablu `x` hvata vrijednošću, a `y` referencom.
`[=, &x]`	Potrebne lokalne varijable zadano hvata vrijednošću, osim `x`, koju hvata referencom.
`[&, x]`	Potrebne lokalne varijable zadano hvata referencom, osim `x`, koju hvata vrijednošću.
`[this]`	Hvata pokazivač na trenutačni objekt.

Zadani načini `[=]` i `[&]` ne znače da lambda kopira ili veže svaku varijablu koja je vidljiva u okolini. Hvataju se samo lokalni objekti automatskoga trajanja koji su lambda funkciji potrebni. Globalne i statičke varijable ne hvataju se; dostupne su prema uobičajenim pravilima vidljivosti. Zapis `[this]` hvata pokazivač na trenutačni objekt, pa životni vijek toga objekta mora nadživjeti svaku kasniju uporabu lambde.

15.6.1 Hvatanje vrijednošću

```
int broj = 10;
auto ispisi = [broj]()
{
    cout << broj << endl;
};
broj = 20;
ispisi();
```

Lambda ispisuje vrijednost 10 jer je kopija varijable stvorena u trenutku definiranja lambda funkcije.

Ako je promjena kopije potrebna, koristi se ključna riječ `mutable`:

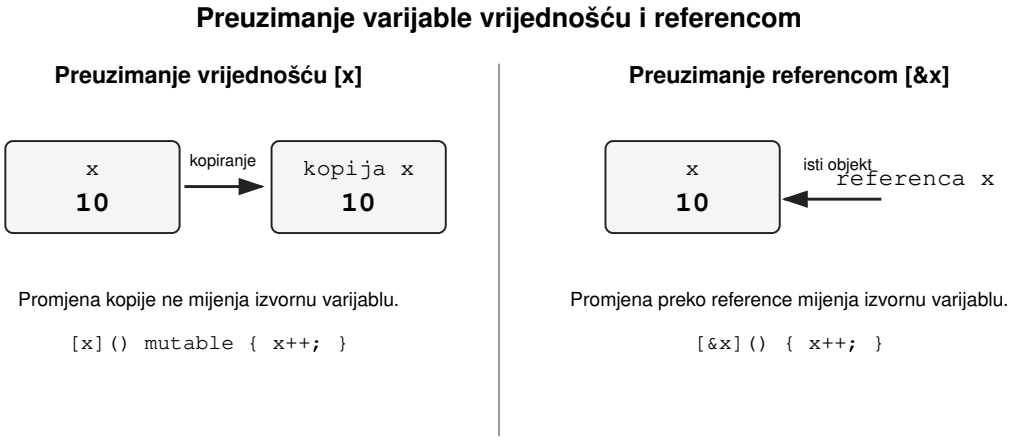
```
int broj = 10;
auto povecaj_kopiju = [broj]() mutable
{
    broj++;
    cout << broj << endl;
};
povecaj_kopiju();
cout << broj << endl;
```

15.6.2 Hvatanje referencom

```
int broj = 10;
auto povecaj = [&broj]()
{
    broj++;
};
povecaj();
cout << broj << endl;
```

Napomena

Pri hvatanju referencom potrebno je paziti na životni vijek objekta. Lambda ne smije koristiti referencu na varijablu koja je prestala postojati.



Slika 15.2: Razlika između hvatanja varijable vrijednošću i referencom

15.7 Lambda funkcije u standardnim algoritmima

15.7.1 Sortiranje

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> brojevi = {4, 1, 7, 3, 9};
    sort(brojevi.begin(), brojevi.end(),
         [](int a, int b)
         {
```

```
            return a > b;
        });
    for (int broj : brojevi)
        cout << broj << " ";
    return 0;
}
```

15.7.2 Obrada svakoga elementa

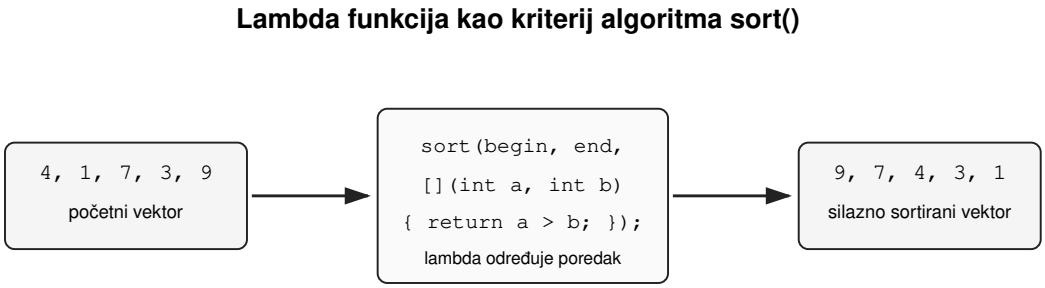
```
for_each(brojevi.begin(), brojevi.end(),
         [](int broj)
         {
             cout << broj << " ";
         });
```

15.7.3 Brojanje elemenata koji zadovoljavaju uvjet

```
int broj_parnih = count_if(
    brojevi.begin(), brojevi.end(),
    [](int broj)
    {
        return broj % 2 == 0;
    });
```

15.7.4 Pretvorba elemenata

```
vector<int> kvadrati(brojevi.size());
transform(
    brojevi.begin(), brojevi.end(), kvadrati.begin(),
    [](int broj)
    {
        return broj * broj;
    });
```



Slika 15.3: Uporaba lambda funkcije kao kriterija algoritma `sort()`

15.8 Usporedba obične i lambda funkcije

Obična funkcija:

```
bool veci(int a, int b)
{
    return a > b;
}
sort(brojevi.begin(), brojevi.end(), veci);
```

Lambda funkcija:

```
sort(brojevi.begin(), brojevi.end(),
     [](int a, int b)
     {
         return a > b;
     });
```

Obična funkcija prikladnija je kada se ista operacija koristi na više mjesta ili kada je složenija. Lambda funkcija prikladnija je za kratku operaciju koja se koristi lokalno.

Najčešće pogreške

- Zaboravljanje uglatih zagrada na početku lambda izraza.
- Pokušaj korištenja vanjske varijable bez odgovarajućega hvatanja.
- Pokušaj mijenjanja varijable uhvaćene vrijednošću bez ključne riječi `mutable`.
- Hvatanje varijable referencom i korištenje nakon što je varijabla prestala postojati.
- Nepotrebno hvatanje svih varijabli zapisima `[=]` ili `[&]`.
- Izostavljanje povratne vrijednosti kada algoritam očekuje logički rezultat.

- Pisanje prevelikih i složenih lambda funkcija umjesto zasebne imenovane funkcije.

Zapamtite

- Lambda funkcija omogućuje definiranje kratke funkcije na mjestu njezine uporabe.
- Uglate zagrade određuju koje se varijable iz okoline hvataju.
- Varijabla uhvaćena vrijednošću kopira se u lambda funkciju.
- Varijabla uhvaćena referencom može se izravno mijenjati.
- Ključna riječ `mutable` dopušta promjenu kopija uhvaćenih vrijednošću.
- Lambda funkcije često se koriste s algoritmima `sort()`, `for_each()`, `count_if()` i `transform()`.

Kratki sažetak

U ovome poglavlju upoznali smo lambda funkcije kao neimenovane funkcijske objekte koji se definiraju izravno na mjestu uporabe. Objašnjeni su njihova sintaksa, parametri, povratne vrijednosti i generičke lambde. Posebna je pozornost posvećena hvatanju lokalnih varijabli iz okoline vrijednošću i referencom te uporabi ključne riječi `mutable`. Prikazana je praktična primjena lambda funkcija u algoritmima standardne biblioteke, uključujući sortiranje, obradu, brojanje i pretvorbu elemenata.

Pitanja za samoprovjeru

1. Što je lambda funkcija?
2. U kojem su standardu C++-a uvedene lambda funkcije?
3. Koju ulogu imaju uglate zagrade u lambda izrazu?
4. Kako se deklarira lambda funkcija koja ne prima parametre?
5. Kako se lambda funkcija sprema u varijablu?
6. Kada je potrebno izričito navesti povratni tip?
7. Što je generička lambda funkcija?
8. Koja je razlika između hvatanja varijable vrijednošću i referencom?
9. Što znače zapisi `[=]` i `[&]`?
10. Čemu služi ključna riječ `mutable`?
11. Koja opasnost postoji pri hvatanju varijable referencom?
12. Zašto su lambda funkcije korisne u algoritmu `sort()`?
13. Kada je bolje napisati običnu funkciju umjesto lambda funkcije?

14. Navedite tri algoritma standardne biblioteke kojima se može proslijediti lambda funkcija.

16. Pametni pokazivači

Pametni pokazivač ne uklanja potrebu za razumijevanjem pokazivača. Njegova je zadaća učiniti vlasništvo nad resursom jasnim i vezati njegovo oslobađanje uz životni vijek objekta.

Problem koji rješavamo

Kako osigurati da se dinamički stvoren objekt oslobodi točno jedanput, čak i ako funkcija završi ranije, dođe do iznimke ili program ima više mogućih izlaznih putova?

Što ćete naučiti?

Nakon ovoga poglavlja moći ćete:

- objasniti odnos načela RAII i automatskoga oslobađanja resursa;
- razlikovati vlasništvo od običnoga pristupa objektu;
- stvoriti i koristiti `std::unique_ptr`, `std::shared_ptr` i `std::weak_ptr`;
- prenijeti jedinstveno vlasništvo funkcijom `std::move()`;
- objasniti brojenje vlasnika i problem cikličkoga vlasništva;
- odabrati prikladnu vrstu pokazivača prema odnosu vlasništva.

16.1 Zašto su potrebni pametni pokazivači?

Prisjetimo se...

Objekt stvoren new-izrazom ostaje u dinamičkoj memoriji sve dok se ne uništi odgovarajućim delete-izrazom. Obični pokazivač samo čuva adresu; iz njegova tipa nije vidljivo tko je odgovoran za oslobađanje objekta.

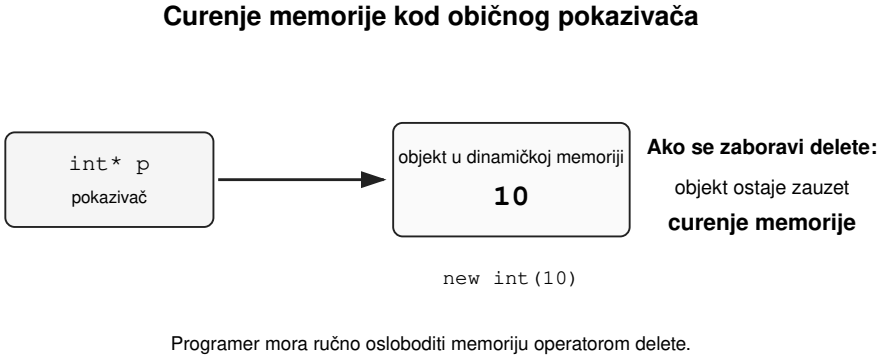
U sljedećem je primjeru odgovornost na programeru:

```
int* p = new int(10);

cout << *p << endl;

delete p;
p = nullptr;
```

Ako se `delete` izostavi, gubi se mogućnost pravilnoga oslobađanja zauzete memorije i nastaje *curenje memorije*. Ako se isti objekt oslobodi više puta ili se pokazivač dereferencira nakon oslobađanja, ponašanje programa nije definirano.



Slika 16.1: Curenje memorije nakon gubitka pristupa objektu koji je stvoren new-izrazom

Pametni pokazivači su objekti standardne biblioteke koji čuvaju obični pokazivač i opisuju vlasništvo nad objektom. Kada vlasnik prestane postojati, njegov destruktor obavlja potrebno oslobađanje. Definirani su u zaglavlju `<memory>`.

Napomena

Pametni pokazivači znatno smanjuju opasnost od curenja memorije i dvostrukoga oslobađanja, ali ne mogu ispraviti svaki pogrešan odnos vlasništva. Primjerice, ciklus objekata koji se međusobno posjeduju pomoću `shared_ptr` i dalje može zadržati memoriju zauzetom.

16.2 Načelo RAII

Pametni pokazivači temelje se na načelu RAII (*Resource Acquisition Is Initialization*). Resurs se pridružuje objektu koji njime upravlja. Konstruktor uspostavlja valjano stanje, a destruktor oslobađa resurs kada objekt izađe iz dosega. Isto se načelo u standardnoj biblioteci primjenjuje i na datoteke, zaključavanja i druge resurse, ne samo na memoriju.

Zbog RAII-ja oslobađanje nije potrebno ručno ponavljati na svakome mogućem izlazu iz funkcije. To je osobito važno kada funkcija sadrži više naredbi `return` ili može završiti iznimkom.

16.3 Jedinstveno vlasništvo: `std::unique_ptr`

Objekt tipa `std::unique_ptr<T>` predstavlja jedinstveno vlasništvo: u nekome trenutku samo jedan takav pokazivač upravlja određenim objektom tipa `T`. Kada vlasnik izađe iz dosega, objekt se automatski uništava.

Objekt je najjednostavnije stvoriti funkcijom `std::make_unique()`, koja je dostupna od standarda C++14:

```
1 #include <iostream>
```

```cpp
#include <memory>

using namespace std;

int main()
{
    unique_ptr<int> broj = make_unique<int>(10);

    cout << "Vrijednost: " << *broj << endl;

    return 0;
}
```

Output

```
Vrijednost: 10
```

Operator * pristupa objektu, a operator -> njegovim članovima:

```cpp
class Student
{
public:
    string ime;

    Student(const string& novo_ime) : ime(novo_ime) {}
};

unique_ptr<Student> student =
    make_unique<Student>("Ana");

cout << student->ime << endl;
```

16.3.1 Prijenos vlasništva funkcijom `std::move()`

`unique_ptr` ne može se kopirati jer bi kopiranjem nastala dva vlasnika istoga objekta. Vlasništvo se može *prenijeti*:

```cpp
#include <iostream>
#include <memory>
#include <utility>

using namespace std;

int main()
{
    unique_ptr<int> p1 = make_unique<int>(25);
    unique_ptr<int> p2;

```

```
    p2 = move(p1);  // Vlasništvo prelazi na p2; p1 ostaje prazan.

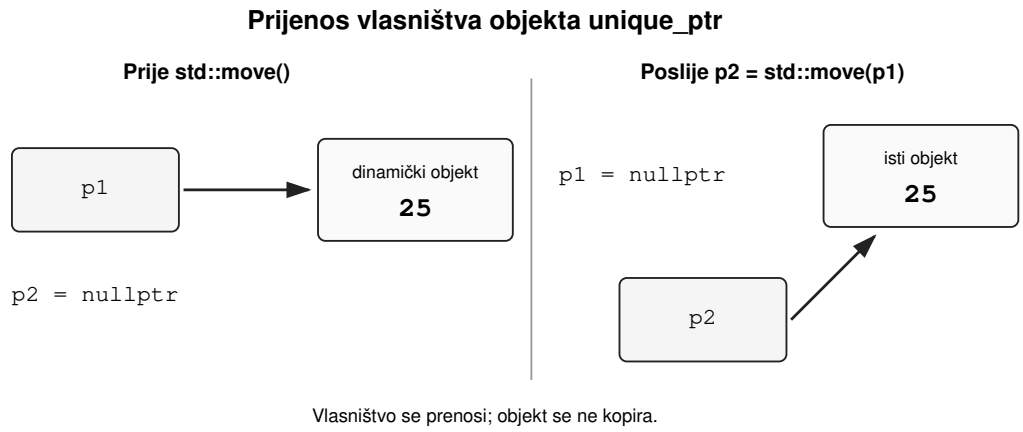
    cout << "p1 je prazan: " << boolalpha
         << (p1 == nullptr) << endl;
    cout << "Vrijednost preko p2: " << *p2 << endl;

    return 0;
}
```

Output

```
p1 je prazan: true
Vrijednost preko p2: 25
```

Nakon prijenosa `p1` ostaje valjan objekt tipa `unique_ptr`, ali više ne posjeduje objekt. Ne smije se dereferencirati dok mu se ponovno ne pridruži vlasništvo.



Slika 16.2: Prijenos vlasništva funkcijom `std::move()`

16.3.2 Dinamička polja

`unique_ptr` može upravljati i dinamičkim poljem. U tom se slučaju u tipu pišu uglate zagrade, a elementima se pristupa operatorom `[]`:

```
unique_ptr<int[]> brojevi = make_unique<int[]>(5);

for (int i = 0; i < 5; i++)
    brojevi[i] = i * i;
```

Pri uništavanju pokazivača automatski se primjenjuje odgovarajući oblik oslobađanja za polje. Ako je veličina promjenjiva i potrebne su uobičajene operacije nad spremnikom, `std::vector` je najčešće prikladniji izbor.

16.4 Zajedničko vlasništvo: `std::shared_ptr`

`std::shared_ptr<T>` koristi se kada više dijelova programa doista dijeli vlasništvo nad istim objektom. Svi pokazivači povezani s istim objektom dijele upravljački blok u kojem se, među ostalim, vodi broj aktivnih vlasnika. Objekt se uništava kada nestane posljednji vlasnik.

```cpp
#include <iostream>
#include <memory>

using namespace std;

int main()
{
    shared_ptr<int> p1 = make_shared<int>(50);

    cout << "Broj vlasnika: " << p1.use_count() << endl;

    {
        shared_ptr<int> p2 = p1;

        cout << "Vrijednost: " << *p2 << endl;
        cout << "Broj vlasnika: " << p1.use_count() << endl;
    }

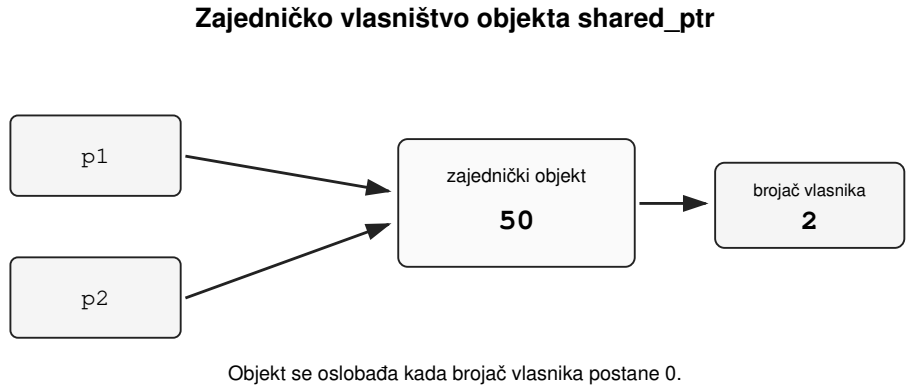
    cout << "Broj vlasnika: " << p1.use_count() << endl;

    return 0;
}
```

Output

```
Broj vlasnika: 1
Vrijednost: 50
Broj vlasnika: 2
Broj vlasnika: 1
```

Objekte je u pravilu najbolje stvarati funkcijom `std::make_shared()`. Zapis je pregledan, a objekt i upravljački blok obično se mogu smjestiti u jednu memorijsku dodjelu.



Slika 16.3: Dva pokazivača `shared_ptr` s vlasništvom nad istim objektom

Napomena

Funkcija `use_count()` korisna je za učenje i dijagnostiku, ali se na njezinoj vrijednosti ne treba temeljiti logika programa. U složenijem ili višenitnom programu broj vlasnika može se promijeniti neposredno nakon provjere.

16.5 Promatranje bez vlasništva: `std::weak_ptr`

`std::weak_ptr<T>` promatra objekt kojim upravlja `shared_ptr`, ali nije njegov vlasnik i ne sprječava njegovo uništavanje. Ne može se izravno dereferencirati.

Za siguran pristup poziva se `lock()`. Ako objekt još postoji, `lock()` vraća privremeni `shared_ptr`; u protivnome vraća prazan pokazivač:

```
#include <iostream>
#include <memory>

using namespace std;

int main()
{
    weak_ptr<int> promatrac;

    {
        shared_ptr<int> vlasnik = make_shared<int>(100);
        promatrac = vlasnik;

        if (shared_ptr<int> p = promatrac.lock())
            cout << "Vrijednost: " << *p << endl;
    }

    cout << "Objekt je istekao: " << boolalpha
```

```
19          << promatrac.expired() << endl;
20
21      return 0;
22  }
```

Output

```
Vrijednost: 100
Objekt je istekao: true
```

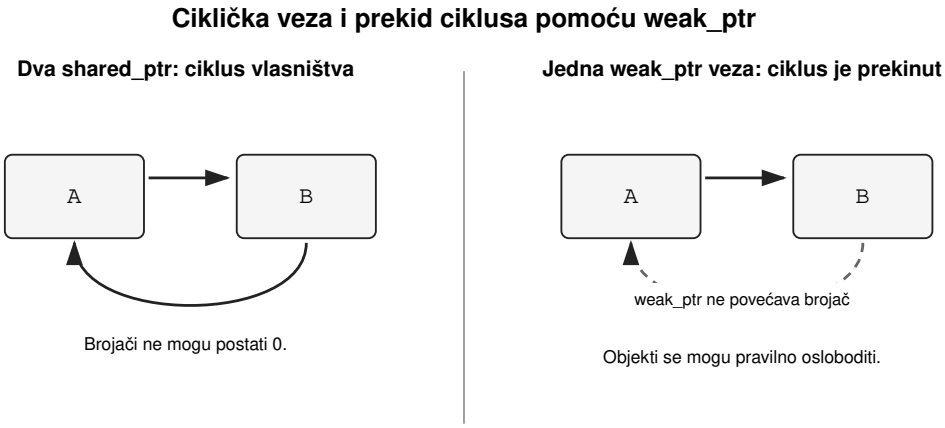
16.6 Cikličko vlasništvo

Ako dva objekta međusobno čuvaju `shared_ptr`, svaki objekt održava drugoga na životu. Broj vlasnika zato ne pada na nulu ni kada vanjski pokazivači nestanu:

```
1   class B;
2
3   class A
4   {
5   public:
6       shared_ptr<B> b;
7   };
8
9   class B
10  {
11  public:
12      shared_ptr<A> a;
13  };
```

Veza koja ne predstavlja vlasništvo treba biti slaba:

```
1   class B;
2
3   class A
4   {
5   public:
6       shared_ptr<B> b;
7   };
8
9   class B
10  {
11  public:
12      weak_ptr<A> a;
13  };
```



Slika 16.4: Prekid cikličkoga vlasništva pokazivačem `weak_ptr`

16.7 Vlasništvo i pristup nisu isto

Nije svaki pristup objektu ujedno i vlasništvo. Ako je životni vijek objekta već osiguran drugdje, funkcija često treba primiti referencu ili obični pokazivač:

```
void ispisi(const Student& student);
void obradi(Student* student);  // Pokazivač može biti nullptr.
```

Pametni pokazivač u sučelju funkcije treba koristiti kada je prijenos ili dijeljenje vlasništva dio značenja te funkcije. Primjerice, primanje `unique_ptr<T>` po vrijednosti znači da funkcija preuzima vlasništvo.

Funkcija `get()` vraća obični pokazivač bez prijenosa vlasništva. Takav pokazivač vrijedi samo dok postoji objekt koji posjeduje resurs:

```
Student* obicni = student.get();
```

16.8 Koji pokazivač odabrati?

Tablica 16.1: Odabir prema odnosu vlasništva

Vrsta	Značenje	Tipična primjena
`unique_ptr`	Jedan vlasnik; može se premjestiti, ali ne kopirati.	Jedan jasno određen vlasnik objekta.
`shared_ptr`	Više vlasnika; objekt živi dok postoji barem jedan.	Stvarno zajedničko vlasništvo bez prirodnoga pojedinačnog vlasnika.
`weak_ptr`	Promatranje objekta kojim upravlja `shared_ptr`.	Ne-vlasnička veza i prekid cikličkoga vlasništva.
Referenca ili obični pokazivač	Pristup bez vlasništva.	Pozivanje funkcije dok je životni vijek objekta osiguran drugdje.

Zapamtite

Ako je potrebno vlasništvo, najprije razmotrite `unique_ptr`. `shared_ptr` nemojte birati samo zato što omogućuje kopiranje, nego tek kada više dijelova programa doista mora dijeliti životni vijek objekta.

16.9 Česte operacije

Tablica 16.2: Česte operacije pametnih pokazivača

Operacija	Značenje
`make_unique<T>(...)`	Stvara objekt i vraća `unique_ptr<T>`.
`make_shared<T>(...)`	Stvara objekt i vraća `shared_ptr<T>`.
`operator*`, `operator->`	Pristup objektu i njegovim članovima.
`get()`	Vraća obični pokazivač bez prijenosa vlasništva.
`reset()`	Napušta trenutačno vlasništvo ili preuzima novi objekt.
`release()`	Samo za `unique_ptr`: predaje obični pokazivač i više ne oslobađa objekt.
`use_count()`	Vraća trenutačni broj vlasnika povezanih s `shared_ptr`.
`lock()`	Pokušava iz `weak_ptr` dobiti privremeni `shared_ptr`.
`expired()`	Provjerava je li promatrani objekt već uništen.

Najčešće pogreške

- Ručno pozivanje `delete` nad objektom kojim upravlja pametni pokazivač.
- Stvaranje pametnoga pokazivača na lokalni objekt ili objekt koji nije dobiven odgovarajućom dinamičkom dodjelom.
- Pokušaj kopiranja objekta tipa `unique_ptr`.
- Dereferenciranje premještenoga ili praznoga pametnog pokazivača.
- Stvaranje dvaju nepovezanih objekata `shared_ptr` iz istoga običnog pokazivača.
- Uporaba `shared_ptr` kada je vlasnik samo jedan.
- Izravno dereferenciranje objekta tipa `weak_ptr` umjesto poziva `lock()`.
- Zadržavanje običnoga pokazivača dobivenoga funkcijom `get()` nakon prestanka života vlasnika.
- Poziv `release()` bez jasnoga plana tko će poslije osloboditi predani objekt.
- Cikličko vlasništvo ostvareno isključivo pokazivačima `shared_ptr`.

Kratki sažetak

Pametni pokazivači povezuju vlasništvo nad dinamički stvorenim objektom s životnim vijekom objekta koji njime upravlja. `unique_ptr` izražava jedinstveno vlasništvo, `shared_ptr` zajedničko vlasništvo, a `weak_ptr` pristup bez produljivanja životnoga vijeka. Funkcije `make_unique()` i `make_shared()` preporučeni su načini stvaranja objekata. Obični pokazivač ili referenca i dalje su prikladni za pristup objektu kada ne prenose vlasništvo.

Što sada znamo?

Nakon ovoga poglavlja znate:

- objasniti načelo RAII i pojam vlasništva;
- koristiti tri standardne vrste pametnih pokazivača;
- prenijeti objekt tipa `unique_ptr`;
- prepoznati zajedničko i cikličko vlasništvo;
- sigurno pristupiti objektu koji promatra `weak_ptr`;
- razlikovati vlasnički pametni pokazivač od ne-vlasničke reference ili običnoga pokazivača.

Pitanja za samoprovjeru

1. Koji problem nastaje ako se objekt stvoren new-izrazom ne oslobodi?
2. Što znači načelo RAII?
3. Koje je zaglavlje potrebno uključiti za rad s pametnim pokazivačima?
4. Zašto se `unique_ptr` ne može kopirati?
5. Što se događa s izvornim pokazivačem nakon prijenosa funkcijom `std::move()`?

6. Kako `unique_ptr` upravlja dinamičkim poljem?
7. Kada je opravdana uporaba pokazivača `shared_ptr`?
8. Kada se objekt kojim upravljaju pokazivači `shared_ptr` uništava?
9. Zašto se logika programa ne treba temeljiti na funkciji `use_count()`?
10. Zašto se `weak_ptr` ne može izravno dereferencirati?
11. Što vraća funkcija `lock()` ako objekt više ne postoji?
12. Kako nastaje cikličko vlasništvo i kako se prekida?
13. Kada je referenca ili obični pokazivač bolji izbor od pametnoga pokazivača?
14. Koja opasnost nastaje nakon poziva funkcije `release()`?

A. Zadatci za laboratorijske vježbe

A.1 Vježba 1

C++

1. Napisati jednostavni C++ program koji na zaslonu ispisuje poruku:
   Moje ime i prezime je _____
2. Oktalni broj 0131764 prikaži dekadski.
3. Heksadekadski broj **0xB3F4** prikaži dekadski.
4. Binarni broj 11111111 prikaži dekadski.
5. Binarni broj 11111111 prikaži oktalno.
6. Binarni broj 11111111 prikaži heksadekadski.
7. Binarni broj 10101010 prikaži dekadski.
8. Binarni broj 10101010 prikaži oktalno.
9. Binarni broj 10101010 prikaži heksadekadski.
10. Ako se u binarnom broju 11111111 promijene vrijednost MSB bita koji dekadski broj sada predstavlja?
11. Ako se u binarnom broju 11111111 promijene vrijednost LSB bita koji dekadski broj sada predstavlja?
12. Dekadski broj 255 prikaži binarno.
13. Dekadski broj 170 prikaži binarno.
14. Iz tablice ASCII koda odgovori koja je dekadska vrijednost znaka 0:

15. Iz tablice ASCII koda odgovori koja je heksadekadska vrijednost znaka 0:

16. Iz tablice ASCII koda odgovori koja je dekadska vrijednost znaka @:

17. Iz tablice ASCII koda odgovori koja je heksadekadska vrijednost znaka @:

18. Iz tablice ASCII koda odgovori koja je dekadska vrijednost znaka A:

19. Iz tablice ASCII koda odgovori koja je heksadekadska vrijednost znaka A:

20. Iz tablice ASCII koda odgovori koja je dekadska vrijednost znaka a:

21. Iz tablice ASCII koda odgovori koja je heksadekadska vrijednost znaka a:

22. Za koliko se prema ASCII kodu razlikuju velika i mala slova (dekadski)?

Arduino

1. U Arduino primjeru blink promjeniti kod na način da LED dioda svijetli 5 sekundi i bude ugašena 2 sekunde. Priložiti kod.

Bilješke uz vježbu 1

A.2 Vježba 2

C++

1. Korištenjem Microsof Visual C++ a napisati program koji ispisuje vaše osnovne podatke na ekran:

   IME, PREZIME, STUDIJ, ŠKOLSKU GODINU, DATUM RODENJA, BROJ
   POLOŽENIH ISPITA, na sljedeći način:

   Ime: ime studenta
   Prezime: prezime studenta
   STUDIJ: ELEKTROTEHNIKA
   SKOLSKA GODINA: 2017/2018
   Dan, mjesec, godina rodjenja: 01.01.1901
   Broj polozenih ispita: 1 od 5

2. Napisati program koji deklarira dvije cjelobrojne varijable a i b te jednu realnu varijablu c (tipa float) i jednu realnu varijablu d (tipa double), kao i jednu znakovnu varijablu s (tipa char).
   Napisati program koji se ponaša na sljedeći način:

   Unesi a=2 // cin»
   Unesi b=3 // cin»
   Zbroj a i b je c:
   c=a+b=5
   Razlika a i b je c:
   c=a-b=-1
   Umnožak a i b je c:
   c=a*b=6
   Cjelobrojni rezultat dijeljenja a i b je c:
   c=a/b=0
   Realni rezultat dijeljenja a i b je c:
   c=a/b=0.666667
   Potencija broja a sa b je c: (koristeći pow(x,y) funkciju
   c=8
   Drugi korijen od b je d koristeći sqrt(x) funkciju:
   d=1.73205
   Unesi znak za zbrajanje:
   Znak za zbrajanje je +

   Napomena: za unos podataka koristiti ulazni tok cin » , a za matematičke funkcije biblioteku cmath.

3. Pridružite odgovarajućim tipovima podataka broj byte-ova koji zauzimaju u memoriji.

   - char
   - short
   - long

- int
- float
- double
- long double
- bool
- wchar_t

Ponuđeni odgovori: 1; 2; 4; 8; 10.

4. Pridružite odgovarajućim tipovima podataka odgovarajući raspon vrijednosti

- char
- short
- long
- int
- float
- double
- long double
- bool
- wchar_t

Ponuđeni odgovori: signed: −128 do +127 unsigned: 0 do 255; signed: −32768 do +32767 unsigned: 0 do 65535; signed: −2147483648 do +2147483647 unsigned: 0 do 4294967295; 3.4e +/−38 (preciznost 7 znamenki); 1.7e +/−308 (preciznost 15 znamenki); 1.2e +/−4932 (preciznost 19 znamenki); 1 ili 0; 16 bita (65536 znakova).

5. Analiziraj slijedeći program i odgovori na pitanja.

```
#include<iostream>
using namespace std;
int main(){
int a,b,c,d,e,f,u;
unsigned short int g=10, h, k=65535,j=65525;
short m=32767;
short int n(0);
a=10;
b=5;
c=1;
d=0;
e= -1;
unsigned char t=64,z=65,v, x=255;
//Uvjetni operator
u=a==2*b ? c:d;
cout<<"Rezultat uvjetnog operatora je "<<u<<"\n";

//Logicki operatori!
```

```
19 if (a>0 && b >0)
20 cout<<"a i b su pozitivni brojevi!!!\n";
21
22
23 else
24
25
26 cout<<"a i b nisu pozitivni brojevi!!!\n";
27
28
29
30 if (e<0 && a>0 && b >0)
31 cout<<"e, a i b nisu pozitvni brojevi!!!\n";
32 if ((c>d) || (a<b))
33 cout<<"c>d ili a<b je tocno!!!\n";
34 if (!(a==b))
35 cout<<"Rezultat je tocno!!!\n";
36 //Bit operatori!
37 f=b & c;
38 cout<<"Rezultat je: "<<f<<"\n";
39 f=a & b;
40 cout<<"Rezultat je: "<<f<<"\n";
41 f=a | b; //Alt-124 |
42 cout<<"Rezultat je: "<<f<<"\n";
43 h=(~g); //Alt-126 ~
44 cout<<"Rezultat je: "<<h<<"\n";
45 cout<<"Rezultat je: "<<(k^j)<<"\n"; //Alt-94 ^
46 k=8;
47 cout<<"Rezultat pomaka za 3 bita lijevo je: "<<(k<<3)<<"\n";
48 j=8;
49
50
51 cout<<"Rezultat pomaka za 3 bita desno je: "<<(k>>3)<<"\n";
52 //Rad s karakterom!
53 cout<<"Varijabla t sadrzi sljedecu vrijednost: "<<t<<"\n";
54 cout<<"Varijabla z sadrzi sljedecu vrijednost: "<<z<<"\n";
55 v=z+32;
56 cout<<"Rezultat je: "<<v<<"\n";
57 cout<<"Rezultat je: "<<z+32<<"\n";
58 //Maksimalni iznos podatkovnog tipa short!
59 cout<<"Varijabla n sadrzi sljedecu vrijednost: "<<n<<"\n";
60 n=m+1;
61 cout<<"Rezultat je: "<<n<<"\n";
62 return 0;
63 }
```

- Koja vrijednost će se ispisati nakon sljedećih linija koda u=a==2*b ? c:d; cout«”Rezultat uvjetnog operatora je”«u«”\n”;
- Koja vrijednost će se ispisati nakon sljedećih linija koda f=b & c; cout«”Rezultat je: ”«f«”\n”;
- Koja vrijednost će se ispisati nakon sljedećih linija koda f=a & b; cout«”Rezultat je: ”«f«”\n”;

- Koja vrijednost će se ispisati nakon sljedećih linija koda f=a | b; //Alt-124 | cout«”Rezultat je: ”«f«”\n”;
- Koja vrijednost će se ispisati nakon sljedećih linija koda h=(~g); //Alt-126 ~ cout«”Rezultat je: ”«h«”\n”;
- Koja vrijednost će se ispisati nakon sljedeće linije koda cout«”Rezultat je: ”«(k^j)«”\n”; //Alt-94 ^
- Koja vrijednost će se ispisati nakon sljedećih linija koda k=8; cout«”Rezultat pomaka za 3 bita lijevo je: ”«(k«3)«”\n”;
- Koja vrijednost će se ispisati nakon sljedeće linije koda cout«”Rezultat pomaka za 3 bita desno je: ”«(k»3)«”\n”;
- Koja vrijednost će se ispisati nakon sljedećih linija koda cout«”Varijabla t sadrzi sljedecu vrijednost: ”«t«”\n”;
- Koja vrijednost će se ispisati nakon sljedećih linija koda cout«”Varijabla z sadrzi sljedecu vrijednost: ”«z«”\n”;
- Koja vrijednost će se ispisati nakon sljedećih linija koda v=z+32; cout«”Rezultat je: ”«v«”\n”;
- Koja vrijednost će se ispisati nakon sljedećih linija koda cout«”Rezultat je: ”«z+32«”\n”;
- Koja vrijednost će se ispisati nakon sljedećih linija koda cout«”Varijabla n sadrzi sljedecu vrijednost: ”«n«”\n”;
- Koja vrijednost će se ispisati nakon sljedećih linija koda n=m+1; cout«”Rezultat je: ”«n«”\n”;

Ponuđeni odgovori: 1; 0; 15; 65525; 10; 64; @; A; a; 97; -32768.

6. Analiziraj slijedeći program i odgovori na pitanja.

```
#include<iostream>
using namesiace std;
int main(){
int k, n=2, m=3;
k=n*n;
cout<<"n="<<n<<endl;


cout<<"m="<<m<<endl;


cout<<"k="<<k<<"\n";
k=k+m*m*m;
cout<<"k="<<k<<"\n";
return (0);
}
```

- Koje vrijednosti će se ispisati nakon sljedeće linije koda int k, n=2, m=3; k=n*n; cout«”n=”«n«endl;

- Koja vrijednost će se ispisati nakon sljedeće linije koda cout«”m=”«m«endl;
- Koja vrijednost će se ispisati nakon sljedeće linije koda cout«”k=”«k«”\n”;
- Koja vrijednost će se ispisati nakon sljedećih linija koda k=k+m*m*m; cout«”k=”«k«”\n”;

Ponuđeni odgovori: 2; 3; 4; 31.

7. Kompajliraj slijedeći program. Koliko ima grešaka u danom programu?

```
#include<iostream>
using namespace std;
int main(){
int a, b=2 c=3;
a=b*c
cout<<"b="<<b<<","<<"c=<<c<<endl;
cout<<"a="<<a<<"\n;
return (0);
}
```

Arduino

1. U Arduino IDE napiši program koji ispisuje na zaslon vaše podatke:
   Ime: ime studenta
   Prezime: prezime studenta
   STUDIJ: ELEKTROTEHNIKA
   SKOLSKA GODINA: 2017/2018
   Dan, mjesec, godina rodjenja: 01.01.1901
   Broj polozenih ispita: 1 od 5

2. Preuredi kod iz zadatka 1 tako da se program ponaša na sljedeći način:
   a) ime i prezime unosi zasebno ali neka se ispisuje u istom redu.
   b) ispisuje sve navedeno u razmaku od 5 sekundi.
   c) između svakog ispisa postoji prazna linija.

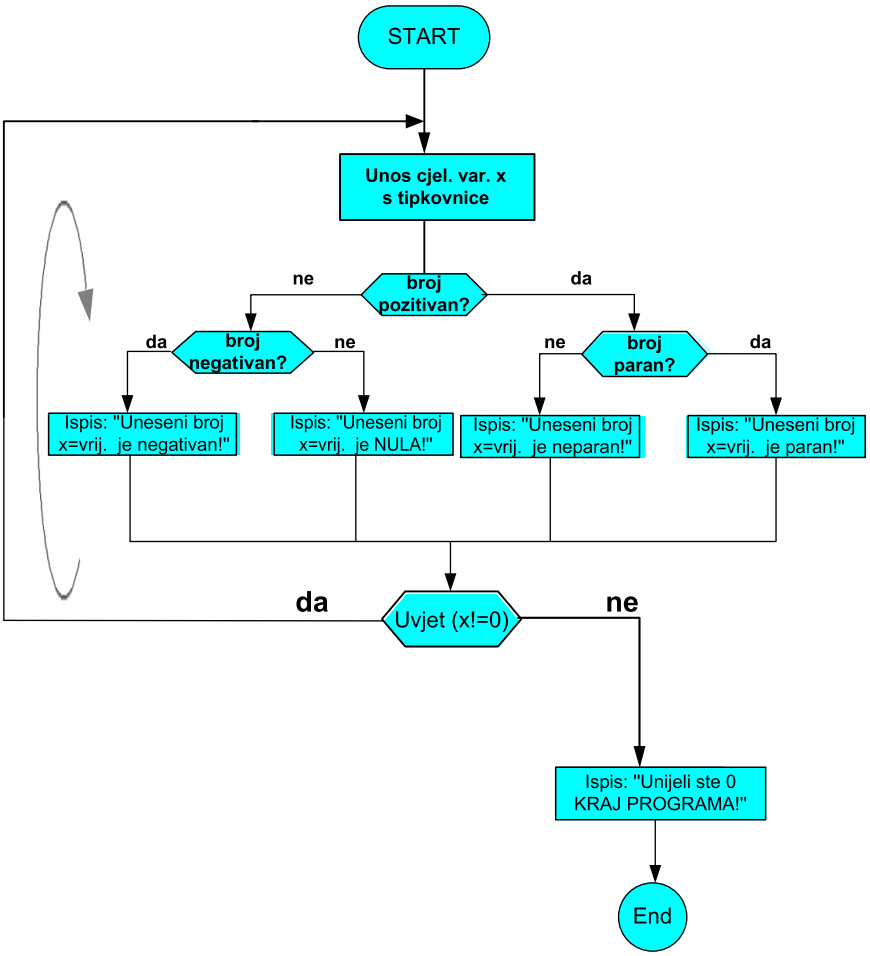
3. Napisati sljedeći program u Arduino IDE:
   a) Deklarirati dvije cjelobrojne varijable a i b (varijabli b pridruži vrijednost 3 u trenutku deklaracije).
   b) Koristiti Serijski Monitor za unos vrijednost varijable a (unijeti vrijednost = 2). Za unos koristiti if naredbu, te Serial.available(), i Serial.read() funkcije.
   c) Izračunati a+b, a-b, a*b, a/b te ispisati na Serijskom monitoru.
   d) Izračunati potenciju a sa b koristeći pow(x,y) funkciju te ispisati na Serijskom monitoru.
   e) Izračunati drugi korijen od b koristeći sqrt(x) funkciju te ispisat na Serijskom monitoru.
   f) Sve ispisivati iz void loop() funkcije te napraviti razmak između ispisa od jednog praznog reda i vremenski od 5 sekundi.

Bilješke uz vježbu 2

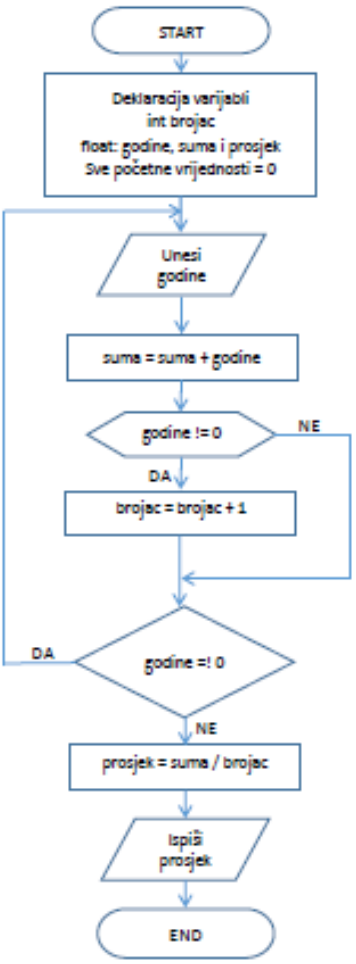
A.3 Vježba 3

C++

1. Koristeći Microsof Visual C++ napiši program koji je zadan sljedećim dijagramom toka. Pri tome koristiti if petlju te modulo operator %, u svrhu prepoznavanja parnih i neparnih brojeva (`a%2 == 0`).



2. Napisati program zadan dijagramom toka na slici




```
Unesi godine: 19.5
Unesi godine: 20
Unesi godine: 21.3
Unesi godine: 22.4
Unesi godine: 0

Prosjecan broj godina je: 20.8
```

3. Isti program iz zadatka 2 napisati primjenom while petlje.

Arduino

1. Preraditi sljedeći „Blink“ program

```
int ledPin = 13;
void setup()
{
pinMode( ledPin, OUTPUT);
}
void loop ()
{
digitalWrite (ledPin, HIGH);
delay (500);
```

```
10 digitalWrite (ledPin, LOW);
11 delay(500);
12 }
```

na način da se treperenje polako usporava do vrijednosti od 3000 milisekundi te potom vraća na početnu vrijednost od 100 milisekundi uz korištenje if petlje. U tu svrhu kreirati varijablu delayPeriod kojoj ćete pridružiti vrijednost 100 te je postepeno uvećavati za 100. Nakon što dosegne vrijednost
3000 postupak vratiti na početak.

2. Preraditi „Blink“ program iz prethodnog zadatka uz korištenje for petlje na način da imamo pet treptaja u trajanju od 500 milisekundi te nakon toga pauzu od 2000 milisekundi.
3. Preraditi „Blink“ program uz korištenje while petlje na način da imamo deset treptaja u trajanju od 200 milisekundi te nakon toga pauzu od 1000 milisekundi.
4. Primjenom switch napiši program koji se ponaša na sljedeći način:

   1. Ako korisnik unese 1 na tipkovnici neka se upali LEDica i ispiše na Serial monitoru:

   LED is ON

   2. Ako korisnik unese 2 na tipkovnici neka se ugasi LEDica i ispiše na Serial monitoru:

   LED is OFF

   3. Ako korisnik unese 3 na tipkovnici neka se upali LEDica i ispiše na Serial monitoru:

   ——- MENU ——-

   1. Switch LED on.
   2. Switch LED off.
   3. This menu.

   ————————

   4. U slučaju unosa bilo kojih drugih vrijednosti neka se ispiše na Serial monitoru:

   ”Invalid option”

Bilješke uz vježbu 3

A.4 Vježba 4

C++

1. Napisati C++ program koji računa sumu kvadrata brojeva od 1 do 100. Za izračun i ispis sume kvadrata napisati funkciju sumaKvadrata te istu pozvati u glavnom programu. Za izračun sume kvadrata u funkciji sumaKvadrata koristiti for petlju.
2. Preraditi program iz 1. zadatka na način da se u glavnom programu unosi početni i krajnji broj za koje će se računati suma kvadrata, a funkcijskim pozivom prema vrijednosti proslijediti funkciji upisane parametre. Korištenjem while petlje u glavnom programu omogućiti unos početnog, krajnjeg broja i izračun sume kvadrata sve dok se kao vrijednost početnog broja ne unese vrijednost 0.

Arduino

1. Preraditi sljedeći program

```
void setup() {
Serial.begin(9600);
Serial.println("-------------------------");
Serial.println("| Program Options Menu |");


Serial.println("-------------------------");
}
void loop() {
}
```

   na način da za ispis znakova ”—————————” na Serial monitoru koristite funkciju dashed_Line koja se poziva s parametrom koji označava broj znakova ”-” koji će se ispisati.

2. Preraditi sljedeći program

```
int ledPin = 13;



void setup()
{
pinMode( ledPin, OUTPUT);
}



void loop ()
{
digitalWrite (ledPin, HIGH);
delay (500);
```

```
16 digitalWrite (ledPin, LOW);
17 delay(500);
18 }
```

na način da se definira funkcija flash koja nema argumenata. Funkcija flash nakon svog pozivanja treba izazvati treptanje diode tako da razmaci između svijetljenja i ugašenosti LED diode iznose 100 milisekundi. U tu svrhu deklarirati i inicijalizirati globalnu varijablu delayPeriod.
Treperenje se treba ponoviti 10 puta s razmakom od 2 sekunde za što treba upotrijebiti for petlju.

3. Preraditi program iz prethodnog zadatka na način da se funkcija flash poziva s parametrima numFlashes - broj treptaja diode i d - dužina trajanja treptaja. Nadopuniti funkciju na način da se na Serial monitoru ispisuje kada je LED dioda upaljena / ugašena. (Led dioda je upaljena / Led dioda je ugašena).

Bilješke uz vježbu 4

A.5 Vježba 5

C++

1. Napiši program koji postavlja elemente dvodimenzionalnog polja na sljedeći način:

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15

   Popunjavanje elemenata polja napravite funkcijskim pozivom (naziv funkcije setVal)

   Ispišite elemente polja korištenjem funkcijskog poziva (naziv funkcije printVal)

2. Napiši program koji postavlja elemente dvodimenzionalnog polja na sljedeći način:

11	12	13	14	15
21	22	23	24	25
31	32	33	34	35

   Popunjavanje elemenata polja napravite funkcijskim pozivom (naziv funkcije postavi)

   Ispišite elemente polja korištenjem funkcijskog poziva (naziv funkcije ispis)

3. Napiši program koji postavlja elemente dvodimenzionalnog polja na sljedeći način:

1	2	3
2	4	6
3	6	9
4	8	12
5	10	15

   Popunjavanje elemenata polja napravite funkcijskim pozivom (naziv funkcije SETVal)

   Ispišite elemente polja korištenjem funkcijskog poziva (naziv funkcije PrintVAL)

4. Napiši program koji zbraja dvije matrice. Unos matrica ponavljati sve dok se za dimenzije matrica koje se zbrajaju (broj redaka i stupaca) različite od nula. Za upisane vrijednosti 0 redaka ili 0 stupaca izaći iz petlje i završiti s programom. Ukoliko su dimenzije različite od 0 ispisati zbroj unesenih matrica i ponoviti unos.

5. Napiši program koji oduzima dvije matrice. Unos matrica ponavljati sve dok se za dimenzije matrica koje se oduzimaju (broj redaka i stupaca) različite od nula. Za upisane vrijednosti 0 redaka ili 0 stupaca izaći iz petlje i završiti s programom. Ukoliko su dimenzije različite od 0, ispisati razliku unesenih matrica i ponoviti unos.

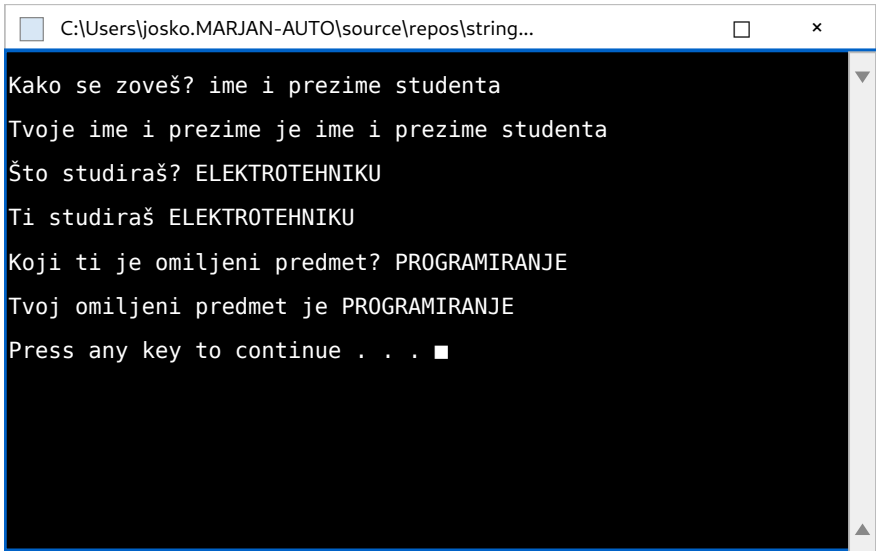
6. Napiši program koji sve elemente matrice množi s upisanim brojem. Unos matrice ponavljati dok su dimenzije matrice koju množite (broj redaka i stupaca) različite od nula ili dok je broj s kojim množite elemente matrice različit od nula. Za upisane vrijednosti 0 redaka, 0 stupaca ili 0 množitelj matrice, izaći iz petlje i završiti s programom. Ukoliko su dimenzije različite od 0, ispisati elemente matrice pomnožene s unesenim brojem.

Bilješke uz vježbu 5

A.6 Vježba 6

C++

1. Napisati funkciju koja kopira sadržaj stringa2 u string1. Izvršiti funkcijski poziv za slučaj kada je sadržaj stringa2 jednak: SVEUCILISNI ODJEL ZA STRUCNE STUDIJE!. Vidi primjer u skripti.
2. Napisati program koji se ponaša na sljedeći način. Za unos imena i prezimena, studija i omiljenog predmeta obavezno upotrijebiti cin.getline metodu.

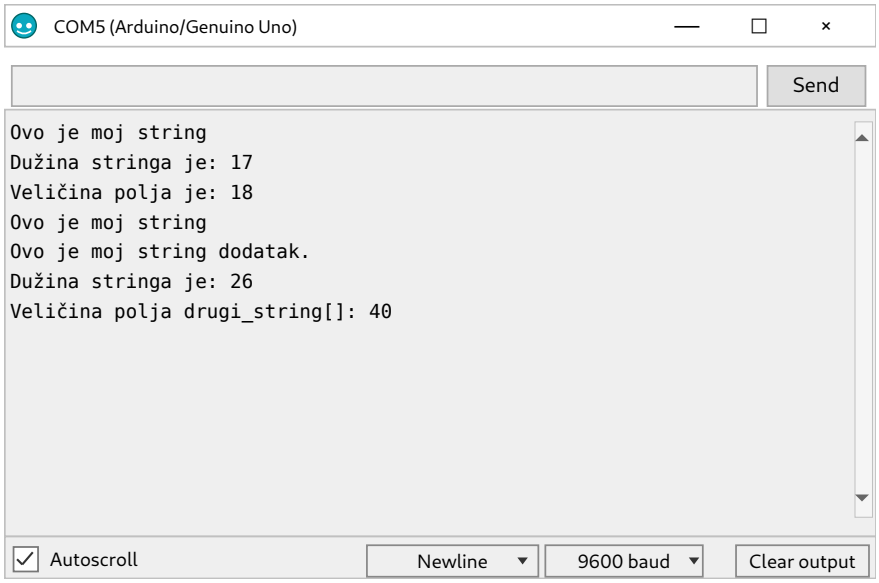


Arduino

1. Napiši Arduino program u kojem ćeš definirati string, imena string te mu pridružiti: “Ovo je moj string“. Zatim deklariraj drugi string naziva drugi_string[40], te jednu cjelobrojnu varijablu int a = 0;. Neka program obavi sljedećih sedam operacija:
   1. Ispiši string string
   ```
   Serial.println(string);
   ```
   2. Izmjeri veličinu stringa (isključuje zadnji znak nule)
   ```
   a = strlen(string);
   ```
   3. Izmjeri veličinu cijelog polja stringova (uključujući znak nule)
   ```
   a = sizeof(string);
   ```
   4. Kopiraj string, string, u drugi string, drugi_string
   ```
   strcpy(drugi_string, string);
   ```
   5. Na kraju teksta od drugi_string dodaj dodatak teksta “dodatak“.
   ```
   strcat(drugi_string, " dodatak.");
   ```
   6. Sad opet izmjeri veličinu stringa drugi_string (isključuje zadnji znak nule)
   ```
   a = strlen(drugi_string);
   ```
   7. Sad opet izmjeri veličinu cijelog stringa, drugi_string s korištenim i nekorištenim poljima

```
a = sizeof(drugi_string);
```

Neka ispis na Serijskom monitoru izgleda ovako:



Neka se sve ovo ispiše samo jednom.

Bilješke uz vježbu 6

A.7 Vježba 7

C++

1. Napiši C++ program koji se ponaša na sljedeći način. Deklariraj sljedeće:

```
int vrijednost1 = 5, vrijednost2 = 15;
int *p1, *p2;
```

Pokazivaču p1 pridruži adresu varijable vrijednost1
Pokazivaču p2 pridruži adresu varijable vrijednost2
Vrijednosti na koju pokazuje p1 pridruži 10
Vrijednosti na koju pokazuje p2 pridruži vrijednost na koju pokazuje p1
Pokazivaču p1 pridruži pokazivač p2
Vrijednost na koju pokazuje p1 postavi na 20
Koje vrijednosti sadrže varijable vrijednost1 i vrijednost2?
Ispiši te vrijednosti kao i vrijednosti p1 i p2!

2. Postoji mnogo poznatih metoda sortiranja (bubble sort, quick sort, shell sort i dr.) od kojih svaka ima svoje prednosti, mane i područje primjene gdje postiže najveću učinkovitost. Svakoj metodi zajedničko je da u skladu s traženim poretkom, zamjenjuje elemente polja, čime se mijenja položaj pojedinih elemenata u polju. Sljedećim pridruživanjem ne mogu se zamijeniti dva elementa:

```
elem1=elem2;
elem2=elem1;
```

ili

```
polje[i]=polje[j];
polje[j]=polje[i];
```

jer će se vrijednost elem1 ili polje[i] izgubiti i oba elementa će sadržavati istu vrijednost.
Mora se koristiti pomoćna varijabla:

```
temp=elem1;
elem1=elem2;
elem2=temp;
```

U sljedećem primjeru prikazat ćemo bubble sort metodu sortiranja. Bubble sort je najjednostavnija metoda sortiranja koju je najlakše shvatiti, a sastoji se od: prolaza kroz polje i uspoređivanja parova brojeva, da bi se utvrdila potreba za njihovom razmjenom. Za cjelovito sortiranje elemenata polja potrebno je napraviti nekoliko prolaza kroz polje, što doprinosi jednostavnosti ove metode. Druge metode koriste elemente bubble sort, ali izbjegavaju velik broj prolaza kroz polje, te time brže sortiraju. Potrebno je u ovom zadataku napisati definiciju funkcije ispisi_polje koja nedostaje. Znači, kada riješite zadatak, šaljete potpuni programski kod zadatka, a ne samo definiciju funkcije ispisi_polje.

```cpp
// sortiranje bubble sort
#include <iostream>
using namespace std;
#define MAX 20
void puni_polje(int a[])
{
int i;
srand(1000);
for (i = 0; i<MAX; i++)
a[i] = rand() % 100;
}
void ispisi_polje(int a[])
//{


// ovdje je potrebno napisati linije koda koje ce ispisivati polje


//

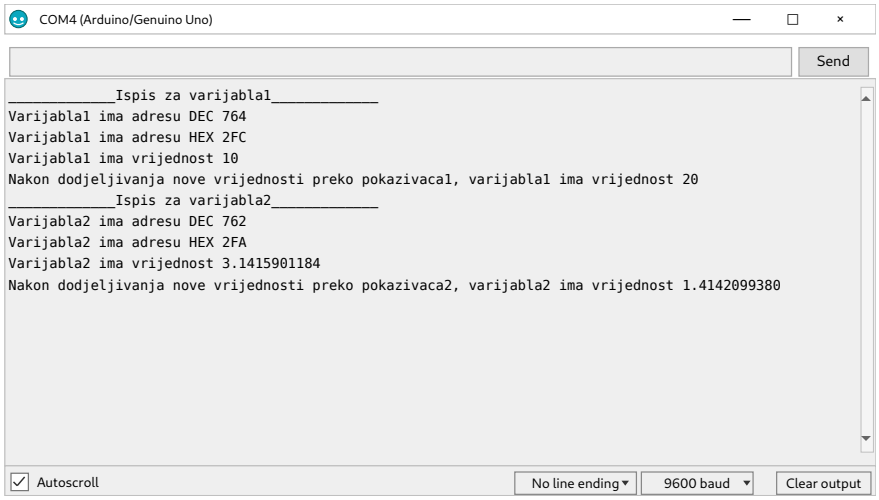

}
void sort_polje(int a[])
{
int temp, i, j;
for (i = 0; i<MAX - 1; i++)
for (j = i + 1; j<MAX; j++)
if (a[i]>a[j])
{
temp = a[i];
a[i] = a[j];
a[j] = temp;
}
}
int main() {
int a[MAX];
//Ispunjava polje slucajnim brojevima od 0 do 100
puni_polje(a);
cout << "Ispis - nesortirano polje: \n";
ispisi_polje(a);
sort_polje(a); //Sortiraj elemente polja
cout << "Ispis - sortirano polje: \n";
ispisi_polje(a);
system("pause");
return 0;
}
```

Arduino

1. U sljedećem primjeru nadopuniti ispis za varijablu2 i varijablu3 po uzoru na ispis za varijablu1. Na istome mjestu preko pokazivača promijeniti vrijednosti varijabla2 u 1.41421, a varijabla3 u "Ovo je POTPUNO NOVO polje znakova, STRING"; te potom napraviti novi ispis. Poslati potpuni kod.

```
int varijabla1 = 10;
float varijabla2 = 3.14159;
int *pokazivac1;
float *pokazivac2;
void setup() {
Serial.begin(9600);
Serial.println("_____________Ispis za varijabla1_____________________________");
pokazivac1 = &varijabla1;
Serial.print("Varijabla1 ima adresu DEC ");
Serial.println( (long) &pokazivac1, DEC);
Serial.print("Varijabla1 ima adresu HEX ");
Serial.println( (long) &pokazivac1, HEX);
Serial.print("Varijabla1 ima vrijednost ");
Serial.println( (long) *pokazivac1, DEC);
*pokazivac1 =20;
Serial.print("Nakon dodjeljivanja nove vrijednosti preko pokazivaca1, varijabla1
    ima vrijednost ");
Serial.println("_____________Ispis za varijabla2_____________________________");
Serial.println("_____________Ispis za varijabla3_____________________________");
}
void loop() {
}
```

Ispis bi trebao izgledati ovako:



2. Dovrši sljedeći kod na način da unutar loop () petlje, koristiš pokazivač p1 kojemu ćeš pridružiti adresu varijable kraj, te promijeniti njezinu vrijednost preko pokazivača, u vrijednost 10, te

pozivom funkcije funkcija_blink() pokrenuti desetorostruke serije treperenja. Između svake serije se na serijskom monitoru ispiše ovo je kraj serije.

```
int ledPin =13;
int i = 0;
int kraj = 0;
int *p1;
int brojilo = 0;
int funkcija_blink()
{
for(int i; i<kraj; i++)
{
digitalWrite(ledPin, HIGH);
delay(300);
digitalWrite(ledPin, LOW);
delay(300);
brojilo = brojilo +1;
Serial.print( "Ovo je ");
Serial.print(brojilo);
Serial.println(" treptaj");
}
Serial.print( "");
Serial.println("Ovo je kraj serije");
Serial.print( "");
}


void setup() {
Serial.begin(9600);
pinMode(ledPin,OUTPUT);
}
void loop() {


// OVDJE JE POTREBNO DOPUNITI PROGRAMSKI KOD
}
```

Bilješke uz vježbu 7

A.8 Vježba 8

C++

1. Napiši C++ program koji se sastoji od sljedećeg (vidi predavanja Strukture):
   Polja strukture podataka student_t dužine 3 elementa sa sljedećim podacima:

   Godina_rodjenja int

   Prezime_ime dužine 50 znakova

   Studij dužine 50 znakova
   Na osnovu strukture student_t deklarirati objekt student.

   U glavnom programu napraviti unos podataka za elemente strukture korištenjem cin.getline(). Po izvršenom unosu ispisati podatke pozivom funkcije ispis().

   Funkciju ispis() deklarirati prije funkcije main (prototype il prauzorak), a definirati nakon funkcije main().

2. Koristiti kod iz prethodnog zadatka ali ga proširiti na način da se upisuje broj studenata za koje će se vršiti unos i dinamički alocirati potrebna količina memorije za objekt student korištenjem funkcije new (pogledati predavanja dinamička memorija) Nakon unosa broja studenata, unos podataka u strukturu realizirati kroz while petlju.

Arduino

1. Kreirati sljedeću strukturu podataka:

```
struct sensorinfo {
char date[9];
int indoortemp;
int outdoortemp;
} morningTemps;
```

   Na osnovu strukture podataka napisati kod tako ta se senzorski podaci očitavaju svakih 30 sekundi i ispisuju na zaslon kao:
   Danas je 09/05/18
   Jutarnja temperatura u kuci je: 25
   Jutarnja temperatura vanka je: 17
   Danas je 09/05/18
   Jutarnja temperatura u kuci je: 25
   Jutarnja temperatura vanka je: 17

Bilješke uz vježbu 8

A.9 Vježba 9

C++

1. Napiši C++ program u kojem ćete deklarirati klasu COtpornik koja ima private članove R1 i R2. Također klasa treba sadržavati dva funkcijska člana za računanje serijskoga i paralelnog spoja ova dva otpornika. U glavnom programu izvesti objekt iz klase COtpornik i pozvati funkcije za izračun i ispisati vrijednosti serijskog i paralelnog otpora za vrijednosti 3 i 4 OHMA.
2. Napiši C++ program u kojem ćete deklarirati klasu CPravokutnik. Unutar klase definirati funkcijski član koji računa opseg pravokutnika i funkcijski član za unos stranica pravokutnika. Korištenjem pokazivača na izvedeni objekt klase CPravokutnik u glavnom programu postaviti stranice i ispisati vrijednost opsega za pravokutnik sa stranicama 5 i 10 cm.

Arduino

1. Arduino: Riješiti zadatak 1. iz materijala za pripremu vježbe.

Bilješke uz vježbu 9

A.10 Vježba 10

C++

1. Napraviti mapu TEST na lokalnom disku i program u C++ koji će u mapi TEST kreirati datoteku naziva vjezba.txt s vašim podacima: ime i prezime, datum rođenja i naziv studija.
2. Napraviti program koji će gore kreiranu datoteku vjezba.txt otvoriti i njen sadržaj ispisati na ekran.

Arduino

1. Ispitaj sadržaj cijele EEPROM memorije. Za lokaciju EEPROM memorije koja je različita od 0 ispisati sadržaj i adresu na serial monitoru.

Bilješke uz vježbu 10

B. Priprema za kolokvij/ispit

Pitanja u ovome poglavlju služe ponavljanju temeljnih pojmova i provjeri razumijevanja prije kolokvija ili ispita. Sastavljena su kao nova pitanja na temelju nastavnih cjelina obuhvaćenih kolegijem. Ne ponavljaju zadatke iz laboratorijskih vježbi i ne zahtijevaju pisanje cjelovitoga programskog koda.

Kod pitanja s oznakama *Točno* i *Netočno* treba odabrati jednu od dviju mogućnosti. Kod ostalih pitanja ponuđena su četiri odgovora, od kojih je samo jedan točan. Rješenja nisu navedena u studentskome izdanju; priložena su u zasebnoj datoteci `Kljuc_odgovora_za_nastavnika.pdf`.

B.1 Brojevni sustavi, ASCII i osnove C++ programa

1. Binarni brojevni sustav koristi samo znamenke 0 i 1.
   a) Točno
   b) Netočno
2. Koja je dekadska vrijednost binarnoga broja 101101_2?
   a) 41
   b) 43
   c) 45
   d) 53
3. Koji je heksadekadski zapis dekadskoga broja 46?
   a) $2A_{16}$
   b) $2E_{16}$
   c) $3C_{16}$
   d) $4E_{16}$
4. Koji je oktalni zapis dekadskoga broja 25?
   a) 27_8
   b) 30_8
   c) 31_8
   d) 35_8
5. U osmerobitnom zapisu 10010110_2 najznačajniji bit (MSB) jednak je 1.
   a) Točno
   b) Netočno
6. Koja je dekadska vrijednost heksadekadske znamenke `D`?
   a) 11
   b) 12
   c) 13
   d) 14

7. Koji zapis sadržava znamenku koja ne pripada zadanomu brojevnom sustavu?

   a) 10102_2
   b) 735_8
   c) 908_{10}
   d) $B7F_{16}$

8. U standardnoj ASCII tablici dekadska vrijednost maloga slova `a` veća je za 32 od vrijednosti velikoga slova `A`.

   a) Točno
   b) Netočno

9. Koja je dekadska ASCII vrijednost znaka `'0'`?

   a) 0
   b) 32
   c) 48
   d) 57

10. Kojim znakovima započinje jednoredni komentar u C++-u?

    a) `/*`
    b) `//`
    c) `##`
    d) `--`

11. Svaki C++ program namijenjen izvođenju u hostiranom okruženju mora sadržavati funkciju `main()`.

    a) Točno
    b) Netočno

12. Koji je zapis ispravna naredba za ispis vrijednosti varijable `broj` na standardni izlaz?

    a) `cin >> broj;`
    b) `cout >> broj;`
    c) `cout << broj;`
    d) `cin << broj;`

B.2 Tipovi podataka i operatori

1. Koji se tip podatka u pravilu koristi za pohranu jednoga znaka?

   a) `bool`
   b) `char`
   c) `float`
   d) `void`

2. Ključna riječ `const` označuje da se vrijednost objekta nakon inicijalizacije ne smije mijenjati.
   a) Točno
   b) Netočno
3. Koji je od navedenih identifikatora neispravan u C++-u?
   a) `broj2`
   b) `_broj`
   c) `broj_studenata`
   d) `2broj`
4. Koji operator ispituje jednakost dviju vrijednosti?
   a) `=`
   b) `==`
   c) `!=`
   d) `:=`
5. Izraz `17 % 5` ima vrijednost 2.
   a) Točno
   b) Netočno
6. Koju vrijednost dobiva varijabla `x`?

```
int a = 4;
int x = a++;

```

   a) 3
   b) 4
   c) 5
   d) Izraz nije valjan.
7. Koju vrijednost dobiva varijabla `x`?

```
int a = 4;
int x = ++a;

```

   a) 3
   b) 4
   c) 5
   d) Izraz nije valjan.
8. Koji izraz predstavlja logičko *i*?
   a) `&`
   b) `&&`
   c) `|`
   d) `^`

9. Operator ~ jest bitovni operator komplementa.

    a) Točno
    b) Netočno

10. Koja je konačna vrijednost varijable `a`?

```
int a = 18;
a /= 3;
a += 2;

```

    a) 4
    b) 6
    c) 8
    d) 20

11. Koju vrijednost ima izraz `(7 > 3) && (2 == 5)`?

    a) **`true`**
    b) **`false`**
    c) 2
    d) Izraz se ne može izračunati.

12. Koju vrijednost dobiva varijabla `rezultat`?

```
int a = 6;
int rezultat = (a % 2 == 0) ? 10 : 20;

```

    a) 0
    b) 6
    c) 10
    d) 20

13. Lokalna varijabla deklarirana unutar funkcije izravno je vidljiva u svim ostalim funkcijama programa.

    a) Točno
    b) Netočno

14. Rezultat operatora **`sizeof`** izražen je u bajtovima.

    a) Točno
    b) Netočno

15. Koja tvrdnja o tipu **`unsigned int`** je točna?

    a) Namijenjen je samo negativnim cijelim brojevima.
    b) Ne može pohraniti vrijednost nula.
    c) Namijenjen je nenegativnim cijelim vrijednostima.
    d) Uvijek zauzima jedan bajt.

B.3 Kontrola toka programa

1. Koja se naredba koristi za odabir jednoga od više slučajeva prema vrijednosti cjelobrojnoga ili znakovnoga izraza?

   a) `while`
   b) `switch`
   c) `continue`
   d) `goto`

2. Tijelo petlje `do while` izvršava se najmanje jedanput.

   a) Točno
   b) Netočno

3. Koliko se puta izvršava tijelo petlje?

```
for (int i = 0; i < 5; ++i) {
    cout << i;
}

```

   a) 4
   b) 5
   c) 6
   d) Beskonačno mnogo puta.

4. Petlja `while` najprije izvršava tijelo, a tek zatim provjerava uvjet.

   a) Točno
   b) Netočno

5. Koja je konačna vrijednost varijable `b`?

```
int a = 4;
int b = 7;
if (a > b)
    ++b;
else
    --b;

```

   a) 4
   b) 6
   c) 7
   d) 8

6. Naredba `break` prekida izvođenje najbliže petlje ili naredbe `switch`.

   a) Točno
   b) Netočno

7. Što u petlji čini naredba **`continue`**?

   a) Završava cijeli program.
   b) Trajno izlazi iz petlje.
   c) Preskače ostatak te iteracije i prelazi na sljedeću provjeru.
   d) Vraća izvođenje na početak funkcije `main()`.

8. Koliko se puta izvršava tijelo sljedeće petlje?

```
int i = 3;
while (i < 3) {
    --i;
}

```

   a) Nijednom.
   b) Jedanput.
   c) Triput.
   d) Beskonačno mnogo puta.

9. Ako u jednoj grani naredbe **`switch`** izostavimo **`break`**, izvođenje se može nastaviti u sljedećoj grani.

   a) Točno
   b) Netočno

10. Koja se vrijednost ispisuje?

```
int a = 0;
do {
    a += 2;
} while (a < 5);
cout << a;

```

   a) 2
   b) 4
   c) 5
   d) 6

11. U zaglavlju petlje **`for`** drugi izraz predstavlja uvjet nastavka izvođenja petlje.

   a) Točno
   b) Netočno

12. Koja se vrijednost ispisuje?

```
int zbroj = 0;
for (int i = 1; i <= 4; ++i) {
    if (i == 3)
        continue;
    zbroj += i;
```

```
}
cout << zbroj;

```

a) 4
b) 6
c) 7
d) 10

13. Koja je konačna vrijednost varijable x?

```
int x = 1;
for (int i = 0; i < 4; ++i) {
    if (i == 2)
        break;
    x *= 3;
}

```

a) 3
b) 6
c) 9
d) 27

14. Grana **default** u naredbi **switch** obvezna je u svakom programu.

a) Točno
b) Netočno

15. Koja tvrdnja najbolje opisuje ugniježđenu petlju?

a) To je petlja čije se tijelo nalazi unutar tijela druge petlje.
b) To je petlja bez uvjeta.
c) To je petlja koja se mora izvesti točno jedanput.
d) To je drugi naziv za rekurzivnu funkciju.

B.4 Funkcije

1. Što određuje tip naveden ispred naziva funkcije?

a) Broj parametara funkcije.
b) Tip vrijednosti koju funkcija vraća.
c) Doseg lokalnih varijabli.
d) Redoslijed pozivanja funkcija.

2. Funkcija tipa **void** ne vraća vrijednost pozivatelju.

a) Točno
b) Netočno

3. Koji je od navedenih zapisa prototip funkcije koja prima dva cijela broja i vraća cijeli broj?

   a) `int zbroj(int, int);`
   b) `void zbroj(int, int)`
   c) `zbroj(int, int): int;`
   d) `int zbroj = (int, int);`

4. Pri prijenosu argumenta po vrijednosti promjena odgovarajućega parametra u funkciji mijenja izvornu varijablu pozivatelja.

   a) Točno
   b) Netočno

5. Što u parametru `int& broj` označuje znak &?

   a) Parametar je pokazivač.
   b) Parametar je referenca na izvorni objekt.
   c) Parametar je konstanta.
   d) Parametar nema tip.

6. Koje su vrijednosti varijabli `x` i `y` nakon poziva?

```
void promijeni(int a, int& b) {
    a += 5;
    b += 5;
}

int x = 2;
int y = 3;
promijeni(x, y);

```

   a) `x = 2, y = 3`
   b) `x = 2, y = 8`
   c) `x = 7, y = 3`
   d) `x = 7, y = 8`

7. Preopterećene funkcije imaju isti naziv, ali se moraju razlikovati u broju ili tipovima parametara.

   a) Točno
   b) Netočno

8. Dvije funkcije mogu se preopteretiti samo promjenom tipa povratne vrijednosti, uz potpuno jednak popis parametara.

   a) Točno
   b) Netočno

9. Koja je vrijednost izraza `umnozak(3)`?

```
int umnozak(int a, int b = 4) {
    return a * b;
```

```
}

```

a) 3
b) 4
c) 7
d) 12

10. Ako jedan parametar ima zadani argument, svi parametri desno od njega također moraju imati zadane argumente.

a) Točno
b) Netočno

11. Što je rekurzija?

a) Pozivanje funkcije prije njezine deklaracije.
b) Svojstvo funkcije da izravno ili neizravno poziva samu sebe.
c) Višestruko preopterećenje istoga operatora.
d) Prijenos svih argumenata po referenci.

12. Rekurzivna funkcija treba imati uvjet završetka kako pozivi ne bi trajali neograničeno.

a) Točno
b) Netočno

13. Koju vrijednost vraća funkcija `izracunaj(2, 5)`?

```
int izracunaj(int a, int b) {
    return a * (b + 1);
}

```

a) 7
b) 10
c) 11
d) 12

14. Lokalna varijabla funkcije u pravilu prestaje postojati kada završi izvođenje te funkcije.

a) Točno
b) Netočno

15. Koja deklaracija kombinira prijenos prvoga argumenta po vrijednosti, a drugoga po referenci?

a) `void obradi(int& a, int b);`
b) `void obradi(int a, int& b);`
c) `void obradi(int* a, int* b);`
d) `void obradi(int a, int b);`

B.5 Polja i C-znakovni nizovi

1. Koji je najveći valjani indeks polja deklariranoga kao `int podaci[8];`?

   a) 6
   b) 7
   c) 8
   d) 9

2. Svi elementi jednoga običnog polja moraju biti istoga tipa.

   a) Točno
   b) Netočno

3. Koji zapis pristupa trećemu elementu polja `rezultati`?

   a) `rezultati[1]`
   b) `rezultati[2]`
   c) `rezultati[3]`
   d) `rezultati(3)`

4. Koju vrijednost ima element `a[3]`?

```
int a[] = {4, 8, 12, 16, 20};

```

   a) 8
   b) 12
   c) 16
   d) 20

5. Pristup elementu izvan granica običnoga polja automatski izaziva standardnu C++ iznimku koju je uvijek moguće uhvatiti.

   a) Točno
   b) Netočno

6. Koji zapis pristupa elementu u drugome retku i četvrtome stupcu dvodimenzionalnoga polja `matrica`?

   a) `matrica[2][4]`
   b) `matrica[1][3]`
   c) `matrica[2, 4]`
   d) `matrica(1)(3)`

7. Koliko elemenata sadržava polje deklarirano kao `double mjerenja[3][5];`?

   a) 8
   b) 10
   c) 15
   d) 25

8. Pri predaji običnoga polja funkciji parametar polja prilagođava se pokazivaču na njegov prvi element.

    a) Točno
    b) Netočno

9. Zašto se funkciji koja obrađuje polje često zasebno predaje broj elemenata?

    a) Zato što funkcija iz prilagođenoga parametra polja u pravilu ne može zaključiti broj elemenata izvornoga polja.
    b) Zato što svaki element polja ima drukčiji tip.
    c) Zato što indeksiranje počinje brojem 1.
    d) Zato što se polje ne može čitati u funkciji.

10. C-znakovni niz završava znakom `'\0'`.

    a) Točno
    b) Netočno

11. Koliko elemenata najmanje mora imati polje znakova za pohranu teksta `"Split"` kao C-znakovnoga niza?

    a) 4
    b) 5
    c) 6
    d) 7

12. Koja funkcija iz zaglavlja `<cstring>` kopira C-znakovni niz iz izvora u odredište?

    a) `strcat()`
    b) `strcmp()`
    c) `strcpy()`
    d) `strlen()`

13. Ako `strcmp(a, b)` vraća 0, nizovi `a` i `b` imaju jednak sadržaj.

    a) Točno
    b) Netočno

14. Koja funkcija dodaje sadržaj jednoga C-znakovnog niza na kraj drugoga?

    a) `strcat()`
    b) `strcpy()`
    c) `strcmp()`
    d) `atoi()`

15. Koja je važna slabost funkcije `atoi()`?

    a) Ne prihvaća niz znakova kao argument.
    b) Ne može pouzdano razlikovati nevaljan zapis broja od valjanoga zapisa nule.
    c) Uvijek vraća vrijednost tipa `double`.
    d) Može pretvarati samo negativne brojeve.

B.6 Klasa `string`

1. Koje je zaglavlje potrebno uključiti za uporabu klase `std::string`?
   a) `<cstring>`
   b) `<string>`
   c) `<vector>`
   d) `<sstream>`
2. Objekt tipa `string` automatski upravlja memorijom potrebnom za svoj sadržaj.
   a) Točno
   b) Netočno
3. Koji funkcijski član vraća broj znakova trenutačno pohranjenih u stringu?
   a) `size()`
   b) `find()`
   c) `clear()`
   d) `c_str()`
4. Koja funkcija učitava cijeli redak, uključujući razmake, u objekt `string tekst`?
   a) `cin >> tekst;`
   b) `getline(cin, tekst);`
   c) `cin.get(tekst);`
   d) `scanf(tekst);`
5. Funkcijski član `at()` provjerava je li zadani indeks unutar granica stringa.
   a) Točno
   b) Netočno
6. Koji je rezultat izraza `string("C++") + "17"`?
   a) `C++`
   b) `17C++`
   c) `C++17`
   d) Izraz nije valjan.
7. Koji funkcijski član traži prvo pojavljivanje podstringa?
   a) `find()`
   b) `substr()`
   c) `replace()`
   d) `compare()`
8. Vrijednost `string::npos` označuje da traženi sadržaj nije pronađen.
   a) Točno
   b) Netočno

9. Čemu služi funkcijski član `c_str()`?

    a) Briše sve znakove stringa.
    b) Vraća pokazivač na nul-terminirani znakovni niz koji predstavlja sadržaj stringa.
    c) Pretvara string u jedan znak.
    d) Vraća broj zauzetih bajtova na *stacku*.

10. Koji je prikladan parametar funkcije koja treba čitati velik objekt `string` bez kopiranja i bez mijenjanja?

    a) `string tekst`
    b) `string* const tekst`
    c) `const string& tekst`
    d) `void tekst`

B.7 Pokazivači

1. Što pokazivač pohranjuje kao svoju vrijednost?

    a) Naziv drugoga objekta.
    b) Adresu objekta ili funkcije odgovarajućega tipa.
    c) Isključivo cijeli broj koji se ispisuje na zaslon.
    d) Tip podatka objekta bez njegove adrese.

2. Koja deklaracija stvara pokazivač `p` na objekt tipa `int`?

    a) `int p;`
    b) `int &p;`
    c) `int *p;`
    d) `pointer<int> p;`

3. Operator &, primijenjen na objekt u izrazu, daje njegovu adresu.

    a) Točno
    b) Netočno

4. Što znači izraz `*p` ako `p` pokazuje na valjani objekt?

    a) Adresu samoga pokazivača.
    b) Objekt na koji `p` pokazuje.
    c) Broj bajtova koje pokazivač zauzima.
    d) Novi pokazivač bez vrijednosti.

5. Koja je konačna vrijednost varijable `a`?

```
int a = 6;
int* p = &a;
*p = 14;

```

a) 6
b) 8
c) 14
d) Adresa varijable `a`.

6. Vrijednost pokazivača `p` koji pokazuje na varijablu `a` jednaka je adresi varijable `a`, a ne njezinu sadržaju.

a) Točno
b) Netočno

7. Koja je preporučena vrijednost pokazivača koji trenutačno ne pokazuje ni na jedan objekt?

a) `0.0`
b) `false`
c) `nullptr`
d) Prazan znakovni niz.

8. Dereferenciranje pokazivača čija je vrijednost `nullptr` ima nedefinirano ponašanje.

a) Točno
b) Netočno

9. Koja se vrijednost ispisuje?

```
int brojevi[] = {3, 6, 9, 12};
int* p = brojevi;
++p;
cout << *p;

```

a) 3
b) 6
c) 9
d) 12

10. Naziv običnoga polja u većini izraza pretvara se u pokazivač na prvi element polja.

a) Točno
b) Netočno

11. Što označuje izraz `p + 2` ako `p` pokazuje na prvi element polja?

a) Adresu drugoga elementa polja.
b) Adresu trećega elementa polja.
c) Vrijednost prvoga elementa uvećanu za 2.
d) Adresu pokazivača `p` uvećanu za dva bajta neovisno o tipu.

12. Pokazivačka aritmetika uzima u obzir veličinu tipa na koji pokazivač pokazuje.

a) Točno
b) Netočno

13. Koji izraz daje adresu trećega elementa polja `a`?

    a) `a + 1`
    b) `a + 2`
    c) `*a + 2`
    d) `&a + 3`

14. Koja je konačna vrijednost varijable `b`?

```
int a = 5;
int b = 8;
int* p = &a;
*p += 3;
p = &b;
*p *= 2;

```

    a) 8
    b) 10
    c) 13
    d) 16

15. Pokazivač na objekt korisnički definiranoga tipa nije dopušten u standardnom C++-u.

    a) Točno
    b) Netočno

B.8 Reference

1. Referenca u C++-u predstavlja drugo ime postojećega objekta.

    a) Točno
    b) Netočno

2. Koja je deklaracija ispravna ako varijabla `broj` već postoji?

    a) `int& r;`
    b) `int& r = broj;`
    c) `int r& = broj;`
    d) `reference<int> r = broj;`

3. Lvalue-referenca mora se inicijalizirati pri deklaraciji.

    a) Točno
    b) Netočno

4. Što se događa u sljedećem kodu?

```
int broj = 5;
int& r = broj;
```

```
r = 9;

```

a) `broj` ostaje 5, a `r` postaje 9.
b) `broj` postaje 9.
c) Stvara se novi objekt neovisan o varijabli `broj`.
d) Kod nije valjan.

5. Nakon što je referenca povezana s objektom, može se preusmjeriti na drugi objekt običnim pridruživanjem.

   a) Točno
   b) Netočno

6. Koji parametar omogućuje funkciji promjenu izvornoga argumenta tipa `int`?

   a) `int x`
   b) `int& x`
   c) `const int& x`
   d) `const int x`

7. Konstantna referenca može pristupati objektu bez kopiranja, ali ga preko te reference ne može mijenjati.

   a) Točno
   b) Netočno

8. Koji je od navedenih povrataka opasan jer stvara viseću referencu?

   a) Vraćanje reference na globalni objekt.
   b) Vraćanje reference na objekt koji pozivatelj i dalje posjeduje.
   c) Vraćanje reference na lokalnu varijablu funkcije.
   d) Vraćanje objekta po vrijednosti.

9. Prema pravilima jezika ispravno povezana referenca može imati vrijednost `nullptr`.

   a) Točno
   b) Netočno

10. Koja je važna razlika između pokazivača i reference?

    a) Pokazivač se ne može dereferencirati.
    b) Referenca se mora povezati pri deklaraciji i poslije se ne može preusmjeriti.
    c) Referenca uvijek dinamički zauzima memoriju.
    d) Pokazivač ne može pohranjivati adresu.

B.9 Strukture, unije i pobrojeni tipovi

1. Članovi strukture deklarirane ključnom riječi `struct` podrazumijevano su javni.
   a) Točno
   b) Netočno
2. Kojim se operatorom pristupa članu objekta strukture?
   a) `.`
   b) `->`
   c) `*`
   d) `::`
3. Kojim se operatorom pristupa članu strukture preko pokazivača na tu strukturu?
   a) `.`
   b) `->`
   c) `&`
   d) `==`
4. Izrazi `p->godina` i `(*p).godina` ekvivalentni su ako `p` pokazuje na odgovarajući objekt strukture.
   a) Točno
   b) Netočno
5. Za razliku od strukture, svi članovi jedne unije dijele isti memorijski prostor.
   a) Točno
   b) Netočno
6. Veličina unije mora biti dovoljna za pohranu njezina najvećega člana, uz moguće dodatno poravnanje.
   a) Točno
   b) Netočno
7. Koja se ključna riječ koristi za deklaraciju pobrojenoga tipa?
   a) `union`
   b) `enum`
   c) `typedef`
   d) `using namespace`
8. Koju vrijednost podrazumijevano dobiva prvi identifikator neomeđenoga pobrojenog tipa ako nije zadana početna vrijednost?
   a) -1
   b) 0
   c) 1
   d) Ovisi o broju identifikatora.
9. Koju vrijednost ima identifikator `srijeda`?

```
enum Dan { ponedjeljak = 3, utorak, srijeda };

```

a) 2
b) 3
c) 4
d) 5

10. Čemu služi deklaracija tipa `typedef unsigned long Velicina;`?

a) Stvara novu varijablu naziva `Velicina`.
b) Uvodi zamjensko ime `Velicina` za postojeći tip.
c) Dinamički zauzima memoriju.
d) Pretvara sve cijele brojeve u realne.

B.10 Dinamička memorija

1. Koji se izraz u C++-u upotrebljava za dinamičko stvaranje objekta?

a) `malloc`
b) `new`
c) `create`
d) `alloc`

2. Rezultat izraza `new int` jest pokazivač na dinamički stvoren objekt tipa `int`.

a) Točno
b) Netočno

3. Kojim se zapisom oslobađa jedan objekt stvoren izrazom `new int`?

a) `delete p;`
b) `delete[] p;`
c) `free p;`
d) `remove(p);`

4. Kojim se zapisom oslobađa polje stvoreno izrazom `new double[20]`?

a) `delete p;`
b) `delete[] p;`
c) `free[] p;`
d) `clear(p);`

5. Pohrana pribavljena izrazom `new T[n]` smije se osloboditi izrazom `delete p` bez uglatih zagrada.

a) Točno
b) Netočno

6. Što je curenje memorije?

   a) Pokušaj čitanja tekstne datoteke.
   b) Gubitak mogućnosti pristupa dinamički zauzetoj memoriji koja nije oslobođena.
   c) Automatsko smanjenje veličine polja.
   d) Pretvorba pokazivača u referencu.

7. Uobičajeni new-izraz pri neuspjeloj dodjeli memorije podrazumijevano baca iznimku `std::bad_alloc`.

   a) Točno
   b) Netočno

8. Koja funkcija iz C standardne biblioteke mijenja veličinu bloka prethodno dobivenoga funkcijama iz iste skupine?

   a) `calloc()`
   b) `realloc()`
   c) `resize()`
   d) `delete[]`

9. Memorija dobivena funkcijom `malloc()` oslobađa se funkcijom `free()`.

   a) Točno
   b) Netočno

10. Koje je pravilo ispravno?

    a) `new` se uparuje s `free()`.
    b) `malloc()` se uparuje s `delete`.
    c) `new[]` se uparuje s `delete[]`.
    d) Svaki oblik dodjele oslobađa se naredbom `close()`.

11. C++ standard propisuje trajanje pohrane objekata, ali ne zahtijeva da svaka implementacija memoriju fizički organizira kao *stack* i *heap*.

    a) Točno
    b) Netočno

12. Gdje se u uobičajenoj implementaciji najčešće nalazi lokalna varijabla automatskoga trajanja?

    a) U *stack frameu* funkcije.
    b) Isključivo u datoteci izvornoga koda.
    c) Uvijek na *heapu*.
    d) U tablici ASCII znakova.

13. *Stack frame* može sadržavati parametre, lokalne objekte i podatke potrebne za povratak iz funkcije.

    a) Točno
    b) Netočno

14. Što najizravnije može uzrokovati *stack overflow*?

    a) Preduboka rekurzija bez pravodobnoga završetka.
    b) Poziv funkcije `free()` nad valjanim C-blokom.
    c) Uključivanje zaglavlja `<iostream>`.
    d) Uporaba konstantne reference.

15. Lokalni pokazivač i dinamički objekt na koji pokazuje mogu imati različito trajanje pohrane i nalaziti se u različitim memorijskim područjima.

    a) Točno
    b) Netočno

16. Što nakon izraza `delete p;` vrijedi za sam objekt pokazivača `p`?

    a) Automatski dobiva vrijednost `nullptr`.
    b) Prestaje postojati bez obzira na svoj scope.
    c) I dalje postoji, ali sadržava adresu uništenoga objekta dok mu se ne pridruži druga vrijednost.
    d) Pretvara se u referencu.

17. Pristup objektu preko pokazivača nakon što je objekt uništen izrazom `delete` dovodi do nedefiniranoga ponašanja.

    a) Točno
    b) Netočno

18. Koja tvrdnja najbolje razlikuje *free store* i *heap*?

    a) To su prema definiciji standarda uvijek potpuno isti pojmovi.
    b) *Free store* je C++ pohrana kojom upravljaju izrazi `new` i `delete`, a uobičajeno se ostvaruje na *heapu*.
    c) *Heap* sadržava samo globalne konstante.
    d) *Free store* služi isključivo spremanju programskoga koda.

19. RAII povezuje životni vijek resursa sa životnim vijekom objekta koji tim resursom upravlja.

    a) Točno
    b) Netočno

20. Za dinamički spremnik promjenjive veličine u modernome C++-u u pravilu je bolji prvi izbor:

    a) ručno polje dobiveno s `malloc()`
    b) `std::vector`
    c) globalno polje najveće moguće veličine
    d) pokazivač bez pridružene adrese

B.11 Klase

1. Klasa može sadržavati podatkovne i funkcijske članove.
   a) Točno
   b) Netočno
2. Koje je podrazumijevano pravo pristupa članovima deklariranima na početku klase uvedene ključnom riječi `class`?
   a) `public`
   b) `private`
   c) `protected`
   d) `global`
3. Konstruktor ima isti naziv kao klasa i nema naveden tip povratne vrijednosti.
   a) Točno
   b) Netočno
4. Koji je ispravan naziv destruktora klase `Tocka`?
   a) `delete Tocka()`
   b) `!Tocka()`
   c) `~Tocka()`
   d) `Tocka~()`
5. Konstruktor i destruktor mogu imati povratni tip `void`.
   a) Točno
   b) Netočno
6. Što znači preopterećenje konstruktora?
   a) Klasa ima više konstruktora s različitim popisima parametara.
   b) Konstruktor se poziva dvaput za isti objekt.
   c) Konstruktor vraća više vrijednosti.
   d) Konstruktor je proglašen prijateljskom funkcijom.
7. Koji operator pristupa javnomu članu `x` objekta `tocka`?
   a) `tocka->x`
   b) `tocka.x`
   c) `tocka::x`
   d) `*tocka.x`
8. Koji operator pristupa javnomu članu `x` preko pokazivača `p` na objekt klase?
   a) `p.x`
   b) `p::x`
   c) `p->x`
   d) `&p.x`

9. Ključna riječ `friend` može određenoj funkciji ili klasi omogućiti pristup nejavnim članovima klase.

   a) Točno
   b) Netočno

10. Koja se riječ navodi pri definiranju preopterećenoga operatora?

    a) `overload`
    b) `operator`
    c) `virtual`
    d) `override`

11. Članovi označeni kao `protected` dostupni su funkcijskim članovima izvedenih klasa.

    a) Točno
    b) Netočno

12. Koji zapis predstavlja javno nasljeđivanje klase `Osnovna`?

    a) `class Izvedena public Osnovna { };`
    b) `class Izvedena : public Osnovna { };`
    c) `class Osnovna : Izvedena public { };`
    d) `public class Izvedena(Osnovna) { };`

13. Čemu služi virtualna funkcija u osnovnoj klasi?

    a) Zabranjuje stvaranje objekata izvedene klase.
    b) Omogućuje polimorfni odabir redefinirane funkcije prema stvarnom tipu objekta.
    c) Svaki podatkovni član pretvara u javni.
    d) Automatski stvara konstruktor bez parametara.

14. Operator `::` upotrebljava se pri definiranju funkcijskoga člana izvan tijela klase kako bi se odredila njegova pripadnost klasi.

    a) Točno
    b) Netočno

15. Što se ispisuje?

```
class Pravokutnik {
    int sirina;
    int visina;
public:
    Pravokutnik(int s, int v) : sirina(s), visina(v) {}
    int povrsina() const { return sirina * visina; }
};

Pravokutnik p(4, 6);
cout << p.povrsina();

```

a) 10
b) 20
c) 24
d) 46

B.12 Namespace i scope resolution operator

1. Čemu prvenstveno služi `namespace`?

   a) Dinamičkoj dodjeli memorije.
   b) Organiziranju imena i sprječavanju njihovih sukoba.
   c) Automatskomu prevođenju funkcija.
   d) Promjeni prava pristupa članovima klase.

2. Operator `::` naziva se *scope resolution operator*.

   a) Točno
   b) Netočno

3. Što označuje zapis `std::cout`?

   a) Funkciju `std` pozvanu s argumentom `cout`.
   b) Ime `cout` iz namespacea `std`.
   c) Statički član objekta `std`.
   d) Pokazivač na standardni izlaz.

4. Dva različita namespacea mogu sadržavati funkcije jednakoga imena.

   a) Točno
   b) Netočno

5. Koji zapis uvodi samo ime `cout` iz namespacea `std`?

   a) `using std::cout;`
   b) `using namespace cout;`
   c) `namespace cout = std;`
   d) `import std.cout;`

6. Direktiva `using namespace std;` uvodi sva imena iz `std` kao zasebne globalne varijable.

   a) Točno
   b) Netočno

7. Zašto se `using namespace std;` ne preporučuje u zaglavnim datotekama?

   a) Zato što onemogućuje uključivanje zaglavlja `<iostream>`.
   b) Zato što može učiniti mnoga imena dostupnima u svakoj datoteci koja uključi zaglavlje i izazvati sukobe.
   c) Zato što briše namespace `std`.
   d) Zato što vrijedi samo u jeziku C.

8. Od standarda C++17 ugniježđeni se namespace može zapisati kao `namespace aplikacija::grafika { }`.
   a) Točno
   b) Netočno
9. Što čini zapis `namespace grafika = aplikacija::grafika;`?
   a) Stvara kopiju svih imena u novome namespaceu.
   b) Uvodi kraće zamjensko ime za postojeći namespace.
   c) Briše ugniježđeni namespace.
   d) Pretvara namespace u klasu.
10. *Scope resolution operator* `::` i operator pristupa članu objekta `.` imaju istu ulogu.
    a) Točno
    b) Netočno

B.13 Templates

1. Koja ključna riječ započinje definiciju *templatea*?
   a) `generic`
   b) `template`
   c) `typenameof`
   d) `pattern`
2. Funkcijski *template* omogućuje opis algoritma koji se može instancirati za različite prikladne tipove.
   a) Točno
   b) Netočno
3. Koji je zapis valjan početak funkcijskoga *templatea* s jednim tipskim parametrom `T`?
   a) `template <typename T>`
   b) `template (T)`
   c) `typename template T`
   d) `generic <T>`
4. Ključne riječi `typename` i `class` mogu u popisu parametara osnovnoga *templatea* uvesti tipski parametar.
   a) Točno
   b) Netočno
5. Što je instanciranje *templatea*?
   a) Brisanje definicije nakon prevođenja.
   b) Stvaranje konkretne funkcije ili klase za zadane argumente templatea.
   c) Pretvaranje klase u namespace.
   d) Dinamičko mijenjanje tipa objekta tijekom izvođenja.

6. Svaki se *template* može upotrijebiti sa svakim mogućim tipom, neovisno o operacijama u njegovoj definiciji.

   a) Točno
   b) Netočno

7. Koji zapis stvara objekt klasnoga *templatea* `Par<T>` za tip `int`?

   a) `Par p<int>;`
   b) `Par<int> p;`
   c) `template Par int p;`
   d) `Par::int p;`

8. Funkcijski i klasni templates temelj su mnogih dijelova standardne biblioteke, uključujući spremnike i algoritme.

   a) Točno
   b) Netočno

9. Zašto se definicija *templatea* najčešće nalazi u zaglavnoj datoteci?

   a) Zato što prevoditelj pri instanciranju mora vidjeti cijelu definiciju.
   b) Zato što se template izvršava prije predprocesora.
   c) Zato što template ne smije sadržavati funkcije.
   d) Zato što se u izvornoj datoteci ne smije pisati ključna riječ `template`.

10. Dedukcija argumenata klasnoga *templatea* (CTAD) može od standarda C++17 u prikladnim slučajevima odrediti argumente iz argumenata konstruktora.

    a) Točno
    b) Netočno

B.14 STL: vector, iteratori i algoritmi

1. Koje je zaglavlje potrebno uključiti za uporabu spremnika `std::vector`?

   a) `<array>`
   b) `<vector>`
   c) `<iterator>`
   d) `<memory>`

2. `vector<int> brojevi{1, 2, 3};` stvara spremnik s tri elementa.

   a) Točno
   b) Netočno

3. Koji funkcijski član dodaje element na kraj *vectora*?

   a) `push_back()`
   b) `push_front()`
   c) `append_front()`
   d) `insert_end_only()`

4. Koji funkcijski član uklanja zadnji element nepraznoga *vectora*?

   a) `erase_front()`
   b) `delete_back()`
   c) `pop_back()`
   d) `clear_last()`

5. Funkcijski član `size()` vraća broj trenutačno pohranjenih elemenata.

   a) Točno
   b) Netočno

6. Koja je razlika između `operator[]` i `at()`?

   a) `at()` provjerava granice, a `operator[]` ih ne provjerava.
   b) `operator[]` radi samo sa stringovima.
   c) `at()` uvijek dodaje novi element.
   d) Među njima nema nikakve razlike ni u načinu prijave pogreške.

7. Izraz `v.begin()` vraća iterator na prvi element, a `v.end()` iterator na položaj iza zadnjega elementa.

   a) Točno
   b) Netočno

8. Koje je zaglavlje potrebno za algoritam `sort()`?

   a) `<algorithm>`
   b) `<numeric>`
   c) `<utility>`
   d) `<fstream>`

9. Što čini poziv `sort(v.begin(), v.end());`?

   a) Sortira sve elemente `v` uz podrazumijevani poredak.
   b) Briše sve elemente `v`.
   c) Obrće redoslijed bez uspoređivanja.
   d) Sortira samo prvi i zadnji element.

10. Algoritam `find()` vraća iterator na pronađeni element ili krajnji iterator ako element nije pronađen.

    a) Točno
    b) Netočno

11. Koji algoritam prebrojava elemente jednake zadanoj vrijednosti?

    a) `count()`
    b) `sum()`
    c) `measure()`
    d) `capacity()`

12. Ponovna alokacija unutarnje pohrane *vectora* može učiniti postojeće iteratore, pokazivače i reference na njegove elemente nevaljanima.

    a) Točno
    b) Netočno

B.15 Lambda funkcije

1. Koji dio zapisa lambda funkcije određuje preuzimanje varijabli iz okoline?

    a) Uglate zagrade `[]`
    b) Obične zagrade `()`
    c) Vitičaste zagrade `{}`
    d) Operator `::`

2. Izraz `[] { return 42; }` definira lambda funkciju bez parametara i bez preuzimanja varijabli iz okoline.

    a) Točno
    b) Netočno

3. Koji zapis preuzima varijablu `x` po vrijednosti?

    a) `[x]`
    b) `[&x]`
    c) `[=x]`
    d) `(x)`

4. Koji zapis preuzima varijablu `x` po referenci?

    a) `[x]`
    b) `[&x]`
    c) `[*x]`
    d) `[ref x]`

5. Podrazumijevano se vrijednost preuzeta po vrijednosti ne može mijenjati u tijelu lambda funkcije bez oznake `mutable`.

    a) Točno
    b) Netočno

6. Što označuje zapis `[=]`?

   a) Podrazumijevano preuzimanje korištenih lokalnih varijabli po vrijednosti.
   b) Podrazumijevano preuzimanje po referenci.
   c) Zabranu svih parametara.
   d) Povratni tip `bool`.

7. Što označuje zapis `[&]`?

   a) Podrazumijevano preuzimanje korištenih lokalnih varijabli po referenci.
   b) Kopiranje samo globalnih varijabli.
   c) Povrat reference iz svake lambda funkcije.
   d) Pretvorbu lambda funkcije u pokazivač.

8. Generička lambda funkcija u C++14 i novijim standardima može imati parametar deklariran ključnom riječi **`auto`**.

   a) Točno
   b) Netočno

9. Koji se dio lambda funkcije može izostaviti kada nema parametara, a sintaksa ne zahtijeva druge specifikatore?

   a) Popis preuzimanja `[]`
   b) Prazan popis parametara `()`
   c) Tijelo `{}`
   d) Cijeli lambda izraz

10. Lambda funkcija često se predaje STL algoritmu kao predikat ili prilagođena funkcija usporedbe.

    a) Točno
    b) Netočno

B.16 Pametni pokazivači

1. Koje je zaglavlje potrebno uključiti za standardne pametne pokazivače?

   a) `<memory>`
   b) `<pointer>`
   c) `<resource>`
   d) `<newdelete>`

2. `std::unique_ptr` predstavlja jedinstveno vlasništvo nad objektom.

   a) Točno
   b) Netočno

3. Koja je preporučena funkcija za stvaranje objekta kojim upravlja `unique_ptr`?
   a) `std::make_unique`
   b) `std::make_shared_only`
   c) `std::alloc_unique`
   d) `std::new_ptr`
4. `unique_ptr` može se kopirati običnim operatorom pridruživanja.
   a) Točno
   b) Netočno
5. Kojom se funkcijom standardne biblioteke prenosi vlasništvo nad `unique_ptr` objektom?
   a) `std::move`
   b) `std::copy`
   c) `std::share`
   d) `std::forward_copy`
6. `std::shared_ptr` omogućuje da više pametnih pokazivača dijeli vlasništvo nad istim objektom.
   a) Točno
   b) Netočno
7. Što se s upravljanim objektom uobičajeno događa kada nestane posljednji vlasnički `shared_ptr`?
   a) Objekt se automatski uništava.
   b) Objekt postaje globalan.
   c) Objekt se kopira na *stack*.
   d) Program se mora ručno zaustaviti.
8. `std::weak_ptr` promatra objekt kojim upravlja `shared_ptr`, ali ne povećava broj vlasnika.
   a) Točno
   b) Netočno
9. Čemu posebno služi `weak_ptr`?
   a) Razbijanju ciklusa zajedničkoga vlasništva.
   b) Zamjeni svih običnih referenci.
   c) Ručnomu pozivanju destruktora.
   d) Pohrani isključivo cjelobrojnih vrijednosti.
10. Prije pristupa objektu preko `weak_ptr` uobičajeno se poziva `lock()` i provjerava dobiveni `shared_ptr`.
    a) Točno
    b) Netočno
11. Koja tvrdnja o funkciji `get()` pametnoga pokazivača vrijedi?
    a) Vraća obični pokazivač, ali ne prenosi vlasništvo.
    b) Uništava upravljani objekt.
    c) Povećava broj vlasnika za dva.
    d) Pretvara objekt u referencu sa statičkim trajanjem.

12. Ručno pozivanje `delete` nad adresom objekta kojim još upravlja pametni pokazivač ispravan je način ranijega oslobađanja.

    a) Točno
    b) Netočno

B.17 Rad s datotekama

1. Koje je zaglavlje potrebno uključiti za klase tokova datoteka?

    a) `<iostream>`
    b) `<fstream>`
    c) `<cstring>`
    d) `<iomanip>`

2. Koja je klasa namijenjena prvenstveno čitanju iz datoteke?

    a) `ofstream`
    b) `ifstream`
    c) `ostream`
    d) `stringstream`

3. Koja je klasa namijenjena prvenstveno pisanju u datoteku?

    a) `ofstream`
    b) `ifstream`
    c) `istream`
    d) `cin`

4. Klasa `fstream` može se koristiti i za čitanje i za pisanje, ovisno o načinu otvaranja.

    a) Točno
    b) Netočno

5. Koji funkcijski član otvara datoteku nakon što je objekt toka već stvoren?

    a) `open()`
    b) `is_open()`
    c) `close()`
    d) `begin()`

6. Funkcijski član `is_open()` provjerava je li tok trenutačno povezan s otvorenom datotekom.

    a) Točno
    b) Netočno

7. Koji način otvaranja označuje čitanje iz datoteke?

    a) `ios::in`
    b) `ios::out`
    c) `ios::app`
    d) `ios::trunc`

8. Koji način otvaranja dodaje novi sadržaj na kraj postojeće datoteke?

    a) `ios::in`
    b) `ios::app`
    c) `ios::trunc`
    d) `ios::binary`

9. Načini otvaranja mogu se kombinirati bitovnim operatorom `|`.

    a) Točno
    b) Netočno

10. Zašto je ispravno provjeravati uspjeh samoga čitanja, primjerice izrazom `while (getline(file, redak))`?

    a) Zato što petlja tada obrađuje samo uspješno pročitane retke.
    b) Zato što `getline()` uvijek vraća broj znakova u retku.
    c) Zato što se datoteka tako automatski otvara za pisanje.
    d) Zato što se svaki redak čita dvaput.

11. Koji funkcijski član pokazuje da prethodna operacija oblikovanoga čitanja nije uspjela?

    a) `fail()`
    b) `open()`
    c) `seekp()`
    d) `flush()`

12. Funkcija `clear()` ponovno postavlja zastavice stanja toka, ali sama ne uklanja nevaljani podatak iz ulaza.

    a) Točno
    b) Netočno

13. Koji funkcijski član postavlja položaj sljedećega čitanja?

    a) `seekg()`
    b) `seekp()`
    c) `tellp()`
    d) `clear()`

14. Koji funkcijski član vraća trenutačni položaj sljedećega zapisivanja?

    a) `tellp()`
    b) `tellg()`
    c) `seekg()`
    d) `eof()`

15. Oznaka `ios::beg` zadaje položaj u odnosu na početak datoteke.

    a) Točno
    b) Netočno

16. Kojim se funkcijskim članom iz binarne datoteke čita zadani broj bajtova?

    a) `read()`
    b) `write()`
    c) `getline()`
    d) `append()`

17. Sirovo zapisivanje bajtovne reprezentacije strukture uvijek daje prenosivu datoteku neovisnu o platformi i prevoditelju.

    a) Točno
    b) Netočno

18. Što u pravilu treba učiniti prije pozicioniranja toka nakon neuspjele operacije čitanja?

    a) Ponovno postaviti stanje funkcijom `clear()`.
    b) Izbrisati datoteku.
    c) Pozvati `pop_back()`.
    d) Promijeniti ekstenziju datoteke.

19. Zašto se pri određivanju veličine datoteke pozicioniranjem na kraj često koristi način `ios::binary`?

    a) Kako bi položaji pouzdano odgovarali bajtovima datoteke bez tekstnih pretvorbi.
    b) Kako bi se svi bajtovi pretvorili u znakove ASCII-ja.
    c) Kako bi se datoteka automatski komprimirala.
    d) Kako bi se onemogućilo čitanje.

20. Pri izmjeni čitanja i pisanja istim objektom `fstream` između operacija treba postaviti položaj ili obaviti odgovarajuću sinkronizaciju toka.

    a) Točno
    b) Netočno

Napomena

Pitanja su namijenjena samostalnoj pripremi. Pri ponavljanju nije dovoljno zapamtiti oblik naredbe: za svaki odabir treba moći objasniti zašto su ostale mogućnosti netočne.

Literatura i normativni izvori

Rukopis je usmjeren na standard C++17. Kada se opisuje svojstvo jezika, mjerodavan je normativni standard, dok su udžbenici i mrežni izvori namijenjeni dodatnom objašnjenju i praktičnim primjerima.

1. ISO/IEC. *ISO/IEC 14882:2017: Programming Languages – C++*. International Organization for Standardization, 2017.
2. Smith, Richard. *Working Draft, Standard for Programming Language C++*, dokument WG21 N4659, 21. ožujka 2017. Dostupno na: `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf`.
3. Stroustrup, Bjarne. *Programming: Principles and Practice Using C++*. 3. izd. Addison-Wesley, 2024.
4. Lippman, Stanley B.; Lajoie, Josée; Moo, Barbara E. *C++ Primer*. 5. izd. Addison-Wesley, 2012.
5. Josuttis, Nicolai M. *C++17 – The Complete Guide*. Leanpub, 2019. Prateći materijali dostupni su na: `https://www.cppstd17.com/`.
6. Stroustrup, Bjarne; Sutter, Herb, ur. *C++ Core Guidelines*. Standard C++ Foundation. Dostupno na: `https://isocpp.org/guidelines`.
7. *cppreference.com: C++ language and standard library reference*. Praktični referentni izvor dostupan na: `https://en.cppreference.com/w/cpp`.

Mrežni izvori provjereni su 8. kolovoza 2026. Za svojstva koja su se mijenjala u novijim standardima u tekstu se izričito navodi inačica standarda od koje je pojedina mogućnost dostupna.

Kazalo pojmova

adresa objekta, 94
algoritam, 1
algoritam standardne biblioteke, 200
aritmetika pokazivača, 94

binarna datoteka, 192
brojevni sustavi, 1

C-znakovni niz, 71
`capacity`, 200

datoteka, 192
datotečni tok, 192
delete-izraz, 122
destruktor, 142

`for`, 44
free store, 122
funkcija, 58

generičko programiranje, 178
`getline`, 86
grananje, 44

heap, 122

`if`, 44
inicijalizacija, 17
iterator, 200

klasa, 142
konstanta, 17
konstantna referenca, 106
konstruktor, 142

lambda funkcija, 210
    hvatanje varijabli, 210

`mutable`, 210

namespace, 142
nasljeđivanje, 142
new-izraz, 122
`nullptr`, 94

objekt klase, 142
operator, 17

pametni pokazivač, 219
parametar funkcije, 58
petlja, 44
pobrojeni tip, 112
pokazivač, 94
polimorfizam, 142
polje, 71
povratna vrijednost, 58
preopterećivanje funkcija, 58
prevoditelj, 1
prijenos argumenta po referenci, 106
programiranje, 1

RAII, 122
referenca, 106
rekurzija, 58
`reserve`, 200

`seekg`, 192
`seekp`, 192
`shared_ptr`, 219
stack, 122
stack frame, 122
standardna biblioteka, 1
stanje toka, 192
STL, 200
`string`, 86
struktura, 112
`switch`, 44

template, 178
    funkcijski, 178
    klasni, 178
tip podatka, 17
trajanje pohrane, 122

unija, 112
`unique_ptr`, 219

varijabla, 17

`vector`, 200
virtualna funkcija, 142
višedimenzijsko polje, 71
`volatile`, 142
`weak_ptr`, 219
`while`, 44

životni vijek objekta, 122