FILIP MARIĆ PREDRAG JANIČIĆ

UVOD U PROGRAMIRANJE

Osnove programiranja kroz programski jezik C++

Beograd
2026.

Sadržaj

1 Predgovor **9**

2 Uvod **11**

2.1 Osnovni elementi jezika i prvi programi 12

2.1.1 Osnovna struktura programa 12

2.1.1.1 Komentari 14

2.1.2 Promenljive, tipovi, ispisivanje i učitavanje podataka 14

2.1.3 Izračunavanje 20

2.1.3.1 Osnovne aritmetičke operacije i izrazi 20

2.1.3.2 Bibliotečke matematičke funkcije 21

2.1.3.3 Imenovane konstante 22

Zadatak: Rastojanje tačaka 22

2.1.4 Grananje 23

2.1.4.1 Relacijski operatori 23

2.1.4.2 Logički operatori 24

2.1.4.3 Naredba if 25

2.1.4.4 Uslovni izraz 26

2.1.5 Petlje 26

2.1.5.1 Petlja `while` 26

2.1.5.2 Petlja `for` 26

2.1.5.3 Petlja `do-while` 28

2.1.6 Definisanje funkcija 28

2.1.7 Strukture podataka 30

2.1.8 Upravljanje izuzecima i greškama 34

3 Promenljive i tipovi **37**

3.1 Promenljive, konstante i deklaracije 37

3.2 Osnovni tipovi podataka 39

3.2.1 Celobrojni tipovi 40

3.2.2 Realni tipovi 42

3.2.3 Logički tip . . . 43
3.2.4 Karakterski tip . . . 43
3.2.5 Niske . . . 44
3.3 Dodele vrednosti promenljivoj . . . 46
3.3.1 Operator dodele . . . 46
3.3.2 Razmena vrednosti promenljivih . . . 47
Zadatak: Cena hleba . . . 48
Rešenje . . . 48

4 Izrazi i izračunavanje 51
4.1 Aritmetički operatori i zapis matematičkih formula . . . 51
4.1.1 Složeni operatori dodele . . . 53
4.1.2 Inkrementiranje i dekrementiranje . . . 53
4.2 Zapis matematičkih formula . . . 54
4.3 Sekvencijalni programi . . . 55
4.3.1 Sekvencijalno izračunavanje vrednosti . . . 55
4.3.2 Celobrojno deljenje i ostatak . . . 56
4.3.3 Pozicioni zapis (brojevi, vreme, uglovi) . . . 56
4.3.3.1 Izračunavanje zbira cifara petocifrenog broja . . . 57
4.3.3.2 Razmenjivanje cifre jedinica i stotina . . . 57
4.3.3.3 Izračunavanje vremena između dva trenutka . . . 58
4.3.3.4 Izračunavanje ugla između kazaljki na satu . . . 60

5 Grananje 63
5.1 Relacijski i logički operatori i istinitosna vrednost izraza . . . 63
5.1.1 Logički tip podataka . . . 63
5.1.2 Relacijski i logički operatori . . . 63
5.1.3 Poređenje i poredak . . . 66
5.1.3.1 Relacija jednakosti . . . 66
5.1.3.2 Relacije poretka . . . 67
5.2 Naredba `if-else` . . . 71
5.2.1 Konstrukcija `else-if` . . . 73
5.3 Operator uslova . . . 74
5.4 Naredba `switch` . . . 75
5.5 Primeri . . . 76
5.5.1 Broj dana u mesecu (grananje na osnovu vrednosti promenljive) 76
5.5.2 Agregatno stanje vode (grananje na osnovu pripadnosti intervalu) . . . 79
5.5.3 Uspeh učenika . . . 80
5.5.4 Kvadrant kom pripada tačka (hijerarhija ugnežđenih uslova) . 80
5.5.5 Poređenje datuma (leksikografsko poređenje torki iste dužine) 81

5.5.6 Vrsta trougla na osnovu stranica 82

6 Petlje **85**
6.1 Petlja `while` 85
6.2 Petlja `for` 86
6.3 Petlja `do-while` 90
6.4 Naredbe `break` i `continue` 90
6.5 Osnovni iterativni algoritmi 92
6.5.1 Sabiranje, prebrojavanje, množenje 93
6.5.2 Minimum i maksimum 97
6.5.3 Linearna pretraga 100
6.5.4 Sortiranost niza 105
6.5.5 Filtriranje, preslikavanje 106
6.5.6 Pozicioni zapis 107
6.5.7 Leksikografsko poređenje 108
6.6 Ugnežđene petlje 109
6.6.1 Elementarni algoritmi sortiranja 110
6.6.1.1 Algoritam *selection sort* 111
6.6.1.2 Algoritam *bubble sort* 112
6.6.1.3 Algoritam *insertion sort* 113

7 Funkcije **115**
7.1 Modularnost i razlaganje problema na potprobleme 115
7.2 Primeri korišćenja funkcija 116
7.3 Parametri funkcije 119
7.4 Povratna vrednost funkcije 120
7.5 Funkcije i metode 121
7.6 Prenos argumenata 121
7.6.1 Prenos argumenata po vrednosti 122
7.6.2 Prenos argumenata po referenci 124
7.6.3 Prenos argumenata po adresi 127
7.7 Konverzije tipova argumenata funkcije 128
7.8 Anonimne funkcije 129
7.9 Složeni tipovi i funkcije 131
7.10 Rekurzivne funkcije – osnovni pregled 134
7.11 Doseg, životni vek i organizacija memorije 135
7.11.1 Doseg identifikatora 135
7.11.2 Životni vek objekata 137
7.11.3 Organizacija memorije dodeljene programu 139
7.11.4 Segment koda 139
7.11.5 Segment podataka 140

7.11.6 Stek segment . . . 140
7.11.7 Implementacija rekurzije . . . 141
7.12 Deklaracija i definicija funkcije . . . 142
7.12.1 Uzajamna rekurzija . . . 143
7.12.2 Razdvojena kompilacija i povezivanje . . . 145

8 Strukture podataka 149
8.1 Korisnički definisani tipovi: nabrojivi tip, strukture, klase . . . 149
8.1.1 Nabrojivi tipovi (`enum`) . . . 149
8.1.2 Strukture . . . 151
8.1.3 Klase . . . 157
8.1.4 Parovi i torke (tipovi `pair<T1, T2>` i `tuple<T1, ..., Tn>`) 160
8.1.5 Imenovanje tipova – `typedef` . . . 162
8.2 Strukture podataka sa sekvencijalnim pristupom . . . 163
8.2.1 Statički alocirani nizovi . . . 164
8.2.1.1 Nizovi i funkcije . . . 168
8.2.2 Tip `vector<T>` . . . 171
8.2.3 Pokazivači i iteratori . . . 174
8.2.4 Tip `list<T>` . . . 176
8.3 Višedimenzioni nizovi i kolekcije . . . 178
8.4 Rad sa matricama . . . 181
8.5 Strukture podataka sa asocijativnim pristupom . . . 185
8.5.1 Skupovi . . . 185
8.5.2 Multiskupovi . . . 186
8.5.3 Mape . . . 186
8.6 Specijalizovane strukture podataka . . . 191
8.6.1 Stek . . . 191
8.6.1.1 Primer upotrebe steka: izrazi u postfiksnoj notaciji . 191
8.6.2 Red . . . 193
8.6.2.1 Primer upotrebe reda: poslednjih k učitanih linija teksta . . . 193
8.6.3 Red sa dva kraja . . . 194
8.6.3.1 Primer upotrebe reda sa dva kraja: istorija veb-pregledača . . . 195
8.6.4 Red sa prioritetom . . . 196
8.6.4.1 Primer upotrebe reda sa prioritetom: zbir najvećih k brojeva . . . 197

9 Pregled standardne biblioteke 201
9.1 Korišćenje bibliotečke implementacije algoritama . . . 202
9.2 Bibliotečke funkcije za sekvencijalne kolekcije . . . 202

9.2.1 Sortiranje . . . 202
9.2.2 Linearna pretraga . . . 206
9.2.3 Binarna pretraga . . . 208
9.2.4 Kopiranje, preslikavanje, filtriranje . . . 209
9.2.5 Brisanje elemenata . . . 211
9.3 Rad sa karakterima . . . 211
9.4 Rad sa niskama . . . 212
9.5 Datoteke/tokovi . . . 214
9.5.1 Ulazni tok za učitavanje iz tekstualne datoteke (`ifstream`) . . . 215
9.5.2 Izlazni tok za upis u tekstualnu datoteku (`ofstream`) . . . 217
9.5.3 Baferovanje . . . 219
9.5.4 Odnos sa C bibliotekom za ulaz/izlaz . . . 220
9.5.5 Tokovi za čitanje iz niske (`istringstream`) i upis u nisku (`ostringstream`) . . . 220
9.5.6 Argumenti komandne linije programa . . . 220

10 Principi pisanja programa i dokumentacije 225
10.1 Timski rad i konvencije . . . 225
10.2 Vizuelni elementi programa . . . 226
10.2.1 Broj karaktera u redu . . . 226
10.2.2 Broj naredbi u redu, zagrade i razmaci . . . 227
10.2.3 Nazubljivanje teksta programa . . . 228
10.3 Imenovanje promenljivih i funkcija . . . 229
10.4 Pisanje izraza . . . 230
10.5 Korišćenje idioma . . . 232
10.6 Korišćenje konstanti . . . 233
10.7 Pisanje komentara . . . 235
10.8 Modularnost . . . 237
10.8.1 Modularnost i podela na funkcije . . . 237
10.8.2 Modularnost i podela na datoteke . . . 238
10.9 Upravljanje izuzecima i greškama . . . 238

11 Razvoj programa 243
11.1 Razvojno okruženje . . . 243
11.2 Pregled procesa debagovanja . . . 245
11.3 Testiranje . . . 246

12 Projektni zadaci 249
12.1 Prelazi između slika . . . 249
12.2 Transformacija slika . . . 253
12.3 ASCII kamera . . . 259
12.4 Mastermajnd . . . 262

13 Dodatak **267**
13.1 Statistike 267
13.2 Menjanje redosleda elemenata niza 269

1. Predgovor

Materijal koji je pred vama pisan je kao udžbenik za predmet „Uvod u programiranje“ sa prve godine smera Informatika na Matematičkom fakultetu u Beogradu.
Ovo je radna verzija materijala i u narednom periodu sigurno će se menjati i doterivati. Svi komentari i sugestije biće veoma dobrodošli. Na dosadašnjim komentarima zahvalni smo kolegi Mladenu Nikoliću i studentima Branku Basariću i Andrijani Milovanović.
oktobar 2026
Autori
Filip Marić
Predrag Janičić

2. Uvod

Jezik C++ (izgovara se obično „ce-plus-plus“) je viši programski jezik opšte namene, koji je inicijalno kreirao danski informatičar Bjarne Strustrup (dan. Bjarne Stroustrup). Prve verzije su objavljene 1985. godine i jezik je predstavljao proširenje programskog jezika C. Vremenom je jezik obogaćen mnogim novim svojstvima. Iako je prvobitno bio namenjen za sistemsko programiranje i programiranje uređaja sa ugrađenim računarom (engl. embedded systems), domen primene se vremenom proširio, te se C++ danas koristi za programiranje video-igara, servera, baza podataka i skoro svih velikih računarskih sistema.

Izvorni program, tj. program na jeziku C++, prevodi se na mašinski, izvršivi kôd kako bi mogao da se izvršava na računaru. Postoji veliki broj prevodilaca, tj. kompilatora koji ovo rade: GNU C++ compiler (g++), LLVM/Clang, Microsoft Visual C++, Intel C++ Compiler, itd.

Jezik C++ se vremenom razvijao i menjao i nove verzije jezika su uvodile i preporučivale sasvim drugačije stilove od originalne verzije jezika. Iako se, zbog kompatibilnosti unazad, C++ i dalje u velikoj meri može tumačiti kao nadskup programskog jezika C i većina C programa se može prevoditi pomoću prevodioca za C++, stil programiranja u savremenom jeziku C++ je veoma drugačiji nego što je to slučaj kada se programira u „čistom“ jeziku C. Danas je C++ jako veliki jezik koji omogućava brojne stilove programiranja i kombinuje više različitih programskih paradigmi (proceduralnu, imperativnu, objektno-orijentisanu, funkcionalnu, itd.). Cilj ovog udžbenika nije detaljno upoznavanje sa ovim programskim jezikom, već izučavanje osnovnih principa programiranja na jednom široko rasprostranjenom, savremenom jeziku koji ima bogatu standardnu biblioteku. Stoga će mnogi važni aspekti jezika biti potpuno izostavljeni (na primer, pokazivači i dinamička alokacija memorije, definisanje klasa i slično). Podskup jezika koji će biti obrađivan čini osnovni fragment jezika zajednički sa programskim jezikom C (bez pokazivača), sa dodatkom standardne biblioteke algoritama i kolekcija (koja u programskom jeziku C ne postoji).

2.1 Osnovni elementi jezika i prvi programi

U nastavku ćemo prikazati nekoliko jednostavnih programa kroz koje ćemo ilustrovati neke osnovne koncepte i jezičke konstrukcije, dovoljne za rešavanje mnogih jednostavnih zadataka korišćenjem programiranja. U narednim poglavljima, jezik C++ će biti prezentovan postupno, koncept po koncept, često iz opšte perspektive programiranja i programskih jezika.
Preporučujemo da se, radi boljeg razumevanja, svaki navedeni program prekuca, prevede i pokrene.

2.1.1 Osnovna struktura programa

Opišimo za početak jednostavan program koji na ekran ispisuje poruku `Zdravo svete!`.

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

int main() {
  // na ekran ispisujemo pozdravnu poruku
  cout << "Zdravo svete!" << endl;
  return 0;
}
```

Jezik C++ pravi razliku između malih i velikih slova i bitno je da li je nešto napisano malim ili velikim slovom.
Svaki program mora da sadrži *funkciju* `main`, tj. u kodu je potrebno da postoji deo oblika

```
int main() {
   // ovde se navode naredbe naseg programa
   return 0; // ovim se signalizira da je program uspesno
    ↪  izvrsen
}
```

Linija `int main()` započinje definiciju funkcije `main`. O funkcijama će biti više reči kasnije, za sada recimo samo da je funkcija `main` glavna funkcija i izvršavanje svakog C++ programa počinje izvršavanjem naredbi navedenih u okviru ove funkcije. Deo funkcije između vitičastih zagrada naziva se telo funkcije. Telo funkcije sadrži naredbe koje se izvršavaju kada se pozove ta funkcija. Kada korisnik pokrene program, tada operativni sistem pozove funkciju `main` tog programa i krene se sa izvršavanjem naredbi navedenih u njenom telu. Poslednja naredba u funkciji `main` je najčešće `return 0;` kojom naš

program operativnom sistemu vraća vrednost 0 i time javlja da je njegovo izvršavanje uspešno završeno. Ako je došlo do neke greške prilikom izvršavanja programa, funkcija može da vrati neku vrednost različitu od 0.
Centralni deo programa čini sledeći programski kôd:

```
// na ekran ispisujemo pozdravnu poruku
cout << "Zdravo svete!" << endl;
```

Linija `cout << "Zdravo svete!" << endl;` predstavlja *naredbu* kojom se na ekran ispisuje pozdravna poruka `Zdravo svete!` (bez dvostrukih navodnika), nakon čega se prelazi u novi red. Objekat `cout` (od engleskog „console output“) predstavlja *standardni izlaz*, što je najčešće ekran, i u njega se „uliva“ prvo tekst `Zdravo svete!`, a zatim i *prelazak u novi red* koji se označava sa `endl` (od engleskog „end line“). Ovo „ulivanje“ je predstavljeno simbolima `<<` (tekst „teče“ i uliva se na `cout`, pa se `cout` naziva i *standardni izlazni tok*).
Linija `// na ekran ispisujemo pozdravnu poruku` je *komentar* i on služi da onome ko čita ovaj program objasni šta se postiže nekim kodom, u ovom slučaju – narednom linijom. Komentari se prilikom prevođenja ignorišu i ne utiču na izvršavanje programa.
U prvoj liniji programa, *pretprocesorskom direktivom* `#include <iostream>` omogućavamo rad sa ulazno-izlaznim tokovima. Na sličan način može se omogućiti korišćenje drugih delova takozvane *standardne biblioteke*. Za svaki takav deo postoji odgovarajuće *zaglavlje*, kao što je `iostream` zaglavlje za rad sa ulazno-izlaznim tokovima. Navođenje direktive `#include <iostream>` omogućava da u našem programu možemo neki tekst da ispišemo na ekran, da neke vrednosti učitamo sa tastature i slično. U ovom programu koristili smo objekat `cout`, kojim ispisujemo tekst na ekran, i `endl`, koji koristimo kada želimo da pređemo u novi red. Da nismo naveli red `#include <iostream>`, dobili bismo poruku o tome da prevodilac našeg programa ne razume šta su `cout` i `endl`. Pošto će svaki program koji budemo pisali ispisivati nešto na ekran, svaki će koristiti direktivu `#include <iostream>`.
Instrukcija `using namespace std;` omogućava da se svi elementi standardne biblioteke koriste bez prefiksa `std::`. Na primer, izlazni tok se označava sa `cout`, te umesto da svuda pišemo da je on deo standardne biblioteke `std`, tj. da pišemo `std::cout`, možemo pisati samo `cout`. Da ne postoji red `using namespace std;`, tada bi centralni deo našeg programa morao da bude napisan u narednom obliku.

```
// na ekran ispisujemo pozdravnu poruku
std::cout << "Zdravo svete!" << std::endl;
```

2.1.1.1 *Komentari*

Već je rečeno da u kodu možemo pisati *komentare* – tekst kojim se objašnjava šta se u nekom delu programa radi i koji je namenjen onome ko bude čitao program (ili onome ko je taj program pisao, ako nekada kasnije bude potrebe da ga doradi ili prepravi). Komentare računar ignoriše prilikom prevođenja programa. U jeziku C++ komentar počinje navođenjem oznake `//` i prostire se do kraja tog reda (ovakve komentare često nazivamo linijskim komentarima). Komentar može i da se proteže kroz nekoliko susednih redova (to je, takozvani *višelinijski komentar*) i on počinje oznakom `/*`, a završava se oznakom `*/`. U daljem tekstu će se komentari navoditi mnogo više nego što je to uobičajena praksa, kako bi pomogli u razumevanju priloženih programa.

2.1.2 *Promenljive, tipovi, ispisivanje i učitavanje podataka*

Ispis na standardni izlaz (to je najčešće ekran računara tj. takozvana konzola) vrši se naredbom oblika `cout << "...";`, pri čemu se tekst koji se ispisuje navodi između dvostrukih navodnika. U jednom programu moguće je navesti i više ovakvih naredbi. Na primer,

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

int main() {
  cout << "Programiranje";
  cout << "Algoritmi";
  cout << "Strukture podataka";
  return 0;
}
```

Iako tekst programa ne mora biti složen ovako uredno (naredbe su uvučene, poravnate jedna ispod druge), to je veoma poželjno zbog čitljivosti programa. Kada se program pokrene, iako su naredbe složene jedna ispod druge, navedene rečenice se ispisuju jedna do druge.

```
ProgramiranjeAlgoritmiStrukture podataka
```

Isti efekat bi se postigao navođenjem jedne naredbe oblika:

```
cout << "Programiranje" << "Algoritmi" << "Strukture
 ↪  podataka";
```

ili malo drugačije složeno

```
cout << "Programiranje"
     << "Algoritmi"
     << "Strukture podataka";
```

Ako se želi da se nakon ispisa teksta pređe u novi red, onda je potrebno nakon niske pod dvostrukim navodnicima ispisati i znak za prelaz u novi red `endl`. Na primer, funkcija

```
int main() {
  cout << "Programiranje" << endl;
  cout << "Algoritmi" << endl;
  cout << "Strukture podataka" << endl;
  return 0;
}
```

ispisuje imena predmeta jedan ispod drugog:

```
Programiranje
Algoritmi
Strukture podataka
```

Tekst može da unese i korisnik programa. Razmotrimo naredni program.

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

int main() {
  cout << "Kako se zovete?" << endl;
  string ime;
  cin >> ime;
  cout << "Dobar dan, Vi se zovete " << ime << endl;
  return 0;
}
```

Naredbom `cout << "Kako se zovete?" << endl;` na ekran se ispisuje tekst `Kako se zovete?`, što je veoma slično prvom programu koji smo analizirali. Nakon toga želimo da korisnik unese svoje ime. Tekst koji korisnik unese moramo negde da upamtimo da bismo ga kasnije ispisali. Da bismo upamtili razne vrednosti (u ovom primeru to je

tekst koji je korisnik uneo, a u narednim primerima će to biti razni brojevi sa kojima ćemo vršiti različita izračunavanja) koristimo *promenljive*. U navedenom primeru, koristimo promenljivu koja se zove `ime` i u nju smeštamo tekst koji je uneo korisnik. Možemo da zamislimo da svakoj promenljivoj odgovara kutijica ili kućica u kojoj se čuva njena vrednost. U svakom trenutku postojeća vrednost može biti promenjena, tj. izbačena iz kutijice i u kutijicu može biti upisana neka nova vrednost. Zato se promenljive i zovu tako. Jezik C++ spada u grupu takozvanih *statički tipiziranih jezika*, što znači da se za svaku promenljivu unapred zadaje njen *tip*, tj. vrsta vrednosti koje se u njoj mogu čuvati. U nekim promenljivim može da se čuva tekst, u drugima celi brojevi, u trećim realni brojevi i slično. Prilikom prvog uvođenja neke promenljive u naš program, pored njenog imena obavezno je navesti njen tip i to čini *deklaraciju* promenljive. U prethodnom primeru, deklaracija je bila linija `string ime;` Njom smo deklarisali promenljivu pod nazivom `ime` i rekli da će ona biti tipa `string`, tj. da će se u njoj čuvati tekst.
Naredbom `cin >> ime` učitava se tekst (jedna niska karaktera) koji je uneo korisnik. Objekat `cin` (od engleskog „console input“) označava *standardni ulaz* i on najčešće odgovara tastaturi. Očekujemo da korisnik unese svoje ime (mada može da unese šta god želi – naš program to neće primetiti). Podaci opet *teku*, ali ovog puta teku sa ulaza tj. sa tastature u promenljivu `ime` (što je naglašeno simbolima `>>`). Zato se `cin` naziva i *standardni ulazni tok*.
Na kraju, naredbom `cout << "Dobar dan, Vi se zovete " << ime << endl`, na standardni izlaz ispisuje se prvo tekst `Dobar dan, Vi se zovete`, zatim sadržaj promenljive `ime` (to je tekst koji je korisnik uneo) i na kraju se prelazi u novi red.
Učitavanjem teksta naredbom oblika `cin >> tekst;` učitava se samo jedna reč tj. tekst do prvog razmaka. Na primer, ako kao odgovor na `cin >> ime;` u prethodnom programu neko otkuca `Petar Petrovic`, promenljiva `ime` će sadržati samo tekst `Petar`.
Čitava linija teksta može se uneti korišćenjem funkcije `getline`. Na primer:

```
string ime_i_prezime;
getline(cin, ime_i_prezime);
```

U ovom slučaju, korisnik može da unese celo ime i prezime i ono će biti smešteno u promenljivu koja je nazvana `ime_i_prezime`.
U prethodnom primeru videli smo kako može da se koristi tekstualni tip podataka `string`. U prvim programima ćemo koristiti i sledeće osnovne tipove podataka (a kasnije ćemo upoznati i mnoge druge).

tip	opis	primer
`string`	tekst (niska karaktera)	`"Zdravo"`
`int`	ceo broj	`1234`
`double`	realan broj	`3.141`

tip	opis	primer
`bool`	logička vrednost	`true` ili `false`

Tipove koji čuvaju neku vrstu brojeva zovemo brojevni tipovi. Treba imati na umu da brojevne promenljive ne mogu da čuvaju proizvoljno male i proizvoljno velike brojeve. Na primer, u promenljivoj tipa `int` najčešće se mogu čuvati celobrojne vrednosti od oko minus dve milijarde pa sve do oko dve milijarde. Slično važi za podatke tipa `double` – i ovaj tip ima ograničeni raspon i preciznost tj. broj decimala. Skup mogućih vrednosti sasvim je dovoljan za početne zadatke, te na početku nećemo obraćati previše pažnje na ograničenja opsega.
Prvobitna dodela vrednosti promenljivoj naziva se *inicijalizacija*. Inicijalizacija može biti navedena u okviru same deklaracije i takvu deklaraciju zovemo *deklaracija sa inicijalizacijom*. Prilikom dodele moguće je promenljivim dodeljivati vrednost nekih izraza ili neke konkretne vrednosti tj. konstante. Na primer:

```
string ime = "Bjarne Stroustrup";
```

Pre nego što se promenljiva tipa `string` inicijalizuje, njena vrednost je prazan tekst (tekst `""`). Međutim, pre nego što je lokalna brojevna promenljiva (deklarisana unutar funkcije) inicijalizovana, njena vrednost je nešto što je zatečeno u njenom prostoru od ranije i nije nužno nula. Čitanje takve vrednosti dovodi do nedefinisanog ponašanja programa, pa lokalne promenljive treba inicijalizovati pre prve upotrebe. U narednom kodu, u opštem slučaju, ne možemo znati vrednost promenljive `x` između njene deklaracije i inicijalizacije:

```
int x;
...
x = 1;
```

Imena promenljivih treba da budu u skladu sa njihovim značenjem (poželjno je izbegavati kratka, neinformativna imena poput `a`, `b`, `x`, `y`, osim ako iz konteksta programa nije potpuno jasno šta bi te promenljive mogle da označavaju). Imena promenljivih ne smeju da sadrže razmake i moraju biti sastavljena samo od slova engleske abecede, cifara i donje crte tj. podvlake (karaktera `_`), ali ne mogu počinjati cifrom.
Prikažimo još neke primere deklaracija i inicijalizacija. Na primer, narednom deklaracijom se u program uvode dve promenljive pod nazivima `x` i `y` i kaže se da će one čuvati celobrojne vrednosti.

```
int x, y;
```

Primetimo da smo u prethodnom primeru jednom deklaracijom uveli dve promenljive, što je kraće nego da smo pisali posebno dve deklaracije:

```
int x;
int y;
```

Naravno, i celobrojne promenljive mogu biti inicijalizovane u okviru deklaracije:

```
int x = 1, y = 2;
int a = 3, b, c = 4;
```

U prethodnom primeru deklarisano je pet promenljivih, a inicijalizovane su četiri.
Kada su u pitanju realne vrednosti, one se navode sa decimalnom tačkom (u skladu sa pravopisom engleskog jezika), a ne sa zapetom (što bi bilo u skladu sa pravopisom srpskog jezika):

```
double pi = 3.14159265;
```

Naredba ispisa koju smo ranije videli može biti upotrebljena i za ispis brojevnih, pa i logičkih vrednosti. Na primer, narednim naredbama

```
cout << 123 << endl;
int x = 5;
cout << x << endl;
cout << 12.345 << endl;
double pi = 3.14159265;
cout << pi << endl;
```

ispisuje se

```
123
5
12.345
3.14159
```

Prilikom ispisa moguće je kombinovati tekst i brojevne vrednosti. Na primer,

```
double pi = 3.1415926;
cout << "Vrednost broja pi je " << pi << endl;
```

Čest zahtev je da se realne vrednosti ispišu zaokružene na zadati broj decimala. Za to je moguće na početku programa navesti direktivu `#include <iomanip>`, a zatim koristiti sledeći oblik naredbe ispisa:

```
double x = 123.4567;
cout << fixed << showpoint << setprecision(2) << x << endl;
```

Elementi `fixed`, `showpoint` i `setprecision` nazivaju se *manipulatori* toka, jer utiču na način na koji tok `cout` ispisuje podatke. Navođenjem manipulatora `fixed` postiže se da se nikada ne koristi tzv. naučni oblik zapisa (npr. $3{,}5 \cdot 10^9$) koji je pregledniji za jako male i jako velike brojeve. Manipulatorom `showpoint` postiže se da se decimalna tačka i decimale ispisuju i kada su jednake nuli (na primer, `3.00` umesto `3`). Uz `fixed` se to već podrazumeva, pa je `showpoint` tada suvišan, ali ga navodimo jer eksplicitno iskazuje nameru. Manipulatorom `setprecision(n)` podešava se preciznost ispisa: ako je pre njega naveden `fixed`, broj se zaokružuje na `n` decimala, a ako nije, broj se zaokružuje na `n` značajnih cifara. Tako naredba iz prethodnog primera ispisuje `123.46`, dok bi bez `fixed` i `showpoint` naredba `cout << setprecision(2) << x` ispisala `1.2e+02`. Slično, `cout << setprecision(3) << 3.14159` daje izlaz `3.14`, a `cout << setprecision(4) << 3.14159` daje izlaz `3.142`.
Pored teksta, sa standardnog ulaza možemo učitavati i brojeve. Razmotrimo naredni program.

```
cout << "Koliko imate godina?" << endl;
int brGodina;
cin >> brGodina;
cout << "Zdravo, Vi imate " << brGodina << " godina." <<
 ↪  endl;
```

Nakon učitavanja jednog celog broja sa tastature, korisniku se ispisuje odgovarajući tekst.
Ukoliko je potrebno uneti više vrednosti sa tastature (na primer, dan i mesec rođenja), onda se u jednoj naredbi može nadovezati učitavanje svih potrebnih vrednosti. Pritom je te vrednosti moguće uneti u jednoj liniji razdvojene razmakom ili u dve zasebne linije.

```
cout << "Kad ste rodjeni?" << endl;
int dan, mesec;
cin >> dan >> mesec;
cout << "Zdravo, Vi ste rodjeni " << dan << ". " << mesec <<
 ↪  "." << endl;
```

2.1.3 Izračunavanje

2.1.3.1 Osnovne aritmetičke operacije i izrazi

Nijedan od programa koje smo do sada sreli nije bio naročito interesantan. Mogli smo da učitamo podatke i da ih ispišemo u neizmenjenom obliku.
Računar je mašina koja obrađuje podatke, tj. koja primenjujući računske operacije na osnovu ulaznih podataka dobija izlazne. U računaru je, na najnižem nivou, sve zapisano pomoću brojeva i sve operacije se svode na osnovne operacije nad brojevima. Računar ili kompjuter (engl. computer) je sprava koja računa, tj. sprava koja je napravljena tako da može veoma brzo i efikasno da izvodi operacije nad brojevima. Računanje se naziva i *aritmetika* (od grčke reči ἀριθμός tj. aritmos koja znači broj, brojanje, računanje), a računske operacije se nazivaju i *aritmetičke operacije*.

- Osnovna aritmetička operacija je sabiranje. Zbir brojeva 3 i 5 se u matematici predstavlja kao $3 + 5$. U programskom jeziku C++ koristi se identičan zapis: `3 + 5`. Sabiranje je primenljivo i na cele i na realne brojeve. Na primer, kôd koji učitava i sabira dva cela broja može biti napisan na sledeći način.

  ```
  int x, y;
  cin >> x >> y;
  cout << x + y << endl;
  ```

- Pored sabiranja možemo razmatrati i oduzimanje. Razlika brojeva 8 i 2 se u matematici predstavlja kao $8 - 2$, a u programskom jeziku C++ koristi se identičan zapis: `8 - 2`. Oduzimanje je primenljivo i na cele i na realne brojeve.

- Još jedna od osnovnih operacija je množenje. Proizvod brojeva 4 i 6 se u matematici predstavlja kao $4 \cdot 6$. U programskom jeziku C++ množenje se označava pomoću operatora `*` i proizvod brojeva 4 i 6 se zapisuje kao `4 * 6`.

- U programskom jeziku C++, naravno, možemo i da delimo, da izračunavamo ostatak pri deljenju i ceo deo količnika. Deljenje realnih brojeva se vrši pomoću operatora `/` i količnik brojeva 7,2 i 6,4 se zapisuje kao `7.2 / 6.4`. Deljenjem dva

cela broja dobija se njihov celobrojni količnik, dok se ostatak pri deljenju dva cela broja dobija operatorom `%`. Na primer, vrednost izraza `14 / 4` jednaka je `3`, a izraza `14 % 4` jednaka je `2`. Ako želimo da odredimo realni količnik dva cela broja, moramo ih predstaviti u realnom obliku (na primer, umesto `14/4` pišemo `14.0/4.0`). Moguće je primeniti i *eksplicitnu konverziju* celih u realne vrednosti navođenjem `(double)` ispred naziva promenljive (npr. umesto `x / y` pišemo `(double) x / (double) y`). Naglasimo da je dovoljno da bilo deljenik bilo delilac budu realni da bi se primenilo realno deljenje. O dubljim vezama između realnog i celobrojnog tipa biće više reči u kasnijim poglavljima.

Slično kao i u matematici, od konstantnih vrednosti i promenljivih, primenom operatora i zagrada grade se *izrazi*. *Prioritet operatora* je usklađen sa uobičajenim prioritetom u matematici, pa je prioritet operatora `*`, `/` i `%` viši od prioriteta operatora `+` i `-`, dok svi navedeni operatori imaju levu asocijativnost (računske operacije se izvode sleva nadesno). Prioritet i asocijativnost se mogu promeniti navođenjem zagrada.
U prvim programima ćemo se truditi da prilikom izvođenja operacija ne mešamo podatke različitog tipa. Ipak, naglasimo da ako je u izrazu jedan broj realan, a drugi ceo, pre izvođenja operacija se taj ceo broj pretvara u realan i operacija se izvršava nad realnim brojevima.
Više puta u prethodnom tekstu pominjali smo tip realnih brojeva. Međutim, vrednosti tog tipa čine samo konačan podskup skupa realnih brojeva: nije moguće zapisati proizvoljno male, proizvoljno velike brojeve, iracionalne brojeve, itd. Obično se realni brojevi čuvaju u zapisu koji zovemo *zapis u pokretnom zarezu*, te je preciznije, na primer, tip `double` zvati *tip brojeva u pokretnom zarezu*, a ne *tip realnih brojeva*. Isto važi i za druge tipove koji mogu da čuvaju (neke) realne vrednosti. Ipak, u svakodnevnom govoru, često se može čuti i termin *tip realnih brojeva*, iako nije precizan. Analogno važi za cele brojeve, jer u računaru nije moguće pohraniti bilo koji ceo broj. Zato bi bilo preciznije govoriti *tip celih brojeva fiksne širine*, umesto uobičajenog *tip celih brojeva*.

2.1.3.2 *Bibliotečke matematičke funkcije*

U mnogim konkretnim primenama, pored osnovnih aritmetičkih operacija primenjuju se i neke naprednije *matematičke funkcije* i neke značajne *matematičke konstante* (npr. π). Da bismo ih koristili u jeziku C++, potrebno je na početku programa uključiti zaglavlje `<cmath>` direktivom `#include <cmath>`. Funkcije koje će biti potrebne u narednim zadacima su:

- `pow(x, y)` – izračunava stepen x^y, pri čemu se može primeniti i za izračunavanje korena (ne samo kvadratnih) znajući da je $\sqrt[n]{x} = x^{\frac{1}{n}}$;
- `sqrt(x)` – izračunava kvadratni koren $\sqrt{x}$;
- `abs(x)` – izračunava apsolutnu vrednost $|x|$;
- `sin(x)`, `cos(x)`, `tan(x)` – izračunavaju sinus, kosinus i tangens ugla zadatog u radijanima

Detaljniji spisak matematičkih funkcija biće prikazan u narednim poglavljima.

2.1.3.3 *Imenovane konstante*

Vrednosti koje su nam potrebne u programu, a koje se neće menjati tokom izvršavanja programa, možemo definisati u vidu *imenovanih konstanti*, koje se definišu kao obične promenljive, uz navođenje ključne reči `const` pre tipa podatka, uz obaveznu inicijalizaciju. Na primer:

```
const double PI = 3.14159265;
```

Imenovanim konstantama nije moguće promeniti vrednost u programu. Vrednost kojom se konstanta inicijalizuje ne mora biti poznata u vreme pisanja programa – može biti i vrednost izraza koja se izračunava tek tokom izvršavanja (na primer, `const double obim = 2 * r * PI;`, gde je `r` prethodno učitana vrednost).

Zadatak: Rastojanje tačaka

Napiši program koji izračunava i ispisuje rastojanje između tačaka zadatih svojim koordinatama.

Opis ulaza

Sa standardnog ulaza unose se četiri realna broja, svaki u posebnom redu. Prva dva broja A_x i A_y predstavljaju koordinate tačke $A = (A_x, A_y)$, dok druga dva broja B_x i B_y predstavljaju koordinate tačke $B = (B_x, B_y)$.

Opis izlaza

Na standardni izlaz ispisati jedan realan broj koji predstavlja rastojanje između tačaka A i B.

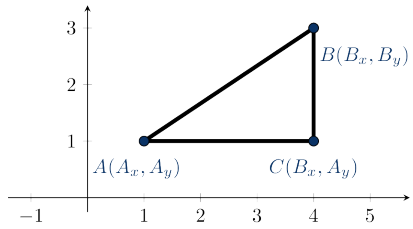
Primer

Ulaz	*Izlaz*

`0`
`0`
`1`
`1`
	`1.41421`

Rešenje

Posmatrajmo tačku C koja ima koordinate (B_x, A_y). Trougao ABC je pravougli trougao sa pravim uglom kod temena C. Traženo rastojanje između tačaka A i B jednako je dužini hipotenuze tog trougla i može se izračunati primenom Pitagorine teoreme koja tvrdi da je kvadrat nad hipotenuzom jednak zbiru kvadrata nad obe katete. Pošto su katete našeg trougla duži AC i BC, važi da je $|AB|^2 = |AC|^2 + |BC|^2$, pa je $|AB| = \sqrt{|AC|^2 + |BC|^2}$. Pošto tačke A i C imaju istu y-koordinatu, dužina duži AC jednaka je $|B_x - A_x|$. Zaista, pošto je duž AC paralelna osi x, njena dužina jednaka je dužini intervala koji predstavlja njenu projekciju na tu osu. To je interval $[A_x, B_x]$ ako je $A_x \leq B_x$ i njegova dužina



Slika 2.1: Rastojanje tačaka

je $B_x - A_x$, tj. interval $[B_x, A_x]$ ako je $B_x \leq A_x$ i njegova dužina je $A_x - B_x$. U oba slučaja, dužina je jednaka $|B_x - A_x|$. Slično, dužina duži BC jednaka je $|B_y - A_y|$. Zato je $|AB| = \sqrt{|B_x - A_x|^2 + |B_y - A_y|^2}$. Pošto se vrednosti $|B_x - A_x|$ i $|B_y - A_y|$ kvadriraju, nije neophodno koristiti apsolutnu vrednost i važi da je $|AB| = \sqrt{(B_x - A_x)^2 + (B_y - A_y)^2}$. Podsetimo se, u jeziku C++ se kvadratni koren može izračunati bibliotečkom funkcijom `sqrt` deklarisanom u zaglavlju `<cmath>`.

```
#include <iostream>
#include <cmath>

using namespace std;

int main() {
  double ax, ay, bx, by;
  cin >> ax >> ay >> bx >> by;
  double dx = bx - ax, dy = by - ay;
  double d = sqrt(dx*dx + dy*dy);
  cout << d << endl;
  return 0;
}
```

2.1.4 Grananje

2.1.4.1 Relacijski operatori

Često je potrebno utvrditi da li su neke dve vrednosti međusobno jednake ili za neke dve vrednosti utvrditi koja je od njih veća. Za poređenje vrednosti promenljivih ili izraza koriste se *relacijski operatori*.

- Osnovni relacijski operator je operator provere jednakosti ==. Na primer, ako že-

limo da ispitamo da li promenljive `a` i `b` imaju istu vrednost, to se može postići relacijskim izrazom `a == b`. Vrednost ovog izraza je `true` ako su vrednosti promenljivih jednake, a `false` inače. Vrednost ovog izraza je tipa `bool` i najčešće se koristi prilikom grananja (o kome će uskoro biti reči), ali se, takođe, vrednost relacijskog izraza može i dodeliti promenljivoj tipa `bool`. Dešava se da se prilikom pisanja koda napravi greška i umesto operatora provere jednakosti `==` iskoristi operator dodele `=`. Napomenimo još i to da poređenje dva realna broja može proizvesti ponašanje koje je drugačije od očekivanog zbog nepreciznosti zapisa realnih vrednosti.

- Pored provere jednakosti, možemo vršiti proveru da li su dve vrednosti različite. To se postiže operatorom `!=`. Uslov da promenljive `a` i `b` imaju različitu vrednost zapisuje se kao `a != b`.
- Za poređenje da li je jedna vrednost manja, manja ili jednaka, veća, veća ili jednaka od druge vrednosti koriste se redom relacijski operatori `<`, `<=`, `>`, `>=`.

Očekivano, relacijski operatori su nižeg prioriteta u odnosu na aritmetičke operatore, pa bi se u izrazu `2+3 == 6-1` najpre izračunale vrednosti `2+3` i `6-1`, a tek onda bi se proveravala jednakost ove dve izračunate vrednosti. Operatori `<`, `<=`, `>`, `>=` su višeg prioriteta od operatora `==` i `!=`. Svi relacijski operatori su levo asocijativni.

2.1.4.2 *Logički operatori*

Za zapis složenih uslova koriste se *logički operatori*. Logički operatori primenjuju se na operande koji su tipa `bool` i daju rezultat tipa `bool`. Binarni logički operatori `&&` i `||` su nižeg prioriteta u odnosu na relacijske i aritmetičke operatore, dok je unarni operator `!` višeg prioriteta od svih binarnih operatora.

- Operator logičke konjunkcije `&&` koristi se za utvrđivanje da li istovremeno važi neki skup uslova. Na primer, vrednost izraza `2 < 3 && 2 > 1` je `true`, a vrednost izraza `2 < 3 && 2 < 1` je `false`.
- Operatorom logičke disjunkcije `||` utvrđuje se da li je tačan bar jedan od datih uslova. Na primer, izraz `2 < 3 || 2 < 1` ima vrednost `true`, a izraz `2 > 3 || 2 < 1` vrednost `false`.
- Operator `!` daje logičku negaciju. Na primer, izraz `!(1 < 3)` ima vrednost `false`, koja je suprotna od vrednosti `true` izraza `1 < 3`.

Operacije logičke konjunkcije i disjunkcije definisane su sledećim tablicama.

&&	false	true
false	false	false
true	false	true

||	false	true
false	false	true
true	true	true

!	
false	true
true	false

Operator logičke konjunkcije višeg je prioriteta od operatora logičke disjunkcije. Dakle, u izrazu `a || b && c` bi se prvo izračunala vrednost izraza `b && c`, a onda bi se izvršila operacija logičke disjunkcije promenljive `a` i vrednosti prethodnog izraza. Oba binarna logička operatora su levo asocijativna.
I za operator konjunkcije i za operator disjunkcije karakteristično je *skraćeno izračunavanje* – iako su pomenuti operatori binarni, vrednost drugog operanda se ne računa ukoliko je vrednost kompletnog izraza već određena vrednošću prvog operanda. Dakle, prilikom izračunavanja vrednosti izraza `A && B`, ukoliko je vrednost izraza `A` jednaka `false`, nema potrebe i ne izračunava se vrednost izraza `B`. Slično, prilikom izračunavanja vrednosti izraza `A || B`, ukoliko je vrednost izraza `A` jednaka `true`, ne izračunava se vrednost izraza `B`.

2.1.4.3 *Naredba if*

Za mnoge programe tok izvršavanja nije uvek isti, već zavisi od ispunjenosti određenih uslova. Za takve programe kažemo da imaju *razgranatu strukturu* i da se u njima vrši *grananje*. Cilj grananja jeste da se na osnovu ispunjenosti (ili neispunjenosti) nekog uslova odredi koju narednu naredbu treba izvršiti. Većina programskih jezika, pa i jezik C++, raspolaže naredbom grananja. Osnovni oblik naredbe grananja u jeziku C++ je:

```
if (uslov)
   naredba1
else
   naredba2
```

U navedenom primeru, ako je ispunjen uslov `uslov`, biće izvršena prva naredba, a ako uslov nije ispunjen, biće izvršena druga naredba. Na primer, ispisivanje da li je dati broj paran ili neparan može imati sledeći oblik:

```
if (broj % 2 == 0)
   cout << "paran" << endl;
else
   cout << "neparan" << endl;
```

Stavka `else` nije obavezan deo naredbe grananja. Dakle, ako bismo hteli da ispišemo da je broj paran ako jeste paran, a ako nije da ne ispisujemo ništa, to bismo mogli da postignemo narednom naredbom:

```
if (broj % 2 == 0)
   cout << "paran" << endl;
```

Umesto pojedinačnih naredbi, u obe grane se može javiti i blok naredbi naveden u vitičastim zagradama.

2.1.4.4 *Uslovni izraz*

Umesto naredbe grananja nekada je pogodnije iskoristiti uslovni izraz (izraz grananja). Uslovni izraz, tj. operator `?:`, ima sledeću formu:

```
uslov ? rezultat_tacno : rezultat_netacno
```

Ovaj operator je ternarni, tj. ima tri argumenta: prvi je uslov čiju ispunjenost proveravamo, drugi argument je vrednost izraza ako je uslov ispunjen, dok se trećim argumentom zadaje vrednost izraza ako uslov nije ispunjen. Na primer, ispisivanje parnosti zadatog broja moglo bi da se realizuje i korišćenjem izraza grananja:

```
cout << (broj % 2 == 0 ? "paran" : "neparan") << endl;
```

Operator grananja je desno asocijativan i nižeg je prioriteta od skoro svih ostalih operatora (izuzetak su operatori dodele, koji su istog prioriteta).

2.1.5 *Petlje*

Petlje se koriste kada neke naredbe treba da se izvrše veći broj puta. U jeziku C++ postoje tri vrste petlji: `while`, `for` i `do-while`.

2.1.5.1 *Petlja `while`*

Osnovni oblik petlje `while` je:

```
while (uslov)
    telo
```

U petlji `while` ispituje se vrednost logičkog izraza `uslov` i dok god je on tačan, izvršavaju se naredbe zadate unutar tela petlje i izvršavanje se vraća na početak (na proveru uslova). Svako izvršavanje tela petlje nazivaćemo jednom *iteracijom*. Ako se telo petlje sastoji od više naredbi, one moraju biti navedene unutar vitičastih zagrada.

2.1.5.2 *Petlja `for`*

Opšti oblik petlje `for` je:

```
for (inicijalizacija; uslov; korak)
    telo
```

Petlja `for` najčešće se koristi tako što se promenljivoj (koja se često naziva *brojačka promenljiva*) redom dodeljuju vrednosti od najmanje do najveće i za svaku od tih vrednosti se izvršavaju naredbe u okviru tela petlje (ako ih je više, moraju se navesti u vitičastim zagradama). Obično se u delu `inicijalizacija` postavlja početna vrednost brojačke promenljive (najčešće se na tom mestu i deklariše brojačka promenljiva), u delu `uslov` se zadaje uslov petlje koji se proverava u svakoj iteraciji i prvi put kada nije ispunjen, izvršavanje petlje se prekida, dok se u delu `korak` menja vrednost brojačke promenljive. Na primer, ukoliko želimo da ispišemo sve brojeve iz intervala `[a, b]`, to možemo da uradimo narednom petljom:

```
for (int i = a; i <= b; i++)
    cout << i << endl;
```

Svaka petlja `for` može se jednostavno izraziti pomoću petlje `while`. Inicijalizaciju je potrebno izvršiti neposredno pre petlje `while`, uslov petlje ostaje isti, dok se korak petlje dodaje kao poslednja naredba u telu petlje `while`. Dakle, prethodno ispisivanje brojeva iz intervala `[a, b]` mogli smo da realizujemo i na sledeći način:

```
int i = a;
while (i <= b) {
   cout << i << endl;
   i++;
}
```

Korišćenjem petlje `for` možemo lako ostvariti ponavljanje istih naredbi određeni broj puta. Na primer, naredni program izračunava i ispisuje površine `n` pravougaonika čije se dužine stranica unose, pri čemu se broj `n` učitava na samom početku programa.

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

int main() {
   cout << "Unesi broj pravougaonika: ";
   int n;
   cin >> n;
   for (int i = 0; i < n; i++) {
       cout << "Unesi duzine stranica pravougaonika: ";
       int a, b;
       cin >> a >> b;
```

```
        cout << "Povrsina pravougaonika je: " << a * b <<
 ↪  endl;
    }
    return 0;
}
```

2.1.5.3 *Petlja `do-while`*

Pored petlje `while`, postoji i petlja `do-while`, koja joj nalikuje, ali se uslov petlje ispituje na kraju tela petlje. Dakle, u ovoj petlji uvek se telo izvršava barem jednom, bez obzira na to da li je uslov ispunjen ili ne (jer se on ispituje na kraju tela petlje). Na primer, naredni blok koda:

```
int i = a;
do {
    cout << i << endl;
    i++;
} while (i <= b);
```

ispravno ispisuje brojeve iz intervala [a, b] ako je $a \leq b$, ali ako je $a > b$, rešenje zasnovano na `for` i `while` petlji ne bi ispisivalo nijedan broj (što bismo i očekivali), dok bi navedeni blok koda ispisivao broj a.

2.1.6 *Definisanje funkcija*

Pored bibliotečkih funkcija koje su nam na raspolaganju, programski jezici, pa i jezik C++, programeru daju mogućnost da *definiše funkcije*, što doprinosi izbegavanju ponovljenih delova programa, dekompoziciji problema na manje potprobleme i boljoj organizaciji koda. Funkcije možemo zamisliti slično kao u matematici. One obično za nekoliko *ulaznih parametara* izračunavaju jednu *rezultujuću vrednost*. Na primer, naredna funkcija izračunava obim pravougaonika datih stranica.

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

// funkcija izracunava obim pravougaonika na osnovu poznatih
// duzina stranica a i b
double obim(double a, double b) {
    // formula za obim pravougaonika
    return 2*a + 2*b;
}
```

```
int main() {
    // ucitavamo duzine stranica pravougaonika
    double a, b;
    cin >> a >> b;
    // izracunavamo obim pomocu definisane funkcije
    double O = obim(a, b);
    // ispisujemo izracunati obim
    cout << O << endl;
}
```

U prethodnom kodu data je *definicija funkcije* `obim`. Prva linija, `double obim(double a, double b)`, govori da se funkcija zove `obim`, da prima dve ulazne vrednosti (dva *parametra*) koji su realni brojevi i nazivaju se `a` i `b` i da vraća rezultat koji je takođe realan broj. U telu funkcije (ono je ograničeno vitičastim zagradama) se opisuje kako se rezultat izračunava na osnovu ulaznih vrednosti. Kada je konačni rezultat izračunat, on se vraća na mesto poziva funkcije (u našem primeru to je funkcija `main`) ključnom rečju `return`. Ispravno definisane funkcije se mogu pozivati iz drugih funkcija. U prethodnom primeru poziv je izvršen u sklopu deklaracije `double O = obim(a, b)`. U pozivu se navode *argumenti* kojima se inicijalizuju parametri funkcije. U ovom slučaju to su vrednosti promenljivih `a` i `b` koje su učitane sa tastature. Argumenti mogu biti i konstantne vrednosti (parametri moraju biti promenljive). Na primer, dopušten je poziv `obim(5.0, 7.0)`.
Radi čitljivosti koda, poželjno je da ime funkcije oslikava ono što ona radi.
Unutar funkcije možemo deklarisati i koristiti i promenljive u kojima čuvamo međurezultate. Na primer, možemo definisati funkciju za izračunavanje površine jednakostraničnog trougla date dužine stranice.

```
double povrsinaJednakostranicnogTrougla(double a)
{
   // izracunavamo visinu trougla
   double h = a * sqrt(3) / 2.0;
   // izracunavamo povrsinu na osnovu duzine stranice i
    ↪ visine
   double P = a * h / 2.0;
   // vracamo konacan rezultat
   return P;
}
```

Ovom funkcijom je realizovano izračunavanje površine jednakostraničnog trougla i kad

god nam to zatreba u programu (a u nekom matematičkom zadatku to može biti potrebno više puta), možemo pozvati funkciju i dobiti željeni rezultat.
Funkcije ne moraju da vrate neku vrednost. Takve funkcije se nekada nazivaju *procedure* i jedini zadatak im je da proizvedu neki propratni efekat (na primer, da nešto ispišu na ekran). Kao povratni tip podataka navodi se `void`. Na primer, možemo definisati proceduru koja oko datog teksta ispisuje ukrasne linije, a onda je pozvati nekoliko puta u programu.

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

void ukrasiTekst(string tekst) {
    cout << "----------------------------------------" <<
 ↪  endl;
    cout << tekst << endl;
    cout << "----------------------------------------" <<
 ↪  endl;
}

int main() {
   ukrasiTekst("Dobar dan!");
   ukrasiTekst("Zdravo, svima!");
   ukrasiTekst("Dovidjenja!");
}
```

2.1.7 *Strukture podataka*

Računari skladište i obrađuju velike količine podataka. Veća količina podataka se čuva u obliku različitih *kolekcija* ili *struktura podataka*.
Jedna od osnovnih kolekcija je *statički niz*. Umesto korišćenja velikog broja pojedinačnih promenljivih, više podataka istog tipa možemo čuvati u jednom nizu. Na primer, podatke o ocenama nekog studenta možemo učitati na sledeći način.

```
int ocene[5];
for (int i = 0; i < 5; i++)
   cin >> ocene[i];
```

Deklaracija `int ocene[5];` obezbeđuje da se u memoriji odvoji prostor za smeštanje 5 podataka tipa `int`. Njima se pristupa kao `ocene[0]`, `ocene[1]`, `ocene[2]`, `ocene[3]`

i `ocene[4]` (indeksi kreću od 0). Kada se podaci jednom upišu u memoriju, mogu se više puta obrađivati. Na primer, možemo izračunati prosečnu ocenu (tako što izračunamo zbir ocena i podelimo ga sa 5).

```
int zbir_ocena = 0;
for (int i = 0; i < 5; i++)
    zbir_ocena = zbir_ocena + ocene[i];
double prosek = zbir_ocena / 5.0;
cout << prosek << endl;
```

Niz možemo inicijalizovati tj. popuniti podacima prilikom deklaracije (tada ne moramo navoditi broj elemenata).

```
int ocene[] = {3, 5, 2, 5, 1};
```

Broj elemenata statičkog niza je određen tokom pisanja programa i niz se ne može proširivati tokom rada programa. Ako broj ocena ne znamo unapred, umesto statičkog niza možemo koristiti vektor tj. strukturu `vector`. U narednom programu se deklariše vektor koji je u početku prazan, a zatim se učitava 5 ocena i jedna po jedna se dodaje u vektor (pozivom `push_back`).

```
vector<int> ocene;
for (int i = 0; i < 5; i++) {
   int ocena;
   cin >> ocena;
   ocene.push_back(ocena);
}
```

Primetimo da se `push_back` ne poziva kao funkcije koje smo do sada koristili (na primer, `sqrt(x)` ili `getline(cin, s)`): navodi se iza imena promenljive i tačke, u obliku `ocene.push_back(ocena)`, i deluje upravo na taj vektor. Ovakve funkcije, koje su vezane za objekte određenog tipa (ovde vektora), nazivaju se *metode*. I `find` i `end`, koje ćemo uskoro koristiti u radu sa mapama, takođe su metode. Mnoge funkcije standardne biblioteke koje ćemo koristiti zapravo su metode klasa koje su definisane u standardnoj biblioteci (na primer, klasa `string`, `vector` i `map`), dok su druge (na primer, `sqrt`, `getline` ili `sort`) obične funkcije. Više o razlici između funkcija i metoda biće reči u poglavljima 7.5 i 8.1.3.
U okviru deklaracije vektor može biti inicijalizovan:

```
vector<int> ocene = {3, 5, 2, 5, 1};
```

Kada je vektor popunjen, elementi se (koristeći indeksni pristup) obrađuju na potpuno isti način kao i elementi statičkog niza (na primer, deo programa koji računa prosečnu ocenu bi se mogao upotrebiti u neizmenjenom obliku).
Elementima niza i vektora se pristupa na osnovu numeričkih indeksa (pozicija tj. rednih brojeva). U nekim situacijama poželjno je podacima pristupati na osnovu tzv. ključeva. Na primer, želimo da upamtimo ocene iz različitih predmeta tako da svakoj oceni možemo pristupiti na osnovu naziva predmeta. Za to je moguće koristiti preslikavanja (ona se nekada nazivaju i mape, rečnici ili asocijativni nizovi). U narednom primeru ključevi su nazivi predmeta, a ocene su vrednosti koje su pridružene tim ključevima.

```
map<string, int> ocene;

// upisujemo podatke u mapu
ocene["programiranje"] = 9;
ocene["analiza 1"] = 7;
ocene["algebra"] = 8;

// citamo podatke iz mape
cout << ocene["algebra"] << endl;
```

I preslikavanje možemo inicijalizovati tokom njegove deklaracije:

```
map<string, int> ocene =
  {{"programiranje", 9}, {"analiza 1", 7}, {"algebra", 8}};
```

Ako pokušamo da pristupimo podatku na osnovu ključa koji ne postoji u mapi, onda se on automatski dodaje u mapu. Proveru da li ključ postoji u mapi možemo izvršiti na sledeći način:

```
string predmet;
cout << "Unesi naziv predmeta: " << endl;
getline(cin, predmet);
if (ocene.find(predmet) != ocene.end())
   cout << ocene[predmet] << endl;
else
   cout << "Ne postoji ocena iz tog predmeta" << endl;
```

Pozivom `find` se traži ključ `predmet` u mapi `ocene` i ako se tokom te pretrage dođe do kraja mape, a ključ se ne nađe, to znači da ključ ne postoji u mapi (kraj mape `ocene` se dobija pozivom `ocene.end()`).
Sve elemente mape `ocene` možemo obrađivati na sledeći način:

```
for (auto [predmet, ocena] : ocene)
    cout << predmet << ": " << ocena << endl;
```

Prethodnim kodom se ispisuju svi elementi mape (predmeti i ocene iz tih predmeta). Ključnom rečju `auto` kompilatoru se nalaže da samostalno odredi tipove promenljivih `predmet` i `ocena`.
Nekada želimo da povežemo više podataka koji su logički povezani u jednu celinu, da bismo olakšali rad sa tim podacima. Na primer, ako razmatramo podatke o studentima, za svakog studenta možemo pamtiti ime, prezime i prosečnu ocenu. Umesto da ove podatke držimo u zasebnim promenljivim, što bi moglo biti nepraktično i lakše vodilo ka greškama, možemo definisati novi tip podataka (strukturu) koji obuhvata sva tri podatka i omogućava lakše upravljanje njima kao jednom celinom:

```
struct student {
   string ime;
   string prezime;
   double prosek;
};
```

Koristeći strukture, podaci se grupišu na način koji bolje odražava njihovu prirodnu povezanost. Na ovaj način možemo lako definisati promenljive koje predstavljaju studente, sa svim njihovim relevantnim podacima na jednom mestu, u jednoj promenljivoj strukturnog tipa. Na primer:[1]

```
student petar;
petar.ime = "Petar";
petar.prezime = "Petrovic";
petar.prosek = 7.52;
```

Strukture omogućavaju jednostavnu i intuitivnu inicijalizaciju podataka, koristeći vitičaste zagrade:

[1] U jeziku C, za razliku od jezika C++, u deklaraciji promenljive potrebno je navoditi ključnu reč `struct`, na primer: `struct student petar`.

```
student petar = {"Petar", "Petrovic", 7.52};
```

Jedna od prednosti korišćenja struktura je i mogućnost lakog definisanja niza ovih objekata, što olakšava rad sa većim brojem podataka. Na primer, možemo definisati niz studenata i odmah ih inicijalizovati:

```
student studenti[] = {
  {"Ana", "Anic", 9.83},
  {"Petar", "Petrovic", 7.52}
};
```

Ako ne bismo koristili strukture, morali bismo da podatke čuvamo u tri odvojena niza (u jednom bismo čuvali imena, u drugom prezimena i u trećem prosečne ocene). Organizacija podataka korišćenjem struktura čini naš kôd čitljivijim, jednostavnijim za održavanje i manje sklonim greškama, naročito kada radimo sa velikim brojem podataka koji su logički povezani.

2.1.8 *Upravljanje izuzecima i greškama*

U jeziku C++ postoji mehanizam za upravljanje greškama – mehanizam izuzetaka (engl. exceptions). Programer izdaje posebnu naredbu (`throw`) koja se aktivira u slučaju greške, kojom se prekida kôd koji se trenutno izvršava i tok programa preusmerava se na poseban deo koda koji se bavi obradom grešaka (`try-catch`). Time se postiže da su normalan tok programa i obrada grešaka fizički razdvojeni u samom kodu, što pojednostavljuje programiranje i čini programe čitljivijim i lakšim za održavanje. Mehanizam izuzetaka koristi se i u implementaciji funkcija iz standardne biblioteke. Korisnik tih funkcija ne mora da misli o načinu aktiviranja izuzetka (i o naredbi `throw`), nego samo o detektovanju i upravljanju izuzetkom (naredbe `try-catch`). Naredni primer ilustruje kako bi mogao da se obradi izuzetak koji se aktivira pri pokušaju kreiranja vektora koji je prevelik.

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  try {
    vector<int> veliki_vektor(1000000000000);
  } catch (const bad_alloc& e) {
```

```
    cerr << "Neuspesna inicijalizacija vektora: " << e.what()
↪   << endl;
  }
  return 0;
}
```

Ne aktivira izuzetak nužno svaka greška u fazi izvršavanja. Na primer, izuzetak se aktivira pri pokušaju kreiranja prevelikog vektora, ali se ne aktivira pri pristupu nepostojećem elementu vektora. Takav pristup dovodi do nedefinisanog ponašanja: program se može prekinuti (na primer, uz poruku *segmentation fault*), ali može i da nastavi rad sa pogrešnim vrednostima.
Više reči o upravljanju greškama i obradi izuzetaka biće u poglavlju 10.9.

3. Promenljive i tipovi

U ovom poglavlju ćemo detaljnije proučiti sledeće elemente programskog jezika C++:

- *Promenljive i konstante,* koje su osnovni oblici podataka kojima se operiše u programu.
- *Tipovi*, koji određuju vrstu podataka, način reprezentacije i skup vrednosti koje promenljive, konstante i izrazi mogu imati, kao i skup operacija koje se nad tim podacima mogu primeniti.
- *Deklaracije,* koje uvode spisak promenljivih koje će se koristiti, određuju kog su tipa i, eventualno, koje su im početne vrednosti.

3.1 Promenljive, konstante i deklaracije

Promenljive su osnovni objekti koji se koriste u programima. Svakoj promenljivoj je pridružen neki prostor u memoriji (možemo ga zamišljati kao kutijicu ili kao kućicu) i u svakom trenutku svog postojanja ima vrednost kojoj se može pristupiti — koja se može pročitati i koristiti, ali i koja se (ukoliko nije traženo drugačije) može menjati. Primetimo da je ovo sasvim različito od promenljivih u matematici koje označavaju neke veličine i koje se ne menjaju tokom vremena[1].

Imena promenljivih (ali i funkcija, struktura, itd.) zadaju se *identifikatorima.* U prethodnim programima korišćene su promenljive čija su imena `a`, `i`, `ime`, `brGodina`, `zbir_ocena` itd. Generalno, identifikator može da sadrži slova i cifre, kao i simbol `_` (koji je pogodan za duga imena), ali identifikator ne može počinjati cifrom. Dodatno, ključne reči jezika (na primer, `if`, `for`, `while`) ne mogu se koristiti kao identifikatori. U identifikatorima, velika i mala slova se razlikuju. Na primer, promenljive sa imenima `a` i `A` se tretiraju kao dve različite promenljive.

[1] Postoje funkcionalni programski jezici, na primer, Haskell, koji u cilju lakšeg rezonovanja o ispravnosti programa teže da zadrže tesne veze sa matematikom i ne dopuštaju mogućnost izmene vrednosti promenljive.

Imena promenljivih i imena funkcija, u principu, treba da oslikavaju njihovo značenje i ulogu u programu, ali za promenljive kao što su indeksi u petljama se obično koriste kratka, jednoslovna imena (na primer `i`). Ako ime promenljive sadrži više reči, onda se, radi bolje čitljivosti, te reči razdvajaju simbolom _ (na primer, `broj_studenata`) ili početnim velikim slovima (na primer, `brojStudenata`) — ovo drugo je takozvana kamilja notacija (CamelCase). Postoje različite konvencije za imenovanje promenljivih. Iako je dozvoljeno, ne preporučuje se korišćenje identifikatora koji počinju simbolom _, jer se oni obično koriste za sistemske funkcije i promenljive.
Kao što je rečeno, jezik C++ je statički tipiziran jezik, što znači da je svakoj promenljivoj (ali i konstanti i izrazu) pridružen jedinstven tip i tip promenljive ne može da se promeni tokom izvršavanja programa. Na osnovu tipa promenljive, kompilator, između ostalog, određuje i količinu memorije potrebnu za smeštanje te promenljive.
Sve promenljive moraju biti deklarisane pre korišćenja. Deklaracija sadrži tip i listu od jedne ili više promenljivih tog tipa, razdvojenih zarezima.

```
int broj;  // deklaracija jedne promenljive
int a, b;  // deklaracija dve promenljive
```

U opštem slučaju nije propisano koju vrednost ima promenljiva neposredno nakon što je deklarisana. Prilikom deklaracije može se izvršiti početna inicijalizacija. Moguće je kombinovati deklaracije sa i bez inicijalizacije.

```
int vrednost = 5;    // deklaracija sa inicijalizacijom
int a = 3, b, c = 5; // deklaracije sa inicijalizacijom i bez
 ↪  inicijalizacije
```

Izraz kojim se promenljiva inicijalizuje zvaćemo inicijalizator.
Kvalifikator `const` može biti dodat deklaraciji promenljive da bi naznačio i obezbedio da se njena vrednost neće menjati, na primer:

```
// ovu promenljivu nije moguce menjati
const double GRAVITY = 9.81;
```

Vrednost tipa `const T` (gde je `T` bilo koji tip, na primer — `int`) može biti dodeljena promenljivoj tipa `T`, ali promenljivoj tipa `const T` ne može biti dodeljena vrednost (osim prilikom inicijalizacije) — pokušaj menjanja vrednosti konstantne promenljive (kao i svakog drugog konstantnog sadržaja) dovodi do greške prilikom prevođenja programa.
Deklaracije promenljivih mogu se navoditi na različitim mestima u programu. Ukoliko je promenljiva deklarisana u nekoj funkciji (na primer, u funkciji `main`), onda kažemo

da je ona *lokalna* za tu funkciju i druge funkcije ne mogu da je koriste. Različite funkcije mogu imati lokalne promenljive istog imena. Promenljive deklarisane van svih funkcija su *globalne* i mogu se koristiti u više funkcija. Vidljivost tj. oblast važenja identifikatora (i njima uvedenih promenljivih) određena je pravilima *dosega identifikatora*, o čemu će više reči biti u poglavlju 7.11.1. Rane verzije programskog jezika C su zahtevale da se sve lokalne promenljive deklarišu na početku tela funkcije. Međutim, u jeziku C++ je uobičajeno i poželjno da se deklaracije navode neposredno pre prve upotrebe promenljive (dakle, prvi put kada nam zatreba) i to u što užem dosegu (na primer, ako se neka promenljiva koristi samo u telu neke petlje ograđenom vitičastim zagradama, najbolje je uvesti je unutar te petlje tj. unutar tih vitičastih zagrada).

Konstante (kaže se nekada i *literali*) su fiksne vrednosti kao, na primer, `0`, `2`, `2007`, `3.5`, `1.4e2`, `'a'` ili `"zdravo"`. Ista vrednost se ponekad može predstaviti različitim konstantama. Za razliku od promenljivih, konstante se ne deklarišu. Međutim, pravilima jezika je za svaku konstantu jednoznačno određen njen tip. To je važno, jer od tipova konstanti zavisi koje operacije je moguće primeniti nad njima, a zavisi i vrednost složenog izraza u kojem figuriše konstanta (na primer, vrednost izraza `5 / 2` je `2`, a vrednost izraza `5.0 / 2` je `2.5`).

3.2 Osnovni tipovi podataka

Kao i u većini drugih programskih jezika, u jeziku C++ podacima (tj. izrazima kojima se podaci predstavljaju) su pridruženi *tipovi*. Jedan tip karakteriše: vrsta podataka koje opisuje, način interne binarne reprezentacije tih podataka u memoriji računara (jer se svi podaci zapisuju binarno) i skup operacija koje se mogu primeniti nad podacima tog tipa.

Standard jezika C++ ne propisuje jednoznačno način interne reprezentacije, čak ni za osnovne tipove podataka, čime se ostavlja sloboda da se neki tipovi različito predstave na različitim sistemima tj. platformama.[2] Na primer, broj bitova za predstavljanje nekog tipa može biti manji na nekom namenskom, ugrađenom računaru koji ima manje memorije, nego na klasičnom računaru opšte namene. Takođe, između računara sa 32-bitnom arhitekturom i 64-bitnom arhitekturom često postoje razlike u predstavljanju podataka. Ovo je veoma važno imati na umu kada se pišu programi za koje želimo da budu prenosivi između različitih sistema. Ipak, ova knjiga predstavlja uvod u programiranje za početnike, tako da ćemo se, jednostavnosti radi, ograničiti na tipične reprezentacije na klasičnim PC računarima i na mnogim mestima nećemo analizirati probleme do kojih može doći usled razlika u predstavljanju tipova podataka na različitim platformama. Detaljima reprezentacije tipova podataka ćemo se vratiti ponovo u narednim tomovima ove knjige.

U nastavku će biti opisani osnovni tipovi podataka.

[2] Pod sistemom podrazumevamo hardver računara, operativni sistem i prevodilac koji se koristi.

3.2.1 *Celobrojni tipovi*

Osnovni celobrojni tip podataka u jeziku C++ je tip `int` (od engleskog integer, ceo broj). On se najčešće predstavlja pomoću 4 bajta (tj. 32 bita), na način koji omogućava predstavljanje i nenegativnih i negativnih vrednosti. Precizni opsezi vrednosti koje se mogu reprezentovati raznim tipovima dati su u tabeli 3.1, međutim, za početak je dovoljno steći osećaj da tip `int` najčešće dopušta reprezentovanje (svih) vrednosti u intervalu od oko minus dve do oko plus dve milijarde.

Konačan opseg tipova treba uvek imati u vidu jer iz ovog razloga neke matematičke operacije neće dati očekivane vrednosti tj. vrednosti koje su rezultat za brojeve koji nisu ograničeni. Tada kažemo da dolazi do *prekoračenja*. Na primer, ako promenljiva `x` ima tip `unsigned char` i vrednost `255`, i ako bude uvećana za `3`, njena vrednost neće biti `258` nego `2`. Kod neoznačenih tipova (o kojima će biti reči u nastavku) ponašanje pri prekoračenju je precizno definisano: računa se po modulu 2^n, gde je n broj bitova tipa. Kod označenih tipova, poput tipa `int`, prekoračenje dovodi do nedefinisanog ponašanja programa. Iako to nije garantovano standardom, ako je tip `int` predstavljen sa 32 bita, naredni program obično štampa negativan rezultat.

```
int a = 2000000000, b = 2000000000;
cout << "Zbir brojeva " << a << " i "  << b << " je " << a
↪  + b << endl;
```

U nekim programima znamo da će brojevi sa kojima baratamo biti mali, u nekim da će biti veliki, pa programski jezik C++ daje programeru mogućnost da preciznije označi (kvalifikuje) celobrojni tip koji želi da upotrebi za smeštanje nekog podatka. Tipu `int` mogu biti pridruženi kvalifikatori `short`, `long` i `long long`. Ime tipa `short int` može se kraće zapisati sa `short`, ime tipa `long int` može se kraće zapisati sa `long`, a ime tipa `long long int` može se kraće zapisati sa `long long`. Veličine ovih tipova nisu precizno određene standardom. Tip `short` obično zauzima 2 bajta (tj. 16 bita), što daje opseg vrednosti od oko minus 30 hiljada do oko plus 30 hiljada, tip `long` može da se poklapa sa tipom `int` (4 bajta, na primer, na sistemu Windows) ili sa tipom `long long` (8 bajtova, na primer, na 64-bitnim sistemima GNU/Linux), dok tip `long long` obično zauzima 8 bajtova (tj. 64 bita), što daje opseg reda veličine milijardu milijardi (od oko $-9{,}2 \cdot 10^{18}$ do oko $9{,}2 \cdot 10^{18}$). U tabeli 3.1 za tip `long` navedene su vrednosti za 64-bitne sisteme GNU/Linux.

Bilo kom celobrojnom tipu može biti pridružen kvalifikator `signed` ili `unsigned`. Kvalifikator `signed` se obično podrazumeva, pa ne utiče na promenu reprezentacije i opsega tipa. Sa druge strane, kvalifikator `unsigned` označava da se broj tretira kao neoznačen, tj. da se pomoću tog tipa predstavljaju samo nenegativni brojevi. Time se mogu predstaviti dvostruko veće pozitivne vrednosti (na primer, `unsigned int` ima opseg od 0 do oko 4 milijarde), što u nekim situacijama može biti korisno. Međutim, osnovna motivacija za korišćenje neoznačenih tipova će nam češće biti da se čitaocu naglasi da se

za neki podatak očekuje da je nenegativan ceo broj. Na primer, broj elemenata vektora dobijen metodom `size` je neoznačenog tipa.

Tabela 3.1: Najčešći opseg celobrojnih tipova

	označeni (signed)	neoznačeni (unsigned)
karakteri (char)	1B = 8b $[-2^7, 2^7 - 1] = [-128, 127]$	1B = 8b $[0, 2^8 - 1] = [0, 255]$
kratki (short int)	2B = 16b $[-32\text{K}, 32\text{K} - 1] = [-2^{15}, 2^{15} - 1]$ $= [-32768, 32767]$	2B = 16b $[0, 64\text{K} - 1] = [0, 2^{16} - 1]$ $= [0, 65535]$
int	4B = 32b $[-2\text{G}, 2\text{G} - 1] = [-2^{31}, 2^{31} - 1]$ $= [-2147483648, 2147483647]$	4B = 32b $[0, 4\text{G} - 1] = [0, 2^{32} - 1]$ $= [0, 4294967295]$
dugi (long int) (64-bitni Linux)	8B = 64b $[-2^{63}, 2^{63} - 1]$ $\approx [-9{,}2 \cdot 10^{18}, 9{,}2 \cdot 10^{18}]$	8B = 64b $[0, 2^{64} - 1]$ $\approx [0, 1{,}84 \cdot 10^{19}]$
veoma dugi (long long int) od C++11	8B = 64b $[-2^{63}, 2^{63} - 1]$ $\approx [-9{,}2 \cdot 10^{18}, 9{,}2 \cdot 10^{18}]$	8B = 64b $[0, 2^{64} - 1]$ $\approx [0, 1{,}84 \cdot 10^{19}]$

Postoje i tipovi za koje je standardom precizno definisan broj bitova (i on je isti na svim sistemima). To su, na primer, tipovi `int8_t`, `int16_t`, `int32_t`, `int64_t`, kao i neoznačeni tipovi `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, za čije je korišćenje potrebno uključiti zaglavlje `<cstdint>`.

Sasvim očekivano, celobrojne dekadne konstante kao što su `123` ili `45678` su tipa `int`. Ako su vrednosti prevelike da bi se mogle predstaviti tipom `int`, konstante su nekog tipa koji ima širi opseg (na primer, `long long`). Standard definiše precizna pravila koja određuju tip svake konstante (ali ih nećemo ovde navoditi). Programer može posebnim sufiksima da precizira tip konstante (na primer, vrednost `5ull` je tipa `unsigned long long`).

Osim u dekadnom, celobrojne konstante mogu biti zapisane i u oktalnom i u heksadekadnom sistemu. Zapis konstante u oktalnom sistemu počinje cifrom `0`, a zapis konstante u heksadekadnom sistemu počinje simbolima `0x` ili `0X`. Na primer, broj `31` se može u programu zapisati na sledeće načine: `31` (dekadni zapis), `037` (oktalni zapis), `0x1f` ili `0X1F` (heksadekadni zapis).

Negativne konstante se dobijaju unarnim operatorom `-`, na primer `-123`. Slično, može se koristiti i operator plus, ali `+123` je isto kao i `123`.

3.2.2 *Realni tipovi*

Realne brojeve ili, preciznije, brojeve u pokretnom zarezu opisuju tipovi `float`, `double` i `long double`. Tip `float` opisuje brojeve u pokretnom zarezu *osnovne tačnosti*, tip `double` opisuje brojeve u pokretnom zarezu *dvostruke tačnosti*, a tip `long double` brojeve u pokretnom zarezu *proširene tačnosti*. Broj bitova i način reprezentacije ovih tipova nije precizno definisan standardom. Uobičajeno je da se u savremenim računarima najčešće zapisuju u skladu sa standardom IEEE 754. Tip `float` obično zauzima 4 bajta (tj. 32 bita), što daje preciznost od oko 7 značajnih cifara. Tip `double` obično zauzima 8 bajtova (tj. 64 bita), što daje preciznost od oko 15 značajnih cifara. Tip `long double` često se poklapa sa tipom `double`, dok na nekim sistemima zauzima 10 bajtova (tj. 80 bitova), što daje preciznost od oko 18–19 značajnih cifara. Na primer, broj 12,34 se pomoću tipa `float` zapisuje dovoljno tačno da se sve njegove značajne cifre sačuvaju (zapisuje se vrednost 12,3400001 ...), dok broj 12345,6789 ima 9 značajnih cifara, pa se sve one sačuvaju tek pomoću tipa `double` (u tipu `float` zapisuje se vrednost 12345,6787 ..., tj. poslednje cifre se gube). Primetimo da se nijedan od ova dva broja ne može zapisati egzaktno korišćenjem uobičajenih konvencija, kao što je standard IEEE 754. Egzaktno se mogu zapisati samo brojevi koji imaju konačan (i dovoljno kratak) zapis u osnovi 2, tj. koji se mogu predstaviti kao razlomak čiji je imenilac stepen dvojke, na primer 0,5 = 1/2, 0,375 = 3/8 ili 12,75 = 51/4. Broj 12,34 nije takav, pa se mora zaokružiti.

Raspon realnih tipova je značajno veći od celobrojnih (pregled je dat u tabeli 3.2), međutim, zbog ograničenog broja značajnih cifara, velike vrednosti se zaokrugljuju i zapisuju se neprecizno. Na primer, broj `123456789` se u okviru tipa `float` ne može zapisati egzaktno i umesto njega se zapisuje najbliža vrednost koja se može predstaviti, a to je `123456792` (koja se podrazumevano ispisuje kao `1.23457e+08`). Dakle, iako su šireg raspona, realni tipovi ne mogu zapisati sve vrednosti koje se mogu zapisati u okviru celobrojnih tipova.

Tabela 3.2: Najčešći opseg realnih tipova

Tip	Veličina	Raspon	Preciznost
float	4B = 32b	±1.17549e-38 do ±3.40282e+38	oko 7 značajnih cifara
double	8B = 64b	±2.22507e-308 do ±1.79769e+308	oko 15 značajnih cifara

Zbog načina zapisa ne mogu se sve dekadne vrednosti precizno zapisati pomoću ovih zapisa i jako je važno da programer razume da kada se koriste ovi tipovi nije nužno da će vrednosti biti tačno zapisane. Na primer, vrednost $\frac{1}{10}$ se zapisuje kao decimalni broj 0,1, međutim, ta se vrednost ne može tačno zapisati ni u jednom od nabrojanih tipova (jer 1/10 nema imenilac koji je stepen dvojke). Zato, na primer, veoma jednostavna provera da li je `0.1 + 0.2 == 0.3` daje vrednost *netačno*. U programiranju sa

realnim brojevima treba zato biti veoma pažljiv i izbegavati bilo kakve provere koje se oslanjaju na preciznost zapisa. Na primer, ne treba nikada proveravati da li su dva broja u pokretnom zarezu jednaka, već samo da li im je vrednost dovoljno bliska.

Standard IEEE 754 uključuje i mogućnost zapisa specijalnih vrednosti kao što su $+\infty$, $-\infty$ i NaN. Na primer, vrednost izraza $1.0/0.0$ je $+\infty$, za razliku od celobrojnog izraza $1/0$ čije izračunavanje obično dovodi do greške u fazi izvršavanja programa (engl. division by zero error). Vrednost izraza $0.0/0.0$ je NaN, tj. *not-a-number* i ta specijalna vrednost se koristi da označi matematički nedefinisane vrednosti (npr. $\infty - \infty$ ili koren ili logaritam negativnog broja). Specijalne vrednosti dalje mogu da učestvuju u izrazima. Na primer, ako se na izraz čija je vrednost $+\infty$ doda neka konstantna vrednost, dobija se opet vrednost $+\infty$, ali vrednost izraza $1.0/+\infty$ jednaka je 0.0. S druge strane, svi izrazi u kojima učestvuje vrednost NaN ponovo imaju vrednost NaN. Vrednost ∞ tipa `double` se u programu može zapisati izrazom `numeric_limits<double>::infinity()` (za čije je korišćenje potrebno uključiti zaglavlje `<limits>`).

Najveći broj funkcija iz standardne biblioteke (pre svega matematičke funkcije definisane u zaglavlju `<cmath>`) koristi tip podataka `double`. Tip `float` se u programima koristi uglavnom zbog uštede memorije ili vremena na računarima na kojima je izvođenje operacija u dvostrukoj tačnosti veoma skupo (u današnje vreme, međutim, većina računara podržava efikasnu manipulaciju brojevima zapisanim u dvostrukoj tačnosti).

Konstante realnih brojeva ili, preciznije, konstantni brojevi u pokretnom zarezu sadrže decimalnu tačku (na primer, `123.4`) ili eksponent (`5e-2`, što označava vrednost $5 \cdot 10^{-2}$) ili i jedno i drugo. Vrednosti ispred i iza decimalne tačke mogu biti izostavljene (ali ne istovremeno). Na primer, ispravne konstante su i `.4` ili `5.` (ali ne i `.`). Negativne konstante dobijaju se unarnim operatorom `-` (unarni operator `+` ne proizvodi nikakav efekat). Tip svih ovih konstanti je `double`, osim ako se na kraju zapisa ne doda neki od sufiksa (na primer, konstanta `12.3f` je tipa `float`, a `12.3l` tipa `long double`).

3.2.3 Logički tip

Za predstavljanje logičkih (istinitosnih) vrednosti koristi se tip `bool` koji ima samo dve vrednosti (`true` označava *tačno*, a `false` *netačno*). Podaci ovog tipa se najčešće koriste u uslovima grananja i petlji, i dobijaju se i kombinuju primenom relacijskih i logičkih operatora, pa će više reči o ovome biti u poglavlju 5.

3.2.4 Karakterski tip

Osnovna namena tipa `char` je predstavljanje kodova karaktera (najčešće u tabeli ASCII). Podaci tog tipa zauzimaju 1 bajt (tj. 8 bita). Pošto se u njemu beleže numeričke vrednosti (mali brojevi iz intervala od -128 do 127 ili od 0 do 255 u zavisnosti od toga da li se koristi označena ili neoznačena varijanta ovog tipa), moguće je ovaj tip koristiti i za predstavljanje malih brojevnih vrednosti (u cilju uštede memorije), međutim, u ovoj knjizi to nećemo raditi.

Direktno specifikovanje karaktera korišćenjem numeričkih kodova nije preporučljivo. Umesto toga, preporučuje se korišćenje karakterskih konstanti. Karakterske konstante u programskom jeziku C++ se navode između apostrofa (`''`). Vrednost date konstante je numerička vrednost datog karaktera u korišćenoj karakterskoj tabeli (na primer, ASCII). Na primer, u ASCII kodiranju, karakterska konstanta `'0'` predstavlja vrednost 48 (koja nema veze sa numeričkom vrednošću 0), `'A'` je karakterska konstanta čija je vrednost u ASCII tabeli 65, `'a'` je karakterska konstanta čija je vrednost u ASCII tabeli 97.
Tipom `char` predstavljaju se pojedinačni karakteri (i konstante tog tipa se navode između apostrofa `''`), a tipom `string` niske karaktera (i konstante tog tipa se navode između dvostrukih navodnika `""`).
Specijalni karakteri se mogu navesti korišćenjem specijalnih sekvenci karaktera koje počinju karakterom `\` (engl. escape sequences). Na primer, karakter `'\n'` označava prelazak u novi red, a `'\t'` tabulator. Karakterska konstanta `'\0'` predstavlja karakter čija je vrednost nula. Ovaj karakter ima specijalnu ulogu u programskom jeziku C jer se koristi za označavanje kraja niske karaktera.
Pošto se karakterske konstante identifikuju sa njihovim numeričkim vrednostima, one mogu ravnopravno da učestvuju u aritmetičkim izrazima. Na primer, na ASCII sistemima (tj. na sistemima na kojima se koristi ASCII tabela karaktera), izraz `'0' <= c && c <= '9'` proverava da li karakterska promenljiva `c` sadrži ASCII kôd neke cifre, dok se izrazom `c - '0'` dobija numerička vrednost cifre čiji je ASCII kôd sadržan u promenljivoj `c`. Ipak, za rad sa pojedinačnim karakterima preporučuju se bibliotečke funkcije iz zaglavlja `<cctype>` (koje su opisane u poglavlju 9).

3.2.5 *Niske*

Niske se u jeziku C++ predstavljaju tipom `string`. Za njegovo korišćenje potrebno je direktivom `#include` uključiti zaglavlje `<string>`. Za razliku od do sada pomenutih elementarnih tipova, niske su složen tip koji sadrži niz karaktera kojima je zapisan tekst. Svaka niska je objekat klase `string`. Stoga su kreiranje niski i operacije nad njima donekle sporije nego nad osnovnim brojevnim tipovima.
Konstantne niske se navode između navodnika `"..."` i mogu se dodeljivati promenljivim tipa `string`.

```
string pozdrav = "Dobar dan!";
```

Tip `string` čuva tekst kao niz bajtova. U njemu se može čuvati i tekst koji sadrži slova van ASCII tabele (na primer, „č“ ili „ž“), ali se u danas najčešće korišćenom kodiranju UTF-8 takva slova zapisuju pomoću više bajtova. Zato tada metoda `size` ne vraća broj slova, već broj bajtova (na primer, za nisku `"čaša"` ona vraća 6), a pojedinačni elementi niske nisu nužno cela slova. U ovom udžbeniku ćemo u niskama obično koristiti samo karaktere ASCII tabele.

Broj bajtova od kojih se niska sastoji (a time i broj karaktera, ako su svi iz ASCII tabele) može se pročitati metodom `length` ili metodom `size` (obe daju isti rezultat).

```
cout << pozdrav.size() << endl;  // ispisuje 10
```

Kao što smo već videli, `size()` je *metoda* klase `string`, tj. funkcija članica te klase koja se poziva za konkretan objekat (ovde `pozdrav`), u obliku `pozdrav.size()`. Metode se razlikuju od „običnih“ funkcija, koje ne pripadaju nijednoj klasi ili strukturi i ne primenjuju se *na* neki objekat.
Niske je moguće nadovezivati operatorom +. Prilikom nadovezivanja bar jedan od dva operanda svakog operatora + mora biti objekat tipa `string`, dok drugi može biti i konstantna niska. Tako su ispravni izrazi `ime + " "` i `"Petar " + prezime`, ali nije ispravan izraz `"Petar" + " " + prezime`, jer se prvo nadovezuju dve konstantne niske.

```
string ime = "Petar";
string prezime = "Petrovic";
string ime_i_prezime = ime + " " + prezime;
cout << ime_i_prezime << endl; // ispisuje Petar Petrovic
```

Slično kao kod nizova i vektora, pojedinačnim karakterima niske možemo pristupiti korišćenjem indeksnog pristupa (operator `[]`). Pozicije se broje od nule. Na primer, izraz `ime[0]` ima vrednost `P`, dok izraz `ime[10]` ima nedefinisanu vrednost i može dovesti do prekida rada programa (jer niska ima 5 karaktera i dozvoljene pozicije su od 0 do 4).
Postoji veliki broj bibliotečkih metoda i funkcija za rad sa niskama. Na primer, metoda `substr` gradi novu nisku dobijenu izdvajanjem dela niske. Na primer, `ime.substr(1, 2)` vraća nisku `et` – prvi argument, 1, predstavlja poziciju početka, a drugi, 2, broj karaktera koji se izdvajaju. Ako se drugi argument izostavi, izdvajaju se karakteri do kraja niske. Na primer, vrednost izraza `ime.substr(1)` je `etar`.
Detaljniji pregled bibliotečke podrške za rad sa niskama dat je u poglavlju 9.
Jezik C++ nasleđuje mogućnost predstavljanja niski i pomoću nizova karaktera kojima je na kraju upisan specijalni karakter čiji je ASCII kôd 0 (engl. null-terminated strings). Na primer,

```
char ime[] = "Petar";
```

Za rad sa takvim niskama na raspolaganju imamo funkcije iz zaglavlja `<cstring>` (na primer, funkcije `strcmp`, `strcpy` i slično). Međutim, taj način je potpuno neprimeren savremenom jeziku C++, podložan je greškama i u ovoj knjizi ga nećemo objašnjavati.

Sa druge strane, ovakvi detalji interne reprezentacije podataka važni su za implementaciju visoko optimizovanih programa i programskih biblioteka (na primer, u implementaciji tipa `string` koriste se niske terminisane nulom), pa ćemo se njima posvetiti u narednim tomovima.

3.3 Dodele vrednosti promenljivoj

Inicijalna vrednost se promenljivoj može zadati u sklopu deklaracije.

```
int x = 42; // deklaracija sa inicijalizacijom
```

Deklarisane promenljive koje nisu inicijalizovane imaju obično[3] nedefinisanu vrednost i ne bi ih trebalo koristiti pre nego što im se dodeli početna vrednost.
Imperativno (kao i objektno-orijentisano) programiranje, koje je dominantni stil programiranja u ovoj knjizi, omogućava da se vrednost promenljive menja tokom izvršavanja programa. Ova, možda naizgled jednostavna, opaska zapravo je ključna karakteristika imperativnog stila programiranja i skoro svi netrivijalni programi se intenzivno zasnivaju na njoj.

3.3.1 Operator dodele

U bilo kom trenutku izvršavanja programa promenljivoj (koja nije označena kvalifikatorom `const`) možemo izmeniti vrednost korišćenjem operatora dodele (operatora `=`). Na primer,

```
x = 43;  // menjamo vrednost promenljive x
```

Pored vrednosti promenljivih, dodelom najčešće menjamo elemente nizova, vektora, niski i sličnih kolekcija (na primer, ako je `a` niz celih brojeva, naredbom `a[0] = 42;` se na početno mesto tog niza upisuje vrednost 42).[4]
Sa desne strane operatora dodele može se naći proizvoljan izraz, međutim, tip tog izraza mora ili biti jednak tipu leve strane ili mora biti takav da je moguća implicitna konverzija u taj tip. Na primer, inicijalizacija u okviru deklaracije

```
int x = 3.8;
```

[3] Postoje situacije u kojima standard garantuje inicijalnu vrednost promenljivih. Na primer, globalne promenljive, koje su deklarisane van svih funkcija, imaju garantovano inicijalnu vrednost 0.

[4] Standard definiše pojam izmenljive L-vrednosti (engl. l-value) i jedino se te vrednosti mogu naći sa leve strane operatora dodele.

je moguća jer postoji implicitna konverzija tipa `double` (što je tip konstantne vrednosti `3.8`) u tip `int`, pri čemu se tom konverzijom dobija celobrojna vrednost `3`.
U izrazu sa desne strane može učestvovati i stara vrednost promenljive kojoj se dodeljuje vrednost. Na primer,

```
x = x + 2; // vrednost promenljive se uvecava za 2
```

Ovo znači da se promenljivoj `x` dodeljuje vrednost izraza `x + 2`, tj. da se izračunava vrednost koja je za 2 veća od trenutne vrednosti promenljive `x` i da se vrednost tog izraza smešta u promenljivu `x`. Na taj način se zapravo vrednost promenljive `x` uvećava za 2.
Operator dodele, znači, ne označava jednakost dve vrednosti (za to se koristi operator poređenja jednakosti `==`, o čemu će biti više reči u poglavlju 5.1).
Dodele se mogu „ulančavati“. Na primer, naredna naredba postavlja vrednosti promenljivih `x` i `y` na nulu.

```
int x, y;
x = y = 0;
```

Operator dodele `=` ima desnu asocijativnost, pa je navedeni izraz ekvivalentan izrazu `x = (y = 0)` i promenljivoj `x` se dodeljuje vrednost izraza `y = 0`. Taj izraz ima vrednost 0. Naime, tip izraza dobijenog primenom operatora dodele je tip leve strane, a vrednost izraza dodele je vrednost koja će biti dodeljena levoj strani (što nije uvek vrednost koju ima desna strana). Promena vrednosti promenljive na levoj strani je *propratni* ili *sporedni efekat* (engl. side effect) do kojeg dolazi prilikom izračunavanja vrednosti izraza.
Stavljanjem simbola `;` na kraj izraza dodele (kao i na kraj bilo kog drugog izraza) dobija se naredba dodele (na primer, `x = 1` je izraz dodele, a `x = 1;` naredba dodele). Prilikom njenog izvršavanja izračunava se vrednost izraza dodele, realizuje se propratni efekat, a izračunata vrednost se zanemaruje.

3.3.2 *Razmena vrednosti promenljivih*

Često je potrebno razmeniti vrednosti dve promenljive.
Naredbe

```
a = b;
b = a;
```

ne bi dovele do željenog efekta, jer bi se promenljivoj `b` dodelila stara vrednost promenljive `a`. Da bismo postigli željeni efekat, potrebno je koristiti pomoćnu promenljivu istog tipa. Pretpostavimo da je potrebno razmeniti vrednosti promenljivih `a` i `b` (i da je raspoloživa pomoćna promenljiva `t`).

```
t = a;
a = b;
b = t;
```

Navedeni algoritam za razmenu dve vrednosti može se uopštiti tako da vrši cikličnu zamenu vrednosti više promenljivih. Na primer, na sledeći način može se izvršiti ciklična zamena vrednosti tri promenljive `a`, `b`, `c` (uz korišćenje pomoćne promenljive `t`):

```
t = a;
a = b;
b = c;
c = t;
```

Napomenimo i da je u jeziku C++ korisnicima na raspolaganju funkcija `swap` (deklarisana u zaglavlju `<utility>`), kojom se razmenjuju vrednosti dve promenljive.

```
swap(a, b);
```

Zadatak: Cena hleba

Hleb je prvo poskupeo 10%, pa je zatim pojeftinio 10%. Ako je poznata početna cena hleba, napiši program koji određuje cenu nakon tih promena.

Opis ulaza

Sa standardnog ulaza se učitava početna cena hleba (realan broj zaokružen na dve decimale).

Opis izlaza

Na standardni izlaz ispisati krajnju cenu hleba (realan broj zaokružen na dve decimale).

Primer

Ulaz	*Izlaz*
`35.50`	`35.15`

Rešenje

Jedno rešenje bilo bi da se uvedu tri promenljive (jedna za početnu cenu, jedna za cenu posle poskupljenja i jedna za cenu posle pojeftinjenja).

```
double pocetna_cena;
cin >> pocetna_cena;
double cena_posle_poskupljenja = 1.1 * pocetna_cena;
double cena_posle_pojeftinjenja = 0.9 *
 ↪  cena_posle_poskupljenja;
cout << fixed << showpoint << setprecision(2)
     << cena_posle_pojeftinjenja << endl;
```

Zadatak možemo da rešimo i korišćenjem samo jedne promenljive za cenu, menjajući vrednost te promenljive tokom izvršavanja programa. Ovaj stil programiranja je karakterističan za imperativno programiranje.

```
double cena;
cin >> cena;
cena = 1.1 * cena;
cena = 0.9 * cena;
cout << fixed << showpoint << setprecision(2) << cena <<
 ↪  endl;
```

4. Izrazi i izračunavanje

Jedan od osnovnih gradivnih elemenata svakog programa jesu *izrazi*, kojima se opisuju *izračunavanja*. To su često aritmetička izračunavanja nad različitim tipovima brojeva, ali u izračunavanjima i izrazima mogu učestvovati i drugi tipovi (videli smo već, na primer, da se i niske mogu sabirati tj. nadovezivati). U ovom poglavlju ćemo detaljnije proučiti sledeće elemente programskog jezika C++:

- *Operatori,* koji odgovaraju operacijama koje su definisane nad podacima određene vrste.
- *Izrazi,* koji kombinuju promenljive i konstante, korišćenjem operatora, dajući nove vrednosti.

4.1 Aritmetički operatori i zapis matematičkih formula

Nad operandima brojevnih tipova mogu se primeniti sledeći aritmetički operatori:

- `+` binarni operator sabiranja;
- `-` binarni operator oduzimanja;
- `*` binarni operator množenja;
- `/` binarni operator (celobrojnog) deljenja;
- `%` binarni operator ostatka pri deljenju;
- `-` unarni operator promene znaka;
- `+` unarni operator.

Operator `%` moguće je primeniti isključivo nad operandima celobrojnog tipa.
Operator deljenja označava različite operacije u zavisnosti od tipa svojih operanada.[1]

[1] Na hardveru su operacije nad celim brojevima i brojevima u pokretnom zarezu implementirane nezavisno i u izvršivom programu koristi se jedna od njih, izabrana u fazi prevođenja u zavisnosti od tipova operanada. Informacije o tipovima iz izvornog programa su na ovaj, ali i na druge slične načine, upotrebljene tokom prevođenja i one se ne čuvaju u izvršivom programu.

Kada se operator deljenja primenjuje na dve celobrojne vrednosti, primenjuje se celobrojno deljenje (tj. rezultat je ceo deo količnika). Na primer, izraz `9/5` ima vrednost `1`. Precizirajmo ovo.

Broj q se naziva *celobrojni količnik*, a broj r *ostatak* pri deljenju prirodnih brojeva a i b ($b \neq 0$) ako je $a = b \cdot q + r$ i ako je $0 \leq r < b$.

Celobrojni količnik brojeva a i b obeležava se često sa $a \operatorname{div} b$ ili sa $\left\lfloor \frac{a}{b} \right\rfloor$ ($\lfloor \ldots \rfloor$ označava zaokruživanje naniže, tj. najveći ceo broj koji je manji ili jednak datom broju), dok se ostatak često označava sa $a \bmod b$. Važi da je $a \operatorname{div} b = \left\lfloor \frac{a}{b} \right\rfloor$, tj. da je $q = a \operatorname{div} b$ najveći ceo broj q takav da je $q \cdot b \leq a$, što opravdava i korišćenje oznake $\left\lfloor \frac{a}{b} \right\rfloor$ za celobrojni količnik brojeva a i b.

Ostatak pri deljenju negativnog broja može biti negativan. Naime, u jeziku C++, vrednost izraza `(-9) / 5` je `-1`, a `(-9) % 5` je `-4`. Pošto različiti programski jezici na različite načine definišu celobrojno deljenje negativnih vrednosti, ovu operaciju je poželjno izbegavati (kada je to moguće).

Kada je bar jedan operand operatora `/` realan, primenjuje se deljenje realnih brojeva (preciznije, deljenje brojeva u pokretnom zarezu). Na primer, izraz `9.0/5.0` ima vrednost `1.8` (jer se koristi deljenje brojeva u pokretnom zarezu). U slučaju da je jedan od operanada ceo broj, a drugi broj u pokretnom zarezu, vrši se implicitna konverzija celobrojnog operanda u broj u pokretnom zarezu i primenjuje se deljenje brojeva u pokretnom zarezu.

Po uzoru na uobičajene matematičke konvencije, operatori `*`, `/` i `%` međusobno imaju isti prioritet, viši od prioriteta binarnih operatora `+` i `-` koji, takođe, međusobno imaju isti prioritet.

Kada se u istom izrazu izostave zagrade, a isti operator se primeni više puta, potrebno je precizirati kojim redosledom se vrši izračunavanje. Na primer, da li je vrednost izraza `1 - 2 - 3` jednaka `(1 - 2) - 3` tj. `-4` ili `1 - (2 - 3)` tj. `2`. Svi navedeni binarni operatori imaju levu asocijativnost, što znači da se izračunavanje operatora istog prioriteta uvek vrši sleva nadesno i prethodni izraz ima vrednost `-4`. Zanemarivanje ovog detalja može biti nekada izvor grešaka. Na primer,

```
x1 = (-b + sqrt(b*b - 4*a*c)) / 2*a;
```

ne daje ispravno rešenje kvadratne jednačine, jer se umesto deljenja sa `2*a`, kao što razmaci sugerišu[2], izračunavanje zapravo vrši sleva nadesno, pa se brojilac deli sa `2`, pa zatim množi sa `a`.

Prefiksni unarni operatori `+` i `-` imaju desnu asocijativnost i viši prioritet od svih binarnih operatora.

[2] Razmaci umesto zagrada, ali i neka druga implicitna pravila, u matematici sugerišu grupisanje operanada i redosled primene operacija. Na primer, podrazumeva se da je vrednost izraza $2x \,/\, 2x$ jednaka 1, što znači da se operacije ne izvršavaju sleva nadesno. U programiranju se razmaci zanemaruju i grupisanje se jedino može postići eksplicitnim korišćenjem zagrada.

4.1.1 Složeni operatori dodele

Pošto se uvećanje vrednosti promenljive za neku vrednost često javlja u programima, uvedeni su posebni operatori složene dodele. Dodela `x = x + 2;` se može kraće zapisati i kao `x += 2;`. Slično, naredba `x = x * (y+1);` ima isto dejstvo kao i `x *= y+1;`. Za većinu binarnih operatora postoje složeni operatori dodele (na primer, `+=`, `*=`, `/=`, `%=`). Ovi operatori imaju niži prioritet od svih ostalih operatora i desnu asocijativnost. Kao i u slučaju operatora dodele, izrazi u kojima učestvuju ovi operatori imaju vrednost, ali se te vrednosti obično zanemaruju, a osnovni razlog primene ovih operatora je njihov propratni efekat (izmena vrednosti promenljive na levoj strani).
Izračunavanje vrednosti izraza `izraz1 op= izraz2` obično ima isto dejstvo kao i izračunavanje vrednosti izraza `izraz1 = izraz1 op izraz2`, gde je `op` jedan od nabrojanih operatora, međutim, postoje i slučajevi kada dva navedena izraza imaju različite vrednosti[3], pa treba biti obazriv prilikom upotrebe ovih operatora.

4.1.2 Inkrementiranje i dekrementiranje

Najčešće promene vrednosti promenljivih su uvećanje ili umanjenje za 1. Zato se uvode posebni operatori *inkrementiranja* i *dekrementiranja*. Inkrementiranje generalno znači postepeno uvećavanje ili uvećavanje za neku konkretnu vrednost. U programiranju se pod inkrementiranjem obično podrazumeva uvećavanje za 1, a pod dekrementiranjem umanjivanje za 1. Operator inkrementiranja (uvećavanja za 1) zapisuje se sa `++`, a operator dekrementiranja (umanjivanja za 1) zapisuje se sa `--`:

- `++` (prefiksno i postfiksno) inkrementiranje
- `--` (prefiksno i postfiksno) dekrementiranje.

Oba operatora mogu se primeniti nad celim brojevima i brojevima u pokretnom zarezu. Inkrementirati i dekrementirati se mogu samo promenljive, elementi nizova i slično (tzv. *l-vrednosti*), ali ne i konstante ili proizvoljni izrazi. Tako, na primer, izraz `5++` nije ispravan.
Oba operatora su unarna (imaju po jedan operand) i mogu se upotrebiti u prefiksnom (na primer, `++x`) ili postfiksnom obliku (na primer, `x++`). Razlika između ova dva oblika je u tome što `++x` uvećava vrednost promenljive `x` pre nego što je ona upotrebljena u širem izrazu, a `x++` je uvećava nakon što je upotrebljena. Preciznije, vrednost izraza `x++` je stara vrednost promenljive `x`, a vrednost izraza `++x` je nova vrednost promenljive `x`, pri čemu se u oba slučaja, prilikom izračunavanja vrednosti izraza, kao propratni efekat, uvećava vrednost promenljive `x`. Na primer, ako promenljiva `x` ima vrednost `5`, onda

[3] To se najčešće dešava u situacijama kada izračunavanje izraza `izraz1` proizvodi neki propratni efekat. Na primer, prilikom izračunavanja vrednosti izraza `a[i++] += 5`, promenljiva `i` se uvećava jednom, a prilikom izračunavanja vrednosti izraza `a[i++] = a[i++] + 5` dva puta.

```
y = x++;
```

dodeljuje promenljivoj y vrednost 5, a

```
y = ++x;
```

dodeljuje promenljivoj y vrednost 6. Promenljiva x u oba slučaja dobija vrednost 6. Ukoliko ne postoji širi kontekst, tj. ako inkrementiranje čini čitavu naredbu, vrednost izraza se i ne koristi i onda nema razlike između naredbe `x++;` i `++x;`.
Primetimo da semantika operatora inkrementiranja može biti veoma komplikovana[4] i stoga se ne savetuje korišćenje složenijih izraza sa ovim operatorima (na primer, izraz poput `x++ + ++x` dovodi do nedefinisanog ponašanja programa, jer se promenljiva x dva puta menja u okviru istog izraza bez utvrđenog redosleda). U jednostavnim izrazima i situacijama upotreba operatora inkrementiranja i dekrementiranja je, naravno, legitimna, a često i poželjna (jer omogućava elegantan zapis). Na primer, u složenim izrazima, veoma česta je upotreba postfiksne varijante u sklopu upisa elementa na narednu slobodnu poziciju u nizu `a[i++] = x` (vrednost x se upisuje na slobodnu poziciju `i`, nakon čega se ta slobodna pozicija uvećava za 1 tj. pomera na naredno mesto u nizu), a prefiksne varijante u sklopu uklanjanja iz niza poslednjeg upisanog elementa `x = a[--i]` (prvo se slobodna pozicija umanjuje za 1, a onda se promenljivoj x dodeljuje element koji je bio postavljen na tu poziciju – ona je bila popunjena, a nakon ove dodele se smatra slobodnom).

4.2 Zapis matematičkih formula

U mnogim oblastima nauke i tehnike vrše se intenzivna izračunavanja u kojima se primenom matematičkih formula rezultati dobijaju na osnovu vrednosti ulaznih podataka. Na primer, na osnovu koordinata temena (x_1, y_1) i (x_2, y_2) nekog pravougaonika čije su ivice paralelne koordinatnim osama možemo odrediti dužinu njegovih stranica ($a = |x_1 - x_2|$, $b = |y_1 - y_2|$), a zatim i dužinu dijagonale $d = \sqrt{a^2 + b^2}$, obim $O = 2(a + b)$ i površinu $P = a \cdot b$.

```
#include <iostream>
#include <iomanip>
#include <cmath>
```

[4] Precizan trenutak realizovanja propratnog efekta je u standardu formalizovan kroz relaciju *redosleda izvršavanja* (engl. sequenced before); u starijim verzijama standarda i u jeziku C koristio se pojam *sekvencione tačke* (engl. sequence point).

```
using namespace std;

int main() {
   int x1, x2, y1, y2, a, b;
   cin >> x1 >> y1 >> x2 >> y2;
   a = abs(x2 - x1);
   b = abs(y2 - y1);
   double dijagonala = sqrt(a*a + b*b);
   int obim = 2*(a + b);
   int povrsina = a*b;
   cout << fixed << showpoint << setprecision(2)
        << dijagonala << endl
        << obim << endl
        << povrsina << endl;
   return 0;
}
```

4.3 Sekvencijalni programi

Sekvencijalni programi tj. sekvencijalne funkcije sastoje se od niza naredbi koje se izvršavaju jedna za drugom. Naredbe se izvršavaju istim redom bez obzira na podatke koji se obrađuju. U kodu nema ni grananja ni petlji.
U ovom poglavlju prikazaćemo primere nekih jednostavnih sekvencijalnih programa.

4.3.1 Sekvencijalno izračunavanje vrednosti

U mnogim izračunavanjima, na primer, u zadacima iz matematike i fizike, potrebno je izračunati neke međurezultate na putu do konačnog rešenja. Takva izračunavanja možemo da opišemo sekvencijalnim programima.
Razmotrimo problem izračunavanja visine `H` pravilnog tetraedra zadate stranice `a`. Najpre se može, korišćenjem Pitagorine teoreme, izračunati visina `h` trougla koji je osnova tetraedra. Zatim se može primeniti Pitagorina teorema na pravougli trougao čija je hipotenuza bočna ivica `a`, a katete visina tetraedra `H` i dve trećine visine `h` (tj. rastojanje od temena osnove do centra osnove):

```
h = sqrt(a*a - (a/2)*(a/2));
x = (2*h)/3;
H = sqrt(a*a - x*x);
```

4.3.2 Celobrojno deljenje i ostatak

Date su dve promenljive `a` i `b` takve da je `b > 0`. Zadatak je u prvu smestiti vrednost `a div b`, a u drugu `a mod b`. Naivni pokušaj da se to uradi na sledeći način

```
a = a / b;
b = a % b;
```

nije ispravan, jer se prilikom izračunavanja ostatka koristi već izmenjena vrednost promenljive `a`. Neophodno je upotrebiti pomoćnu promenljivu, na primer:

```
tmp = a;
a = a / b;
b = tmp % b;
```

4.3.3 Pozicioni zapis (brojevi, vreme, uglovi)

U pozicionom zapisu brojeva, doprinos cifre ukupnoj vrednosti broja ne zavisi samo od vrednosti cifre, već i od njene pozicije u zapisu. Zapis $c_n c_{n-1} ... c_0$ u pozicionom sistemu sa osnovom b (to nekad obeležavamo sa $(c_n c_{n-1} ... c_0)_b$) odgovara broju

$$c_n \cdot b^n + c_{n-1} \cdot b^{n-1} + ... + c_1 \cdot b + c_0,$$

pri čemu za svaku cifru c_i važi $0 \leq c_i < b$.
Na primer, broj 1234 u osnovi 10 jednak je $1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10 + 4$.
Najčešće korišćeni brojevni sistem jeste dekadni sistem, tj. sistem sa osnovom 10. Broj x se u dekadnom sistemu može predstaviti u obliku $x = c_n \cdot 10^n + c_{n-1} \cdot 10^{n-1} + ... + c_1 \cdot 10 + c_0$, gde je c_0 cifra jedinica, c_1 cifra desetica, c_2 cifra stotina itd. i za svaku od njih važi $0 \leq c_i \leq 9$.
Za zapis i vremena i uglova koristi se pozicioni zapis u osnovi 60 (jedan sat ima 60 minuta, jedan vremenski minut ima 60 vremenskih sekundi; jedan ugaoni stepen ima 60 ugaonih minuta, jedan ugaoni minut ima 60 ugaonih sekundi).
Vrednost broja se može odrediti i pomoću Hornerove sheme

$$(... ((c_n \cdot b + c_{n-1}) \cdot b + c_{n-2}) \cdot b + ... + c_1) \cdot b + c_0.$$

Na primer, broj 1234 u osnovi 10 jednak je $((1 \cdot 10 + 2) \cdot 10 + 3) \cdot 10 + 4$.
Poslednja cifra u dekadnom zapisu broja može se odrediti operacijom izračunavanja ostatka pri deljenju sa 10. Na primer, poslednja cifra broja 1234 je 4, što je upravo ostatak pri deljenju tog broja sa 10. Slično, poslednja cifra zapisa broja u osnovi b može se odrediti operacijom izračunavanja ostatka pri deljenju broja sa b. Dokažimo ovo. Broj x,

čije su cifre redom $c_n c_{n-1} \ldots c_1 c_0$, predstavlja se u obliku $x = c_n \cdot b^n + c_{n-1} b^{n-1} + \ldots + c_1 \cdot b + c_0$. Pošto su svi sabirci osim poslednjeg deljivi sa b, tj. broj se može napisati u obliku $x = (c_n \cdot b^{n-1} + c_{n-1} \cdot b^{n-2} + \ldots + c_1) \cdot b + c_0$ i pošto je $0 \leq c_0 < b$, na osnovu definicije celobrojnog količnika i ostatka važi da je $x \bmod b$ jednako c_0.
Opštije, cifru c_k uz koeficijent b^k u zapisu broja x možemo odrediti kao

$$(x \operatorname{div} b^k) \bmod b.$$

Dokažimo i ovo. Pokažimo najpre da je $x \operatorname{div} b^k = c_n \cdot b^{n-k} + c_{n-1} \cdot b^{n-k-1} + \ldots + c_{k+1} \cdot b + c_k$. Zaista, važi da je $x = (c_n \cdot b^{n-k} + c_{n-1} \cdot b^{n-k-1} + \ldots + c_{k+1} \cdot b + c_k) \cdot b^k + c_{k-1} b^{k-1} + \ldots + c_1 b + c_0$. Pošto za svaku cifru c_i važi $0 \leq c_i \leq b - 1$, važi da je $c_{k-1} b^{k-1} + \ldots + c_1 b + c_0 \leq (b - 1) \cdot (b^{k-1} + \ldots + b + 1) = b^k - 1$. Zato je $x \bmod b^k = c_{k-1} b^{k-1} + \ldots + c_1 b + c_0$, dok je $x \operatorname{div} b^k = c_n \cdot b^{n-k} + c_{n-1} \cdot b^{n-k-1} + \ldots + c_{k+1} \cdot b + c_k$. Zato se do cifre c_k može doći određivanjem ostatka pri deljenju ovog broja sa b (svi sabirci osim poslednjeg su deljivi sa b, dok je $0 \leq c_k < b$).

4.3.3.1 *Izračunavanje zbira cifara petocifrenog broja*

Na osnovu gore navedenih zapažanja, veoma jednostavno se može napisati program koji izračunava zbir svih cifara unetog petocifrenog broja.

```
// polazni broj i njegovo ucitavanje
int broj;
cin >> broj;

// izracunavanje zbira cifara
int cifraJedinica         = (broj / 1) % 10;
int cifraDesetica         = (broj / 10) % 10;
int cifraStotina          = (broj / 100) % 10;
int cifraHiljada          = (broj / 1000) % 10;
int cifraDesetinaHiljada  = (broj / 10000) % 10;
int zbirCifara = cifraJedinica + cifraDesetica +
                 cifraStotina  + cifraHiljada +
                 cifraDesetinaHiljada;

// prikaz rezultata
cout << zbirCifara << endl;
```

4.3.3.2 *Razmenjivanje cifre jedinica i stotina*

Naredni program prvo izdvaja, a zatim i razmenjuje cifre jedinica i stotina datog broja (ako je broj manji od 100, njegova cifra stotina je 0).

```
// ucitavamo broj
int broj;
cin >> broj;

// odredjujemo cifru jedinica i cifru stotina
int c0 = (broj / 1)   % 10;
int c2 = (broj / 100) % 10;

// uklanjamo cifre i dodajemo ih u razmenjenom poretku
int broj_r = broj - c0 - c2 * 100 +
                    c2 + c0 * 100;

// ispisujemo rezultat
cout << broj_r << endl;
```

4.3.3.3 *Izračunavanje vremena između dva trenutka*

Na sličan način možemo organizovati i računanje u osnovi 60, što je veoma pogodno za rešavanje problema sa vremenom i uglovima. Na primer, ako su poznati sat, minut i sekund početka i kraja vožnje autobusom (pretpostavljamo da je vožnja počela i završila se u istom danu), naredni program određuje koliko sati, minuta i sekundi je trajala ta vožnja. Najjednostavnije rešenje je ako se prvo sati, minuti i sekundi pretvore u sekunde, izvrše željene aritmetičke operacije i zatim dobijeni rezultat prevede nazad u sate, minute i sekunde.

```
// pocetak i kraj voznje
int hPocetak, mPocetak, sPocetak;
cin >> hPocetak >> mPocetak >> sPocetak;
int hKraj, mKraj, sKraj;
cin >> hKraj >> mKraj >> sKraj;

// trajanje voznje
// prevodimo pocetak u sekunde (protekle od ponoci)
int SPocetak = hPocetak*60*60 + mPocetak*60 + sPocetak;
// prevodimo kraj u sekunde (protekle od ponoci)
int SKraj    = hKraj*60*60 + mKraj*60 + sKraj;
// trajanje voznje u sekundama
int STrajanje = SKraj - SPocetak;
// prevodimo sekunde u sate, minute i sekunde
int sTrajanje = STrajanje % 60;
```

```
int mTrajanje = (STrajanje / 60) % 60;
int hTrajanje = STrajanje / (60*60);

cout << hTrajanje << ":" << mTrajanje << ":" << sTrajanje <<
 ↪  endl;
```

Po sličnom principu možemo vršiti izračunavanja i u sistemu sa mešovitim brojevnim osnovama. Na primer, vreme se ponekad izražava brojem milisekundi proteklih od nekog fiksnog trenutka (na UNIX sistemima to je 1. januar 1970. godine, a u narednom zadatku trenutak uključivanja računara). Narednim funkcijama se na osnovu broja proteklih dana, sati, minuta, sekundi i milisekundi od tog trenutka izračunava broj proteklih milisekundi i obratno. Primetimo da tip `int` može da zapiše samo oko 24 dana izraženih u milisekundama, što je u ovom zadatku dovoljno; za veće vremenske raspone potrebno je koristiti tip `long long`.

```
struct Vreme {
  int dan, sat, min, sek, mili;
};

// prevodi dane, sate, minute, sekunde i milisekunde u
 ↪  milisekunde
int uMilisekunde(Vreme v) {
  // Hornerova shema
  return (((v.dan*24 + v.sat)*60 + v.min)*60 + v.sek)*1000 +
   ↪  v.mili;
}

// prevodi milisekunde u dane, sate, minute, sekunde i
 ↪  milisekunde
Vreme odMilisekundi(int ms) {
  Vreme v;
  v.mili = (ms / 1) % 1000;
  v.sek  = (ms / 1000) % 60;
  v.min  = (ms / (1000*60)) % 60;
  v.sat  = (ms / (1000*60*60)) % 24;
  v.dan  = (ms / (1000*60*60*24));
  return v;
}
```

4.3.3.4 *Izračunavanje ugla između kazaljki na satu*

Računanje u osnovi 60 se može koristiti i za računanje sa uglovima. Na primer, narednim programom izračunavamo ugao između kazaljki na satu (izražen brojem ugaonih stepeni i ugaonih minuta).

Neka je dati vremenski trenutak opisan parametrima *sat* i *minut*.

Ugao koji minutna kazaljka zaklapa u odnosu na početni položaj od nula minuta (takozvani ugaoni otklon minutne kazaljke) jednak je $minut \cdot 6^\circ$. Zaista, na svaki minut vremena minutna kazaljka se pomera za $\frac{360^\circ}{60} = 6^\circ$.

Ugao koji satna kazaljka zauzima u odnosu na položaj 12h (ugaoni otklon satne kazaljke) jednak je $sat \cdot 30^\circ + minut \cdot 0{,}5^\circ$. Zaista, na svaki sat kazaljka se pomeri za $\frac{360^\circ}{12} = 30^\circ$. Na svaki minut vremena satna kazaljka se pomeri dodatno za $\frac{30^\circ}{60} = 0{,}5^\circ$. Zaista, ona se za jedan minut vremena pomeri 12 puta manje nego minutna kazaljka, za koju smo ustanovili da se za minut vremena pomeri za 6°.

Da bismo izbegli računanje sa realnim brojevima, možemo ove uglove izraziti u ugaonim minutima. Minutna kazaljka se u svakom minutu pomeri za $6^\circ = 6^\circ \cdot \frac{60'}{1^\circ} = 360'$. Satna kazaljka se u svakom satu pomeri za $30 \cdot 60' = 1800'$ i u svakom minutu dodatno za $0{,}5^\circ = 30'$.

Ugao između kazaljki izražen u minutima izračunavaćemo kao apsolutnu vrednost razlike njihovih ugaonih otklona. Tako dobijeni ugao može biti i veći od 180° (na primer, u 9 sati on iznosi 270°, dok manji ugao između kazaljki iznosi 90°). Na kraju je dobijeni rezultat potrebno prevesti u stepene i minute.

```
// ucitavamo vreme
int sat, minut;
cin >> sat >> minut;

// sat svodimo na interval [0, 12)
sat %= 12;

// ugao u minutima koji satna kazaljka zauzima sa polozajem
 ↪  12h
int ugaoSatne   = sat * 30 * 60 + minut * 30;
// ugao u minutima koji minutna kazaljka zauzima sa polozajem
 ↪  12h
int ugaoMinutne = minut * 360;
// ugao izmedju satne i minutne kazaljki u minutima
int ugaoIzmedju = abs(ugaoSatne - ugaoMinutne);
// ugao izmedju kazaljki u stepenima i minutima
int ugaoIzmedjuStepeni = ugaoIzmedju / 60;
```

```
int ugaoIzmedjuMinuti  = ugaoIzmedju % 60;

// ispis rezultata
cout << ugaoIzmedjuStepeni << ":" << ugaoIzmedjuMinuti <<
 ↪  endl;
```

5. Grananje

Naredbe grananja (ili naredbe uslova), na osnovu vrednosti nekog izraza, određuju naredbu (ili grupu naredbi) koja će biti izvršena. Uslovi se izražavaju korišćenjem relacijskih i logičkih operatora.

5.1 Relacijski i logički operatori i istinitosna vrednost izraza

5.1.1 Logički tip podataka

Tip `bool` je tip za predstavljanje istinitosnih (logičkih) vrednosti i ima sledeće moguće vrednosti: `true` koja označava *tačno* i `false` koja označava *netačno*. Dodatno, svaki brojevni izraz ima istinitosnu vrednost: *netačno* ako je jednak `0`, a *tačno* inače — možemo smatrati da se na taj način vrši konverzija brojevnih tipova u tip `bool`. Konverzija tipa `bool` u tip `int` se vrši tako što se vrednost `true` tumači kao broj 1, a `false` kao 0.

5.1.2 Relacijski i logički operatori

Nad celim brojevima i brojevima u pokretnom zarezu mogu se koristiti sledeći binarni relacijski operatori:

- `==` jednako;
- `!=` različito;
- `>` veće;
- `>=` veće ili jednako;
- `<` manje;
- `<=` manje ili jednako.

Relacijski operatori poretka `<`, `<=`, `>` i `>=` imaju isti prioritet, viši od operatora jednakosti `==` i različitosti `!=`, i svi imaju levu asocijativnost. Rezultat relacijskog operatora

primenjenog nad dva broja je tipa `bool`, tj. može imati vrednosti `false` ili `true`. Na primer, izraz `3 > 5` ima vrednost `false`.
Konverzije između brojevnih tipova i tipa `bool` omogućavaju i neka neobična i početnicima neočekivana ponašanja, koja mogu ponekad dovesti do grešaka. Na primer, u matematici se pripadnost promenljive x intervalu $(3, 5)$ može zapisati kao $3 < x < 5$. Međutim, ako `x` ima, na primer, vrednost `2`, onda u programu ovakav izraz ima vrednost `true` (što je različito od možda očekivane vrednosti `false`, jer `2` nije između `3` i `5`). Naime, izraz se izračunava sleva nadesno — podizraz `3 < x` ima vrednost `false`, koji se (u daljoj kombinaciji sa operatorom `<`) konvertuje u vrednost `0`, a zatim izraz `0 < 5` daje vrednost `true`. Ovo je opasna greška jer u ovakvim situacijama kompilator ne prijavljuje grešku, a u fazi izvršavanja se dobija rezultat neočekivan za početnike koji su navikli na uobičajenu matematičku notaciju. Imajući u vidu ponašanje relacijskih operatora, proveru da li je neka vrednost između dve zadate neophodno je vršiti uz primenu logičkog operatora `&&` izrazom `3 < x && x < 5`.
Kao i u matematici, binarni relacijski operatori imaju niži prioritet od binarnih aritmetičkih operatora (na primer, u izrazu `3 + 5*6 < 7*2` prvo se izračunavaju vrednosti `3 + 5*6` i `7*2`, pa se tek onda porede). U izračunavanju vrednosti izraza `3 + 5*6` prioritet ima operacija `*`.
Logički operatori primenjuju se nad vrednostima tipa `bool` i imaju tip rezultata `bool`. Postoje sledeći logički operatori:

- `!` logička negacija — $\neg$;
- `&&` logička konjunkcija — $\wedge$;
- `||` logička disjunkcija — $\vee$.

Operator `&&` ima viši prioritet u odnosu na operator `||`, a oba su levo asocijativna. Binarni logički operatori imaju niži prioritet u odnosu na binarne aritmetičke i relacijske operatore. Operator `!`, kao unarni operator, ima viši prioritet u odnosu na bilo koji binarni operator i desno je asocijativan. Na primer,

- vrednost izraza `!(2 > 3)` jednaka je `true`;
- izrazom `3 < x && x < 5` proverava se da li je vrednost promenljive `x` između `3` i `5`;
- izraz `a > b || b > c && b > d` ekvivalentan je izrazu `(a>b) || ((b>c) && (b>d))`;
- izrazom `g % 4 == 0 && g % 100 != 0 || g % 400 == 0` proverava se da li je godina g prestupna.

Implicitne konverzije brojevnih tipova u tip `bool` omogućavaju i da se logički operatori primene i na brojevne vrednosti (kao što je već rečeno, jedino se vrednost `0` konvertuje u `false`, dok se sve ostale brojevne vrednosti konvertuju u `true`). Na primer:

- vrednost izraza `5 && 4.3` jednaka je `true`;

- vrednost izraza `10.2 || 0` jednaka je `true`;
- vrednost izraza `0 && true` jednaka je `false`;
- vrednost izraza `!0` jednaka je `true`;
- vrednost izraza `!1` jednaka je `false`;
- vrednost izraza `!9.2` jednaka je `false`.

Pošto programski jezik C u svojim ranijim verzijama nije uopšte posedovao tip `bool`, korišćenje brojevnih vrednosti za predstavljanje istinitosnih se svojevremeno raširilo kao programerska praksa. Ipak, takvi programi su manje razumljivi i podložni greškama, pa savremeniji programski jezici proizišli iz jezika C (na primer, Java i C#) ukidaju mogućnost implicitne konverzije između brojevnog i logičkog tipa. Zato ćemo i mi izbegavati upotrebu takvih konverzija (iako ih jezik C++, po uzoru na jezik C, dopušta).
Kao i u matematici, logički operatori imaju manji prioritet u odnosu na relacijske. Na primer, u izrazu `3 < 4 && 2 > x`, prvo se izračunavaju vrednosti podizraza `3 < 4` i `2 > x` i zatim se tako dobijene logičke vrednosti kombinuju operatorom `&&`.
U izračunavanju vrednosti logičkih izraza koristi se strategija *skraćenog izračunavanja* (engl. *short-circuit evaluation*), koje se ponekad naziva i *lenjo izračunavanje*, mada se taj termin češće koristi za opštiju strategiju, karakterističnu za funkcionalne jezike, u kojoj se nijedan izraz ne izračunava dok njegova vrednost nije potrebna. Osnovna karakteristika ove strategije je da se izračunavanje vrednosti operanada vrši sleva nadesno i prestaje čim je moguće izračunati vrednost celog izraza na osnovu vrednosti do sada izračunatih operanada (računa se samo ono što je neophodno). Na primer, prilikom izračunavanja vrednosti izraza

```
2 < 1 && f(0)
```

biće izračunato da je vrednost podizraza `2 < 1` jednaka `false`, pa je sigurno i vrednost čitavog izraza (zbog svojstva logičkog $\wedge$ tj. `&&`) jednaka `false`. Zato nema potrebe izračunavati vrednost podizraza `f(0)`, pa funkcija `f` uopšte neće biti pozvana. S druge strane, tokom izračunavanja vrednosti izraza

```
f(0) && 2 < 1
```

funkcija `f` će biti pozvana (jer se vrednost logičkih izraza izračunava sleva nadesno). U izrazima u kojima se javlja operator `&&`, ukoliko je vrednost prvog operanda jednaka `true`, onda se izračunava i vrednost drugog operanda.
U izrazu u kojem se javlja logičko $\vee$ tj. operator `||`, ukoliko je vrednost prvog operanda jednaka `true`, onda se ne izračunava vrednost drugog operanda, jer se unapred može zaključiti da je vrednost celog izraza `true`. Ukoliko je vrednost prvog operanda jednaka `false`, onda se izračunava i vrednost drugog operanda. Na primer, u izračunavanju izraza

```
1 < 2 || f(0)
```

se ne poziva funkcija `f`, a poziva se u izračunavanju izraza

```
2 < 1 || f(0)
```

Skraćeno izračunavanje je vid optimizacije programa (jer se štedi vreme tako što se izbegavaju nepotrebna izračunavanja). Pored toga, oslanjanje na skraćeno izračunavanje može nekada da doprinese elegantnom pisanju programa. Na primer, razmotrimo kôd u kome se određuje pozicija prvog neparnog elementa datog niza ili vektora (čija je dužina `n`).

```
i = 0;
while (i < n && a[i] % 2 == 0)
  i++;
```

Ako je uslov `i < n` ispunjen, indeks `i` se sigurno nalazi u granicama niza (jer je ujedno i nenegativan) i bezbedno se ispituje da li je element na poziciji `i` paran. Međutim, ako uslov `i < n` nije ispunjen, petlja se odmah prekida, jer se stiglo do kraja niza. Usled skraćenog izračunavanja operatora `&&`, tada se uslov `a[i] % 2 == 0` ne proverava. U tom trenutku proveru tog uslova ne bismo ni smeli da vršimo, jer je `i` van granica niza, tako da nas skraćeno izračunavanje u ovom slučaju štiti od nedefinisanog ponašanja programa i potencijalne greške.

5.1.3 *Poređenje i poredak*

U mnogim problemima potrebno je uporediti dva objekta. U nekim situacijama potrebno je proveriti da li su dva objekta jednaka, a u nekim da li je jedan manji (ili veći) od drugog. Za poređenje vrednosti osnovnih, brojevnih tipova na raspolaganju su operatori `==`, `!=`, `<`, `>`, `<=`, `>=` sa odgovarajućim, uobičajenim matematičkim značenjem. Poređenje vrednosti složenih tipova svodi se na poređenje vrednosti osnovnih tipova.

5.1.3.1 *Relacija jednakosti*

Relacija jednakosti je relacija ekvivalencije: ona je refleksivna, simetrična i tranzitivna. Relacijski operator jednakosti (`==`), nad raspoloživim tipovima (na primer, `int`, `double`, `string`) zadovoljava ove uslove.[1] Ovaj operator može se koristiti za proveru jednakosti

[1] Za brojeve u pokretnom zarezu (ako se sledi standard IEEE 754, što standard jezika C++ ne propisuje), ovo važi samo za skup vrednosti bez pozitivne i negativne vrednosti „not-a-number“ (`NaN`). Naime, izraz `NaN==NaN` nije tačan. Dodatno, treba naglasiti da se svojstva relacije ekviva-

dve vrednosti osnovnih tipova, a za korisnički definisane tipove u jeziku C++ može da se definiše (za razliku od jezika C).
Za dve vrednosti tipa neke strukture, provera jednakosti svodi se na proveru jednakosti svih članova pojedinačno ili možda na neki drugi način. Na primer, dva razlomka su jednaka ne samo ako su im i imenilac i brojilac jednaki, nego i u nekim drugim slučajevima. Ako je struktura `razlomak` zadata na sledeći način

```
struct razlomak {
  int brojilac;
  int imenilac;
};
```

onda se jednakost dva razlomka (definisanih vrednosti) može ispitati narednom funkcijom:

```
bool jednaki_razlomci(const razlomak& a, const razlomak& b)
{
  return a.imenilac * b.brojilac == b.imenilac * a.brojilac;
}
```

Kvalifikatorom `const` mogu, kao i sve promenljive, biti označeni parametri funkcije, čime se obezbeđuje da neki parametar ili sadržaj na koji ukazuje neki parametar neće biti menjan u funkciji. Znak `&` u `const razlomak&` određuje način prenosa argumenta, o čemu će biti reči u poglavlju 7.6.2.
Primetimo da u prethodnoj funkciji postoji opasnost od nastanka prekoračenja, pa je treba koristiti veoma obazrivo.

5.1.3.2 *Relacije poretka*

Relacija poretka je relacija koja je refleksivna, antisimetrična i tranzitivna. Takva je, na primer, relacija $\leq$ nad skupom prirodnih brojeva. Slično, relacijski operatori `<=` i `>=` nad osnovnim brojevnim tipovima određuju relacije poretka. *Relacija strogog poretka* je relacija koja je antirefleksivna, antisimetrična i tranzitivna. Takva je, na primer, relacija

lencije odnose samo na vrednosti koje pripadaju istim tipovima. Na primer, nakon naredbe `float x = 0.1;`, promenljiva `x` ima (na nekom sistemu) vrednost `0.100000001490116119384765625`, a nakon naredbe `double x = 0.1;`, promenljiva `x` ima (na istom tom sistemu) vrednost `0.1000000000000000055511151231257827021181583404541015625`. Treba, dakle, imati na umu i da su rezultati operacija (čak i jednostavnih dodela) nad brojevima u pokretnom zarezu često dobijeni zaokruživanjem. Zbog toga, vrednosti dva izraza mogu biti različite i kada su vrednosti odgovarajućih izraza nad realnim brojevima jednake. Drugim rečima, treba uvek imati na umu da su matematička pravila za brojeve u pokretnom zarezu drugačija od matematičkih pravila koja važe za realne brojeve.

< nad skupom prirodnih brojeva. Slično, relacijski operatori < i > nad osnovnim brojevnim tipovima određuju relacije strogog poretka.[2] Vrednosti osnovnih, brojevnih tipova mogu, međutim, da se porede i na neki drugi način – na primer, ako je potrebno pronaći broj koji je najbliži zadatoj vrednosti, cele brojeve je potrebno porediti prema odnosu njihovih rastojanja od te zadate vrednosti (važi da je a bliže x od b, ako je $|a-x| < |b-x|$). Takođe, za druge tipove potrebno je implementirati funkcije koje vrše poređenje, a one se obično zasnivaju na poređenju za jednostavnije tipove. Funkcije za poređenje dve vrednosti obično vraćaju vrednost manju od nule ako je prvi argument manji, nulu ako su argumenti jednaki i vrednost veću od nule ako je drugi argument manji. Na primer, naredna funkcija poredi dvoslovne oznake država po standardu ISO 3166[3]. Ona vraća vrednost -1 ako je prvi kôd manji, 0 ako su zadati kodovi jednaki i 1 ako je drugi kôd manji:

```
int porediKodoveDrzava(const string& a, const string& b)
{
  if (a[0] < b[0])
    return -1;
  if (a[0] > b[0])
    return 1;
  if (a[1] < b[1])
    return -1;
  if (a[1] > b[1])
    return 1;
  return 0;
}
```

Parovi karaktera se, dakle, mogu porediti tako što se najpre porede prvi karakteri u parovima, a zatim, ako je potrebno, drugi karakteri. Slično se mogu porediti i datumi opisani narednom strukturom:

```
struct datum {
  unsigned dan;
  unsigned mesec;
  unsigned godina;
```

[2] Ako se sledi standard IEEE 754 za brojeve u pokretnom zarezu, ovo važi samo za skup vrednosti bez pozitivne i negativne vrednosti „not-a-number“ (`NaN`).

[3] Svrha standarda ISO 3166 je definisanje međunarodno priznatih dvoslovnih kodova za države ili neke njihove delove. Kodovi su sačinjeni od po dva slova engleskog alfabeta. Koriste se za oznaku nacionalnih internet domena, od strane poštanskih organizacija, i dr. Dvoslovna oznaka za Srbiju je „rs“, za Portugaliju „pt“, za Rusiju „ru“, itd.

```
};
```

Prvo se porede godine – ako su godine različite, redosled dva datuma može se odrediti na osnovu njihovog odnosa. Ako su godine jednake, onda se prelazi na poređenje meseci. Na kraju, ako su i meseci jednaki, prelazi se na poređenje dana. Naredna funkcija implementira ovaj algoritam i vraća -1 ako je prvi datum pre drugog, 1 ako je drugi datum pre prvog i 0 ako su jednaki.

```
int porediDatume(const datum& d1, const datum& d2)
{
  if (d1.godina < d2.godina)
    return -1;
  if (d1.godina > d2.godina)
    return 1;
  if (d1.mesec < d2.mesec)
    return -1;
  if (d1.mesec > d2.mesec)
    return 1;
  if (d1.dan < d2.dan)
    return -1;
  if (d1.dan > d2.dan)
    return 1;
  return 0;
}
```

Može se napisati i jedinstven logički izraz kojim se proverava da li je prvi datum ispred drugog:

```
bool datumPre(const datum& d1, const datum& d2)
{
  return
    d1.godina < d2.godina ||
    (d1.godina == d2.godina && d1.mesec < d2.mesec) ||
    (d1.godina == d2.godina && d1.mesec == d2.mesec &&
     d1.dan < d2.dan);
}
```

Generalno, torke sa fiksnim jednakim brojem elemenata mogu se porediti tako što se najpre porede njihovi prvi elementi, zatim, ako je potrebno, njihovi drugi elementi i tako dalje, sve dok se ne naiđe na neki različit par elemenata, na osnovu kojeg se određuje

poredak. Dakle, relacija poređenja pojedinačnih elemenata može se proširiti na relaciju poređenja n-torki elemenata, tj. ako je na skupu X (na primer, na skupu karaktera) definisana relacija poretka $\prec$, onda se može definisati i relacija poretka $\prec^l$ na skupu X^n (na primer, na niskama karaktera dužine n). Štaviše, relacija poretka nad pojedinačnim elementima skupa X može se proširiti i na skup $X^* = \bigcup_{n=0}^{+\infty} X^n$, tj. može se proširiti i na skup svih torki svih dužina. Poređenje se ponovo vrši redom i čim se naiđe na prvu poziciju na kojoj se u dve torke nalazi različit element, na osnovu njega određuje se poredak torki. Kada su torke različite dužine, tada se može desiti da se dođe do kraja jedne od njih. Ako se istovremeno došlo i do kraja druge, tada su torke jednake, a ako nije, tada je kraća torka prefiks one duže. Tada se smatra da je kraća torka manja (ide pre one duže torke). Ovako definisana relacija poretka naziva se *leksikografski poredak* (jer se koristi i u poretku odrednica u leksikonima i rečnicima). Za niske (potencijalno različitih dužina) može se definisati leksikografski poredak koji je zasnovan na poređenju karaktera. Kada se operatori `<`, `>`, `<=` i `>=` primene na tip `string`, vrši se upravo ovakav način poređenja. Pored ovoga, definisana je i metoda `compare` koja poredi niske. Poziv `str1.compare(str2)` vraća negativnu vrednost ako niska `str1` leksikografski prethodi niski `str2`, pozitivnu vrednost ako niska `str2` leksikografski prethodi niski `str1`, a nulu ako su niske jednake.

Niske mogu da se porede i na neki drugi način, na primer, samo po dužini:

```
int porediNiske(const string& a, const string& b)
{
  if (a.length() < b.length())
    return -1;
  if (a.length() > b.length())
    return 1;
  return 0;
}
```

Dva razlomka (čiji su imenioci pozitivni) mogu da se porede sledećom funkcijom (pod pretpostavkom da smo sigurni da prilikom množenja neće doći do prekoračenja), koja vraća negativan rezultat ako je prvi razlomak manji od drugog, pozitivan rezultat ako je prvi razlomak veći od drugog, a nulu ako su razlomci jednaki:

```
int porediRazlomke(const razlomak& a, const razlomak& b)
{
  return a.brojilac * b.imenilac - b.brojilac * a.imenilac;
}
```

5.2 Naredba `if-else`

Naredba uslova `if` ima sledeći opšti oblik:

```
if (izraz)
  naredba1
else
  naredba2
```

Deo naredbe `else` je opcioni, tj. može da postoji samo `if` grana.
Naredba `naredba1` i naredba `naredba2` su ili pojedinačne naredbe (kada se završavaju simbolom `;`) ili blokovi naredbi zapisani između vitičastih zagrada (iza kojih se ne piše simbol `;`). Vitičaste zagrade je moguće staviti i oko pojedinačnih naredbi, ali to nije neophodno. U praksi se događa da programeri naknadno požele da u postojeću naredbu `if` dodaju nove naredbe koje će se uslovno izvršiti. Ako se zagrade ne navedu, doći će ili do sintaksičke greške ili do semantičke greške, tj. dobiće se program koji ili ne može da se prevede ili se prevodi, ali ne radi na očekivani način. Na primer, pošto u narednom kodu nisu navedene zagrade, bez obzira na to kako je kôd poravnat[4], tj. kako su naredbe uvučene, `naredba1` će se izvršiti samo ako je uslov ispunjen, dok će se `naredba2` izvršiti i kada uslov jeste i kada nije ispunjen.

```
if (izraz)
   naredba1;
   naredba2;
```

Pošto vitičaste zagrade nisu navedene, prethodni kôd će se zapravo tumačiti kao:

```
if (izraz)
   naredba1;
naredba2;
```

Da se ovakve greške ne bi događale, nekada se savetuje da se i kada se u grani naredbe `if` (a kasnije ćemo videti i u telu petlje) nalazi samo jedna naredba, za svaki slučaj navode vitičaste zagrade.
Izraz `izraz` predstavlja logički uslov i najčešće je u pitanju izraz tipa `bool` (ali, usled implicitne konverzije, može biti i izraz brojevnog tipa, što nije preporučljivo). Na primer, nakon koda

[4] Interesantno, neki programski jezici, poput jezika Python, koriste poravnavanje naredbi da bi odredili koje se naredbe izvršavaju uslovno u sklopu naredbe grananja i koje se naredbe ponavljaju u sklopu petlji.

```
int a = 5, b = 3;
if (a > b)
  maksimum = a;
else
  maksimum = b;
```

promenljiva `maksimum` će imati vrednost `5` (jer je uslov `a > b` ispunjen).
Kako se ispituje istinitosna vrednost izraza koji je naveden kao uslov, ponekad je moguće taj uslov zapisati kraće. Na primer, `if (n != 0)` je ekvivalentno sa `if (n)`, što smanjuje čitljivost programa i stvara prostor za greške (ali se može često sresti, usled nasleđene tradicije iz ranih verzija programskog jezika C koje nisu imale poseban tip `bool`).
Dodela čini izraz čija se vrednost može konvertovati u istinitosnu vrednost, pa je naredni kôd sintaksički ispravan, ali je verovatno semantički pogrešan (tj. ne opisuje ono što je bila namera programera):

```
a = 3;
if (a = 0)
   cout << "a je nula" << endl;
else
   cout << "a nije nula" << endl;
```

Naime, efekat navedenog koda je da postavlja vrednost promenljive `a` na nulu (a ne da ispita da li je `a` jednako `0`), a zatim ispisuje tekst `a nije nula`, jer je vrednost izraza `a = 0` nula, što se smatra netačnim. Nehotično mešanje operatora `==` sa operatorom `=` u naredbi `if` je česta greška. Međutim, operator `==` koji ispituje da li su neke dve vrednosti jednake i operator dodele `=` različiti su operatori i imaju potpuno drugačije značenje. Na sreću, većina kompilatora daje upozorenje ako se operator dodele `=` koristi u okviru nekog logičkog uslova.
Naredbe koje se izvršavaju uslovno mogu da sadrže nove naredbe uslova, tj. može biti više ugnežđenih `if` naredbi. U takvim situacijama može biti nejasno na koju naredbu `if` se odnosi navedeni `else` (ta pojava se naziva `if-else` *višeznačnost*). Ukoliko vitičastim zagradama nije obezbeđeno drugačije, `else` se odnosi na poslednji prethodni neuparen `if`. Ukoliko se želi drugačije ponašanje, neophodno je navesti vitičaste zagrade. U narednom primeru, `else` se odnosi na drugo, a ne na prvo `if` (iako nazubljivanje sugeriše drugačije):

```
if (izraz1)
  if (izraz2)
```

```
    naredba1
else
  naredba2
```

U narednom primeru, `else` se odnosi na prvo, a ne na drugo `if` :

```
if (izraz1) {
  if (izraz2)
    naredba1
} else
  naredba2
```

Da bi se izbegle višeznačnosti ovog tipa i smanjila mogućnost nastajanja greške usled toga što su programer i kompilator shvatili program na različite načine, preporučuje se da se prilikom ugnežđavanja naredbi `if-else` uvek koriste zagrade (čak i kada se u telu nalazi samo jedna naredba).

5.2.1 *Konstrukcija `else-if`*

Za višestruke odluke često se koristi konstrukcija sledećeg oblika:

```
if (izraz1)
  naredba1
else if (izraz2)
  naredba2
else if (izraz3)
  naredba3
else
  naredba4
```

U ovako konstruisanoj naredbi, uslovi se ispituju jedan za drugim. Kada je jedan uslov ispunjen, onda se izvršava naredba koja mu je pridružena i time se završava izvršavanje čitave naredbe. Naredba `naredba4` u gore navedenom primeru se izvršava ako nije ispunjen nijedan od uslova `izraz1`, `izraz2`, `izraz3`. Naredni primer ilustruje ovaj tip uslovnog grananja.

```
if (a > 0)
  cout << "A je veci od nule" << endl;
else if (a < 0)
  cout << "A je manji od nule" << endl;
```

```
else /* if (a == 0) */
  cout << "A je nula" << endl;
```

5.3 Operator uslova

Naredna naredba

```
if (a > b)
  maksimum = a;
else
  maksimum = b;
```

određuje i smešta u promenljivu `maksimum` veću od vrednosti `a` i `b`. Naredba ovakvog oblika se može zapisati kraće korišćenjem ternarnog operatora uslova `?:`, na sledeći način:

```
maksimum = a > b ? a : b;
```

Naravno, maksimum dva broja u jeziku C++ uvek je bolje određivati korišćenjem bibliotečke funkcije `max` (potrebno je uključiti zaglavlje `<algorithm>`).
Ternarni operator uslova `?:` se koristi u sledećem opštem obliku:

```
izraz1 ? izraz2 : izraz3
```

Prioritet ovog operatora je niži od prioriteta skoro svih ostalih operatora (izuzetak su operatori dodele, koji su istog prioriteta).
Izraz `izraz1` se izračunava prvi. Ako on ima vrednost različitu od nule (tj. ako ima istinitosnu vrednost *tačno*), onda se izračunava vrednost izraza `izraz2` i to je vrednost čitavog uslovnog izraza. U suprotnom se izračunava vrednost izraza `izraz3` i to je vrednost čitavog uslovnog izraza. Na primer, vrednost izraza

```
x < 0 ? -x : x
```

je apsolutna vrednost broja `x`.
I kod ternarnog uslovnog operatora se izračunava samo jedan od izraza `izraz2` i `izraz3`. Naime, ako je vrednost izraza `izraz1` tačno, tada se izraz `izraz3` uopšte ne izračunava, a ako je netačno, tada se izraz `izraz2` uopšte ne izračunava.

5.4 Naredba switch

Naredba switch se koristi za višestruko odlučivanje i ima sledeći opšti oblik:

```
switch (izraz) {
    case konstantan_izraz1: naredbe1
    case konstantan_izraz2: naredbe2
    ...
    default:                naredbe_n
}
```

Naredbe koje treba izvršiti označene su *slučajevima* (engl. *case*) za različite moguće pojedinačne vrednosti zadatog izraza izraz. Svakom slučaju pridružen je konstantni celobrojni izraz. Ukoliko zadati izraz izraz ima vrednost konstantnog izraza navedenog u nekom slučaju, onda se izvršavanje nastavlja od prve naredbe pridružene tom slučaju, pa se nastavlja i sa izvršavanjem naredbi koje odgovaraju sledećim slučajevima, iako izraz nije imao njihovu vrednost, sve dok se ne naiđe na kraj ili naredbu break. Na slučaj default se prelazi ako vrednost izraza izraz nije navedena ni uz jedan slučaj. Slučaj default je opcioni i ukoliko nije naveden, a nijedan postojeći slučaj nije ispunjen, onda se ne izvršava nijedna naredba u okviru bloka switch. Slučajevi mogu biti navedeni u proizvoljnom poretku (uključujući i slučaj default), ali različiti poreci mogu da daju različito ponašanje programa. Iako to standard ne zahteva, slučaj default se gotovo uvek navodi kao poslednji slučaj. I ukoliko slučaj default nije naveden kao poslednji, ako vrednost izraza izraz nije navedena ni uz jedan drugi slučaj, prelazi se na izvršavanje naredbi od naredbe pridružene slučaju default.

U okviru naredbe switch često se koristi naredba break. Kada se naiđe na naredbu break, napušta se naredba switch. Najčešće se naredbe pridružene svakom slučaju završavaju naredbom break (čak i nakon poslednjeg navedenog slučaja, što je najčešće default). Time se obezbeđuje da poredak slučajeva ne utiče na izvršavanje programa, te je takav kôd jednostavniji za održavanje.

Izostavljanje naredbe break, tj. previđanje činjenice da se, ukoliko nema naredbi break, nastavlja sa izvršavanjem naredbi narednih slučajeva, često dovodi do grešaka u programu, pa je zato ta mogućnost zabranjena u nekim savremenijim programskim jezicima (na primer, u jeziku C#). S druge strane, izostavljanje naredbe break može biti pogodno (i opravdano) za pokrivanje više različitih slučajeva jednom naredbom (ili blokom naredbi).

U narednom primeru proverava se da li je uneti prirodan broj deljiv sa tri, korišćenjem naredbe switch.

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

```
int main() {
  int n;
  cin >> n;

  switch (n % 3) {
    case 1:
    case 2:
             cout << "Uneti broj nije deljiv sa 3";
             break;
    default: cout << "Uneti broj je deljiv sa 3";
  }
  return 0;
}
```

U navedenom primeru, bilo da je vrednost izraza `n % 3` jednaka `1` ili `2`, biće ispisan tekst `Uneti broj nije deljiv sa 3`, a inače će biti ispisan tekst `Uneti broj je deljiv sa 3`. Da nije navedena naredba `break`, onda bi u slučaju da je vrednost izraza `n % 3` jednaka `1` (ili `2`), nakon teksta `Uneti broj nije deljiv sa 3`, bio ispisan i tekst `Uneti broj je deljiv sa 3` (jer bi bilo nastavljeno izvršavanje svih naredbi za sve naredne slučajeve).

5.5 Primeri

Prikažimo u nastavku nekoliko tipičnih scenarija upotrebe grananja. Naredni spisak primera nije ni na koji način iscrpan, već samo pokazuje neke tipove problema koji se često mogu sresti. Programer uvek treba da pažljivom analizom problema osmisli strukturu uslova koje je potrebno ispitati da bi se pokrili svi mogući slučajevi i da bi problem bio ispravno rešen.

5.5.1 Broj dana u mesecu (grananje na osnovu vrednosti promenljive)

Veoma česta i jednostavna situacija je kada se grananje vrši na osnovu različitih pojedinačnih vrednosti neke promenljive.

Na primer, u narednom programu određuje se broj dana u mesecu na osnovu rednog broja meseca (od 1 do 12) i godine. Pošto broj dana u februaru zavisi od toga da li je godina prestupna, pogodno je definisati pomoćnu funkciju kojom se korišćenjem logičkog izraza ispituje da li je godina prestupna (podsetimo se, godina je prestupna ako je deljiva sa 4, a nije deljiva sa 100, ili ako je deljiva sa 400).

```
// provera da li je data godina prestupna
bool prestupna(int godina) {
  // godina je prestupna ako je deljiva sa 4 i nije deljiva
   ↪  sa 100,
  // ili ako je deljiva sa 400
  return (godina % 4 == 0 && godina % 100 != 0) || (godina %
   ↪  400 == 0);
}
```

Grananje je sada moguće realizovati uz korišćenje konstrukcije `else-if`.

```
int main() {
  // ucitavamo mesec i godinu
  int mesec, godina;
  cin >> mesec >> godina;

  // odredjujemo broj dana u tom mesecu
  int brojDana = 0;
  // januar, mart, maj, jul, avgust, oktobar, decembar
  if (mesec == 1 || mesec == 3 || mesec == 5 || mesec == 7 ||
      mesec == 8 || mesec == 10 || mesec == 12)
    brojDana = 31;
  // april, jun, septembar, novembar
  else if (mesec == 4 || mesec == 6 || mesec == 9 || mesec ==
   ↪  11)
    brojDana = 30;
  // februar
  else if (mesec == 2)
    brojDana = prestupna(godina) ? 29 : 28;

  // ispisujemo rezultat
  cout << brojDana << endl;

  return 0;
}
```

Naravno, moguće je upotrebiti i naredbu `switch-case`.

```
int main() {
  // ucitavamo mesec i godinu
  int mesec, godina;
  cin >> mesec >> godina;

  // odredjujemo broj dana u tom mesecu
  int brojDana = 0;
  switch(mesec) {
    // januar, mart, maj, jul, avgust, oktobar, decembar
  case 1: case 3: case 5: case 7: case 8: case 10: case 12:
    brojDana = 31;
    break;
    // april, jun, septembar, novembar
  case 4: case 6: case 9: case 11:
    brojDana = 30;
    break;
    // februar
  case 2:
    brojDana = prestupna(godina) ? 29 : 28;
    break;
  }

  // ispisujemo rezultat
  cout << brojDana << endl;
  return 0;
}
```

Ovakvi zadaci se mogu rešavati i bez grananja, tako što se svi elementi smeste u niz (ili mapu).

```
int main() {
  // ucitavamo mesec i godinu
  int mesec, godina;
  cin >> mesec >> godina;

  // broj dana u svakom mesecu
  int brojDanaUMesecu[] =
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

```
  // citamo broj dana
  int brojDana = brojDanaUMesecu[mesec];
  // posebno obradjujemo februar prestupnih godina
  if (mesec == 2 && prestupna(godina))
    brojDana++;

  // ispisujemo rezultat
  cout << brojDana << endl;

  return 0;
}
```

5.5.2 *Agregatno stanje vode (grananje na osnovu pripadnosti intervalu)*

Još jedan tipičan oblik upotrebe konstrukcije `else-if` je provera kojem od nekoliko nadovezanih intervala brojevne prave pripada data vrednost. Na primer, možemo na osnovu temperature vode određivati njeno agregatno stanje (smatramo da je ona u čvrstom stanju zaključno sa 0 stepeni, tečnom stanju do 100 stepeni (bez vrednosti 100) i gasovitom stanju počevši od 100 stepeni). Jedan način je da se pripadnost svakom od intervala ispita nezavisno.

```
int t; // Temperatura
cin >> t;

if (t <= 0)
  cout << "cvrsto" << endl;
if (t > 0 && t < 100)
  cout << "tecno" << endl;
if (t >= 100)
  cout << "gasovito" << endl;
```

Bolje rešenje dobija se ako se uslovi nadovežu, korišćenjem konstrukcije `else-if`.

```
int t; // Temperatura
cin >> t;

if (t <= 0)
  cout << "cvrsto" << endl;
else if (t < 100)
```

```
  cout << "tecno" << endl;
else
  cout << "gasovito" << endl;
```

5.5.3 *Uspeh učenika*

Na sličan način je, na primer, moguće odrediti ocenu na osnovu broja poena na ispitu ili uspeh učenika na osnovu prosečne ocene.

```
double prosek;  // prosek ocena ucenika
cin >> prosek;
if (prosek >= 4.5)
  cout << "odlican" << endl;
else if (prosek >= 3.5)
  cout << "vrlo dobar" << endl;
else if (prosek >= 2.5)
  cout << "dobar" << endl;
else if (prosek >= 2)
  cout << "dovoljan" << endl;
else
  cout << "nedovoljan" << endl;
```

5.5.4 *Kvadrant kom pripada tačka (hijerarhija ugnežđenih uslova)*

U nekim slučajevima je grananje hijerarhijsko. Slučajevi se klasifikuju prvo na osnovu nekog polaznog kriterijuma, onda se svaka klasa na osnovu nekog drugog kriterijuma dalje deli na potklase i tako sve dok se ne dođe do slučaja koji može direktno da se reši. Na primer, možemo na osnovu koordinata tačke u ravni određivati kom kvadrantu ili kojoj koordinatnoj osi tačka pripada. Prvo je moguće vršiti klasifikaciju na osnovu vrednosti jedne koordinate, pa zatim na osnovu druge.

```
int x, y;
cin >> x >> y;
if (x > 0) {
  if (y > 0) {
    cout << "1. kvadrant" << endl;
  } else if (y < 0) {
    cout << "4. kvadrant" << endl;
  } else {
```

```
    cout << "pozitivni deo x ose" << endl;
  }
} else if (x < 0) {
  if (y > 0) {
    cout << "2. kvadrant" << endl;
  } else if (y < 0) {
    cout << "3. kvadrant" << endl;
  } else {
    cout << "negativni deo x ose" << endl;
  }
} else {
  if (y > 0) {
    cout << "pozitivni deo y ose" << endl;
  } else if (y < 0) {
    cout << "negativni deo y ose" << endl;
  } else {
    cout << "koordinatni pocetak" << endl;
  }
}
```

5.5.5 *Poređenje datuma (leksikografsko poređenje torki iste dužine)*

Grananje se koristi i u leksikografskom poređenju torki iste dužine. Tipičan primer je poređenje datuma. Svaki datum se može predstaviti trojkom brojeva (dan, mesec, godina). Kada se porede dva datuma, prvo se porede godine, pa ako je godina u nekom datumu manja, onda je i taj datum manji (raniji). Ako su godine jednake, onda se porede meseci, pa ako su jednaki i meseci, tek onda se porede i dani. Na primer, ako je poznat datum rođenja osobe i današnji datum, možemo utvrditi da li je osoba punoletna. Ceo ovaj složeni uslov se može izraziti jednim velikim logičkim izrazom.

```
int d1, m1, g1; // datum rodjenja
int d2, m2, g2; // datum u kom se ispituje punoletstvo
cin >> d1 >> m1 >> g1
    >> d2 >> m2 >> g2;
if ((g2 > g1 + 18) ||
    (g2 == g1 + 18 && m2 > m1) ||
    (g2 == g1 + 18 && m2 == m1 && d2 >= d1))
  cout << "DA" << endl;
else
  cout << "NE" << endl;
```

Do rešenja je moguće doći i pomoću više ugnežđenih naredbi `if`.

```
bool punoletan;
if (g2 > g1 + 18)
  punoletan = true;
else if (g2 < g1 + 18)
  punoletan = false;
else { // g2 == g1 + 18
  if (m2 > m1)
    punoletan = true;
  else if (m2 < m1)
    punoletan = false;
  else { // m1 == m2
    if (d2 >= d1)
      punoletan = true;
    else
      punoletan = false;
    // punoletan = d2 >= d1;
  }
}
```

5.5.6 *Vrsta trougla na osnovu stranica*

Na kraju naglasimo da smo u prethodnom tekstu prikazali samo nekoliko tipičnih primera, a da je često potrebno osmisliti strukturu grananja ne prateći neki šablon, nego pažljivom analizom zahteva zadatka. Na primer, u narednom primeru se određuje vrsta trougla (jednakostranični, jednakokraki, raznostranični) na osnovu poznatih dužina njegovih stranica. Prvo se proverava da li date dužine zadovoljavaju nejednakost trougla, a zatim, redom, da li je trougao jednakostranični (za to je dovoljno da ima dva para jednakih stranica), zatim da li je jednakokraki (za to je dovoljno da ima jedan par jednakih stranica). Ako nijedan od tih uslova nije ispunjen, trougao je raznostranični. Obratimo pažnju na to da, zahvaljujući konstrukciji `else if`, prilikom ispitivanja da li je trougao jednakokraki, već znamo da ne postoje dva para jednakih stranica, pa je dovoljno ispitati da li postoji bar jedan par jednakih stranica (ako postoji bar jedan, pošto ne postoje dva, tada postoji tačno jedan par). Slično, uslov da su svi parovi stranica različiti ne treba proveravati, jer ako ne postoji nijedan par jednakih stranica, taj uslov automatski važi.

```
if (a + b > c && a + c > b && b + c > a) {
  if (a == b && b == c /* && a == c */)
    cout << "jednakostranicni" << endl;
  else if (a == b || b == c || a == c)
    cout << "jednakokraki" << endl;
  else
    cout << "raznostranicni" << endl;
} else
  cout << "trougao ne postoji" << endl;
```

6. Petlje

Petlje (*ciklusi* ili *repetitivne naredbe*) uzrokuju da se određena naredba (ili grupa naredbi) izvršava više puta (sve dok je neki logički uslov ispunjen).

6.1 Petlja `while`

Petlja `while` ima sledeći opšti oblik:

```
while (<izraz>)
   <naredba>
```

U petlji `while` ispituje se vrednost izraza `izraz` i ako ona ima istinitosnu vrednost *tačno* (tj. vrednost različita od nule ako je `izraz` brojevnog tipa), izvršava se `naredba` (što je ili pojedinačna naredba ili blok naredbi). Zatim se uslov `izraz` iznova proverava i sve se ponavlja dok mu istinitosna vrednost ne postane *netačno* (tj. vrednost nula ako je `izraz` brojevnog tipa). Tada se izlazi iz petlje i nastavlja sa izvršavanjem prve sledeće naredbe u programu. Dakle, ako u telu petlje nema prekida (naredbi `break` ili `return`), možemo biti sigurni da nakon završetka petlje njen uslov neće biti ispunjen.
Ukoliko iza `while` sledi samo jedna naredba, onda, kao i obično, nema potrebe za vitičastim zagradama. Na primer (naredni kôd ispisuje brojeve od 0 do 9):

```
int i = 0;
while (i < 10)
  cout << i++ << endl;
```

Sledeća petlja `while` se izvršava beskonačno:

```
while (true)
  cout << "Zdravo" << endl;
```

Uslov petlje se proverava pre prvog izvršavanja tela (kažemo da je ovo petlja sa proverom uslova na početku), pa, ako uslov nije ispunjen, moguće je da se telo nijednom ne izvrši.

6.2 Petlja `for`

Petlja `for` ima sledeći opšti oblik:

```
for (<izraz1>; <izraz2>; <izraz3>)
  <naredba>
```

Komponente `<izraz1>`, `<izraz2>` i `<izraz3>` su izrazi.[1] Obično su `<izraz1>` i `<izraz3>` izrazi dodele ili inkrementiranja, a `<izraz2>` je relacijski izraz. Izraz `<izraz1>` se obično naziva *inicijalizacija* i koristi se za postavljanje početnih vrednosti promenljivih, izraz `<izraz2>` je *uslov* ostanka u petlji, a `<izraz3>` je *korak* i njime se menjaju vrednosti relevantnih promenljivih. Naredba `<naredba>` naziva se *telo* petlje. Inicijalizacija (izraz `<izraz1>`) se izračunava samo jednom, na početku izvršavanja petlje. Često se u njoj deklariše tzv. brojačka promenljiva koja je lokalna za tu petlju (vrednost joj se može koristiti samo u sklopu petlje, uključujući i njeno telo). Petlja se izvršava sve dok uslov (izraz `<izraz2>`) ima istinitosnu vrednost *tačno*, a korak (izraz `<izraz3>`) izračunava se na kraju svakog prolaska kroz petlju. Redosled izvršavanja je, dakle, oblika: *inicijalizacija*, *uslov*, *telo*, *korak*, *uslov*, *telo*, *korak*, ..., *uslov*, *telo*, *korak*, *uslov*, pri čemu je *uslov* ispunjen svaki put osim poslednjeg. Dakle, gore navedena opšta forma petlje `for` ekvivalentna je konstrukciji koja koristi petlju `while`[2]:

```
<izraz1>;
while (<izraz2>) {
  <naredba>
  <izraz3>;
}
```

[1] Zapravo, `<izraz1>` je izraz ili deklaracija sa inicijalizacijom.

[2] Ova ekvivalencija važi uz dva izuzetka. Ako telo petlje sadrži naredbu `continue` (o kojoj će biti reči u poglavlju 6.4), u petlji `for` se nakon nje izvršava korak, dok se u navedenoj konstrukciji sa petljom `while` korak preskače. Takođe, promenljiva deklarisana u inicijalizaciji petlje `for` može se koristiti samo u petlji, dok je u navedenoj konstrukciji ona dostupna i nakon petlje `while`.

Petlja `for` se obično koristi kada je potrebno izvršiti jednostavno početno dodeljivanje vrednosti promenljivim i jednostavno ih menjati sve dok je ispunjen zadati uslov (pri čemu su i početno dodeljivanje i uslov i izmene lako vidljivi u definiciji petlje). To ilustruje sledeća tipična forma petlje `for`:

```
for (int i = 0; i < n; i++)
   cout << i << endl;
```

Naredba u telu petlje se ponavlja `n` puta, pri čemu promenljiva `i` redom uzima vrednosti od `0` do `n-1` (ona se naziva *brojačka promenljiva*). U narednoj petlji se telo takođe ponavlja `n` puta, pri čemu promenljiva `i` uzima vrednosti od `1` do `n`.

```
for (int i = 1; i <= n; i++)
   cout << i << endl;
```

Umesto da se vrednost brojačke promenljive uvećava za 1, čest je slučaj da se uveća i za neku drugu vrednost, čime se dobija nabrajanje elemenata nekog aritmetičkog niza. Na primer, naredni program ispisuje vrednosti $5, 10, 15, \ldots, 100$.

```
for (int i = 5; i <= 100; i += 5)
   cout << i << endl;
```

Korišćenjem petlje `for` moguće je nabrojati i elemente geometrijskog niza. Na primer, naredni program ispisuje stepene broja 2, krenuvši od 1, pa sve do 1024. To se postiže tako što se u svakom koraku tekuća vrednost brojačke promenljive množi sa 2.

```
for (int i = 1; i <= 1024; i *= 2)
   cout << i << endl;
```

Često je potrebno da se nabroje i ravnomerno razmaknute tačke unutar nekog intervala. Na primer, za potrebe crtanja grafika funkcije želimo da izračunamo njenu vrednost u $n \geq 2$ ravnomerno razmaknutih tačaka intervala $[a, b]$. Prva tačka je a, razmak između dve tačke je $d = \frac{b-a}{n-1}$, pa petlja može imati sledeći oblik

```
double d = (b - a) / (n - 1);
for (double x = a; x <= b; x += d)
   ...
```

Međutim, usled problema sa tačnošću zapisa realnih brojeva, može se desiti da se ovakvom petljom nabroji tačka više ili tačka manje od onoga što je očekivano (jer bi, na primer, moglo da se desi da vrednost `x` malko premaši `b` onda kada očekujemo da one budu jednake). Stoga je uvek bolje koristiti celobrojne brojačke promenljive, kao što je ilustrovano u narednom kodu.

```
double d = (b - a) / (n - 1);
double x = a;
for (int i = 0; i < n; i++) {
    ...
    x += d;
}
```

Primetimo da se `i` u ovom kodu, pošto se vrednost `x` u svakom koraku uvećava za `d`, greške zaokruživanja postepeno sabiraju. Zato je još bolje svaku tačku izračunati direktno, kao `a + i * d`.
Bilo koji od izraza `<izraz1>`, `<izraz2>`, `<izraz3>` u petlji `for` može biti izostavljen, ali simboli `;` i tada moraju biti navedeni. Ukoliko je izostavljen izraz `<izraz2>`, smatra se da je njegova istinitosna vrednost *tačno*. Na primer, sledeća petlja `for` se izvršava beskonačno (ako u bloku naredbi koji ovde nije naveden nema neke naredbe koja prekida izvršavanje, na primer, `break` ili `return`):

```
for (;;)
  <naredba>
```

Ovo se često koristi tako što se uslov petlje proveri negde u sklopu njenog tela i ako nije ispunjen, naredbom `break` se prekine izvršavanje petlje (o čemu će biti više reči u poglavlju 6.4). Isti efekat postiže se i naredbom `while (true)`, koja se često smatra čitljivijom.[3]

```
while (true)
  <naredba>
```

Ako je potrebno da neki od izraza `<izraz1>`, `<izraz2>`, `<izraz3>` objedini više izraza, može se koristiti operator `,`.

[3] Starije verzije izvesnog kompilatora za programski jezik C su `while (1)` prepoznavale kao beskonačnu petlju i izdavale upozorenje korisniku, dok za `for(;;)` to nisu radile. Da bi izbegli upozorenje, programeri su krenuli da koriste oblik `for(;;)` i ta praksa je ostala i do danas.

```
for (int i = 0, j = 10; i < j; i++, j--)
   cout << "i = " << i << ", j = " << j << endl;
```

Prethodni kôd ispisuje

```
i = 0, j = 10
i = 1, j = 9
i = 2, j = 8
i = 3, j = 7
i = 4, j = 6
```

U okviru inicijalizacije petlje `for` nije moguće istovremeno deklarisati dve promenljive različitog tipa.
Sledeći program, koji ispisuje tablicu množenja, ilustruje dvostruku petlju `for`:

```
  int n = 3;
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++)
      cout << i << " * " << j << " = " << i*j << "   ";
    cout << endl;
  }
```

```
1 * 1 = 1   1 * 2 = 2   1 * 3 = 3
2 * 1 = 2   2 * 2 = 4   2 * 3 = 6
3 * 1 = 3   3 * 2 = 6   3 * 3 = 9
```

Još jedan oblik petlje `for` u jeziku C++ je petlja kojom se nabrajaju redom svi elementi neke kolekcije (na primer, vektora). Narednom petljom se nabrajaju svi elementi vektora `temperature`. Promenljiva `t` ovaj put nije brojačka promenljiva, jer ne sadrži indekse elemenata vektora, već je promenljiva koja će sadržati elemente vektora (u svakoj iteraciji petlje po jedan, redom, od prvog do poslednjeg).

```
vector<double> temperature;
for (double t : temperature)
   cout << t << endl;
```

6.3 Petlja do-while

Petlja `do-while` ima sledeći opšti oblik:

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

Telo (blok naredbi `naredbe`) navedeno između vitičastih zagrada se izvršava i onda se izračunava uslov (izraz `izraz`). Ako je on tačan, telo se izvršava ponovo i to se nastavlja sve dok izraz `izraz` ne bude imao istinitosnu vrednost *netačno*.
Za razliku od petlje `while`, naredbe u bloku ove petlje se uvek izvršavaju barem jednom. Kažemo da je ovo petlja sa proverom uslova na kraju.
Na primer, naredni kôd učitava ocenu sve dok se ne unese ispravno (tj. dok se ne unese vrednost između 1 i 5).

```
int ocena;
do {
   cout << "Unesite ocenu: ";
   cin >> ocena;
} while (ocena < 1 || ocena > 5);
```

Korišćenjem petlje `do-while` umesto `while` je osigurano da će se prvo učitati ocena, pa tek onda proveravati njena ispravnost.

6.4 Naredbe break i continue

U nekim situacijama pogodno je napustiti petlju ne zbog toga što nije ispunjen uslov petlje, već iz nekog drugog razloga. To je moguće postići naredbom `break` kojom se izlazi iz tekuće petlje (ili naredbe `switch`).[4] Na primer, naredna petlja prolazi kroz elemente niza i obrađuje ih, ali se prekida ako se naiđe na neki negativni element:

```
for (i = 0; i < n; i++) {
  if (a[i] <  0)
    break;
  ...
}
```

[4] Naredbom `break` ne izlazi se iz bloka naredbe `if`.

Korišćenjem naredbe `break` se narušava strukturiranost koda i to može da oteža njegovu analizu (na primer, analizu ispravnosti ili analizu složenosti). Na primer, ne možemo više da tvrdimo da nakon petlje uslov petlje neće biti ispunjen. U nekim situacijama, korišćenje naredbe `break` može da dovede do kraćeg koda, ali kôd koji koristi naredbu `break` uvek se može napisati i bez nje. U datom primeru, odgovarajući alternativni kôd je, na primer:

```
for(i = 0; i < n && a[i] >= 0; i++)
  ...
```

Česta upotreba naredbe `break` je u petljama u kojima se uslov ne proverava ni na početku, ni na kraju, već u sredini. Tada se koristi neki zapis beskonačne petlje (na primer, `while (true)`, a uslov se proverava u sredini, naredbom `if`). Na primer, naredni program učitava brojeve sve dok se ne unese 0 i obrađuje ih (pri čemu se uneta nula ne obrađuje).

```
while (true) {
   int x;
   cin >> x;
   if (x == 0) break;
   // obrada elementa x
   ...
}
```

Naredbom `continue` se prelazi na sledeću iteraciju u petlji. Na primer,

```
for(i = 0; i < n; i++) {
  if (i % 10 == 0)
    continue;  // preskoci brojeve deljive sa 10
  ...
}
```

Slično kao za naredbu `break`, korišćenjem naredbe `continue` se narušava strukturiranost koda, ali se može dobiti kraći kôd. Kôd koji koristi naredbu `continue` uvek se može napisati i bez nje. U datom primeru, odgovarajući alternativni kôd je, na primer:

```
for (i = 0; i < n; i++)
  if (i % 10 != 0)  { // samo brojevi koji nisu deljivi sa 10
    ...
```

```
    }
```

U slučaju ugnežđenih petlji, naredbe `break` i `continue` imaju dejstvo samo na unutrašnju petlju. Tako, na primer, fragment

```
for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
        if (i + j > 2) break;
        cout << i << " " << j << "  ";
    }
}
```

ispisuje

```
0 0  0 1  0 2  1 0  1 1  2 0
```

6.5 Osnovni iterativni algoritmi

Jedan od osnovnih mehanizama programiranja i osnovnih tehnika konstrukcije algoritama je *iteracija*, koja podrazumeva izračunavanje rezultata postupnom izmenom vrednosti neke promenljive. Vrednost promenljive se na početku izračunavanja inicijalizuje, a zatim se postupno menja, kroz niz koraka, sve dok se ne dostigne željeni rezultat. Pošto se koraci istog oblika ponavljaju više puta, za implementaciju iterativnih postupaka se po pravilu koriste petlje, pa se termin iteracija često identifikuje sa upotrebom petlji. U nastavku će biti prikazano nekoliko tipičnih algoritama ovog tipa. Obrađivaćemo konačne *serije*[5] elemenata, oblika $x_0, \ldots, x_{n-1}$, koje se sastoje od elemenata x_i, koji mogu biti učitavani sa standardnog ulaza, iz nekog niza ili vektora, računati po nekom pravilu i slično. Većina osnovnih algoritama za obradu serija je veoma jednostavna, međutim, njihovim kombinovanjem dobijaju se složeniji algoritmi kojima se može rešiti veliki broj praktičnih problema.

Kada su elementi smešteni u niz, vektor ili neku drugu sličnu kolekciju, tada se umesto ručne implementacije ovih algoritama mogu upotrebiti i bibliotečke funkcije. Naime, standardne biblioteke savremenih programskih jezika obično nude korisnicima funkcije kojima se implementiraju ovi algoritmi. Ipak, mi ćemo iz metodičkih razloga sve ove algoritme implementirati samostalno (a njihove bibliotečke implementacije ćemo prikazati u poglavlju 9 posvećenom pregledu standardne biblioteke).

[5] Umesto serije možemo reći i sekvence, ali namerno ne koristimo termin *niz* ili *lista*, jer se ti termini koriste za specifične strukture podataka.

6.5.1 *Sabiranje, prebrojavanje, množenje*

Sabiranje elemenata neke serije brojeva se može vršiti tako što se zbir inicijalizuje na nulu, a zatim se u svakom koraku zbir uvećava za tekući element serije. Na primer, program koji izračunava obim trougla učitavajući dužine njegovih stranica jednu po jednu se može napisati i na sledeći način.

```
int obim = 0;
int stranica;
cin >> stranica;
obim = obim + stranica;
cin >> stranica;
obim = obim + stranica;
cin >> stranica;
obim = obim + stranica;
cout << obim << endl;
```

Ako izvršimo prethodni program korak-po-korak, možemo primetiti da u prvom koraku promenljiva `obim` ima vrednost dužine prve stranice, u drugom zbira dužina prve i druge stranice, a da u trećem zbira dužina sve tri stranice.
Umesto naredbe `obim = obim + stranica`, možemo upotrebiti i operator `+=` koji služi za uvećavanje vrednosti promenljive tj. ovu naredbu možemo zapisati i kao `obim += stranica`.
Ponavljanje naredbi se, naravno, može ostvariti i uz pomoć petlji. Pošto unapred znamo potreban broj koraka, uobičajeno je da se upotrebi petlja `for`.

```
int obim = 0;
for (int i = 0; i < 3; i++) {
   int stranica;
   cin >> stranica;
   obim += stranica;
}
cout << obim << endl;
```

Napokon, ovaj program možemo veoma jednostavno uopštiti tako da radi i za mnogouglove.

```
int n;
cin >> n;
int obim = 0;
```

```
for (int i = 0; i < n; i++) {
   int stranica;
   cin >> stranica;
   obim += stranica;
}
cout << obim << endl;
```

Kažemo da smo u ovom programu upotrebili *algoritam sabiranja serije*. Zasniva se na tome da se promenljiva koja treba da sadrži konačan rezultat inicijalizuje na nulu, a zatim da se u svakom koraku petlje ažurira i nova vrednost joj se izračuna sabiranjem njene trenutne vrednosti i tekućeg elementa serije koja se obrađuje. Primetimo da i pre petlje i nakon svakog izvršavanja tela petlje i nakon petlje, promenljiva `obim` sadrži tačno zbir svih do tada učitanih elemenata serije tj. svih do tada učitanih dužina stranica mnogougla (svojstvo koje važi pre petlje, tokom petlje i nakon petlje naziva se *invarijanta petlje* i obično garantuje korektnost algoritma).
Elementi koji se sabiraju ne moraju da se učitavaju sa standardnog ulaza, već mogu biti određeni i na neki drugi način. Na primer, pod pretpostavkom da vektor `cene` sadrži cene svih kupljenih proizvoda, naredni program izračunava njihovu ukupnu cenu.

```
vector<double> cene{153.99, 49.00, 213.50};
double ukupno = 0.0;
for (double cena : cene)
   ukupno += cena;
cout << ukupno << endl;
```

Na način veoma sličan sabiranju možemo da prebrojimo elemente neke serije (u pitanju je, dakle, *algoritam prebrojavanja serije*). Broj elemenata inicijalizujemo na nulu, a zatim se u svakom koraku petlje taj broj uvećava za 1. Na primer, naredni program učitava brojeve sve dok se ne učita vrednost `0` i određuje broj tako učitanih elemenata.

```
int broj = 0;
int x;
cin >> x;
while (x != 0) {
   broj = broj + 1;
   cin >> x;
}
cout << broj << endl;
```

Uvećanje brojača je moguće izvršiti i operatorom `++`, čiji je efekat da se vrednost promenljive uveća za 1, tj. umesto naredbe `broj = broj + 1;`, moguće je kraće pisati

`broj++;` ili `++broj;`.
Izračunavanje zbira često omogućava i izračunavanje proseka tj. aritmetičke sredine. Program može da učitava ocene sve dok se ne unese broj koji ne predstavlja ispravnu ocenu (nije između 1 i 5) i da se tada na ekran ispiše prosek svih učitanih ocena. Pošto broj ocena nije unapred poznat, ponovo koristimo petlju `while` i istovremeno izračunavamo i zbir i broj učitanih ocena. Jednostavnosti radi, pretpostavićemo da će uvek biti uneta bar jedna ocena (u suprotnom, izračunavanje proseka nema smisla i prosek nije definisan).

```
int broj = 0;
int zbir = 0;
int ocena;
cin >> ocena;
while (1 <= ocena && ocena <= 5) {
   broj++;
   zbir += ocena;
   cin >> ocena;
}
cout << (double)zbir / (double)broj << endl;
```

Primetimo da smo i zbir i broj smeštali u celobrojne promenljive, što znači da bi se primenom operatora `/` na njih izvršilo njihovo celobrojno deljenje, a ne deljenje brojeva u pokretnom zarezu. Zato je neophodno izvršiti eksplicitnu konverziju tipa i bar jedan od operanada konvertovati u realni tip (ovde je upotrebljen tip `double`). Oba operanda su konvertovana samo radi simetrije (to nije neophodno raditi).
Primetimo da smo u prethodnom programu ocene učitavali na dva mesta u programu: jednom pre petlje i jednom na kraju petlje. Kada je u pitanju ovako kratka i jednostavna naredba, njeno ponavljanje ne predstavlja problem, međutim, da je u pitanju bio neki komplikovaniji fragment koda, bilo bi poželjno izbeći ponavljanje. Jedan način da se to uradi je da se uslov ne proverava na početku, već na sredini petlje.

```
int broj = 0;
int zbir = 0;
while (true) {
   int ocena;
   cin >> ocena;
   if (ocena < 1 || ocena > 5)
      break;
   broj++;
   zbir += ocena;
```

```
}
cout << (double)zbir / (double)broj << endl;
```

Algoritam množenja serije brojeva se ostvaruje na način veoma sličan sabiranju. Proizvod se mora inicijalizovati na 1 (ne na 0) i zatim u svakom koraku množiti tekućim elementom serije. Na primer, narednim programom učitavamo dužine jedne po jedne od tri stranice kvadra i izračunavamo zapreminu tog kvadra.

```
int zapremina = 1;
for (int i = 0; i < 3; i++) {
   int stranica;
   cin >> stranica;
   zapremina *= stranica;
}
cout << zapremina << endl;
```

Primetimo veliku sličnost algoritma izračunavanja zbira i algoritma izračunavanja proizvoda. Razlika je to što se u inicijalizaciji zbir inicijalizuje na nulu, a proizvod na jedinicu i to što se tokom ažuriranja zbira koristi sabiranje tj. operator +, a tokom ažuriranja proizvoda koristi množenje tj. operator $\star$. Naime, za operaciju sabiranja *neutralni element* je broj 0 (jer za svako x važi $x + 0 = 0 + x = x$), dok je za operaciju množenja neutralni element broj 1 (jer za svako x važi $x \cdot 1 = 1 \cdot x = x$). Ako serija nije prazna, umesto inicijalizacije na neutralni element, moguće je inicijalizovati rezultat na prvi element serije, no obično je rešenje sa neutralnim elementom elegantnije (kada je operacija takva da neutral postoji).
Ni u algoritmu množenja serija brojeva koja se obrađuje, naravno, ne mora da se učitava sa standardnog ulaza. Na primer, vrednost stepena x^n se može izračunati tako što se početna vrednost 1 pomnoži n puta brojem x (u pitanju je proizvod konstantne, n-točlane serije $x, x, \ldots, x$).

```
int n;
cin >> n;
double x;
cin >> x;
double stepen = 1;
for (int i = 0; i < n; i++)
   stepen *= x;
cout << stepen << endl;
```

Slično se može izračunati i vrednost $n!$, kao proizvod $1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$. U tom slučaju se vrednosti brojačke promenljive koriste kao elementi serije koja se množi.

```
int n;
cin >> n;
int faktorijel = 1;
for (int i = 1; i <= n; i++)
   faktorijel *= i;
cout << faktorijel << endl;
```

Pošto proizvod često može da bude veliki i kada su brojevi koji se množe relativno mali, u ovakvim situacijama treba voditi računa o tome da već za male vrednosti n rezultat može biti netačan usled prekoračenja dozvoljenog opsega vrednosti promenljive u kojoj se čuva proizvod.

6.5.2 *Minimum i maksimum*

Razmotrimo sada *algoritam za određivanje minimuma ili maksimuma serije brojeva* (tj. najmanjeg ili najvećeg broja u seriji). Jedan način da se to uradi je korišćenje bibliotečke funkcije `min` za određivanje minimuma (ili `max` za određivanje maksimuma). Na primer, `min({a, b, c})` određuje najmanji od brojeva `a`, `b` i `c`. Da bismo mogli da rešavamo i srodne probleme, izučićemo kako se implementira algoritam za određivanje minimuma ili maksimuma proizvoljne serije brojeva.
Minimum 3 broja se može odrediti ugnežđenim naredbama `if`.

```
int minimum;
if (a <= b) {
   if (a <= c)
      minimum = a;
   else
      minimum = c;
} else {
   if (b <= c)
      minimum = b;
   else
      minimum = c;
}
```

Ovo rešenje je komplikovano i teško se uopštava na više brojeva. Stoga je poželjno koristiti iterativni algoritam, koji ćemo u nastavku izvesti. Ako upotrebimo bibliotečku funkciju `min` za određivanje minimuma dva broja, onda minimum četiri broja možemo odrediti njenom uzastopnom primenom.

```
int minimum = min(min(min(a, b), c), d);
```

Prisetimo se da smo i zbir više brojeva računali tako što smo iterativno primenjivali operaciju sabiranja dva broja. Tako je zbir `a + b + c + d` računat kao `((a + b) + c) + d`, tj. kao `zbir(zbir(zbir(a, b), c), d)`. Ovo nam ukazuje na to da i minimum više brojeva možemo izračunati na sličan način na koji smo računali zbir – minimum možemo inicijalizovati na prvi element i onda u svakom narednom koraku ažurirati na manju od vrednosti dosadašnjeg minimuma i vrednosti tekućeg elementa.

```
int minimum = a;
minimum = min(minimum, b);
minimum = min(minimum, c);
minimum = min(minimum, d);
cout << minimum << endl;
```

Ako se elementi nalaze u nizu ili se učitavaju sa ulaza, onda možemo da upotrebimo i petlje, što, naravno, omogućava da se isti postupak primeni i na serije koje su proizvoljne dužine.[6]

```
vector<int> niz{a, b, c, d};
int minimum = niz[0];
for (int i = 1; i < niz.size(); i++)
    minimum = min(minimum, niz[i]);
cout << minimum << endl;
```

Minimum se ažurira samo ako je vrednost trenutnog elementa koji se obrađuje manja od dotadašnjeg minimuma. Tako se upotreba funkcije `min` može zameniti grananjem.

```
vector<int> niz{a, b, c, d};
int minimum = niz[0];
for (int i = 1; i < niz.size(); i++)
    if (niz[i] < minimum)
       minimum = niz[i];
cout << minimum << endl;
```

[6] Metoda `size` vraća vrednost neoznačenog tipa (`size_t`), pa prilikom poređenja `i < niz.size()` neki prevodioci (npr. `g++` sa opcijom `-Wall`) daju upozorenje o poređenju označene i neoznačene vrednosti. U ovakvim petljama to je bezbedno, jer promenljiva `i` nikada nije negativna. Upozorenje se može izbeći ako se brojač deklariše kao `size_t i` ili ako se dužina konvertuje u označen tip, npr. `i < (int)niz.size()`.

Primetimo da se prvi element obrađuje drugačije od ostalih. Kod algoritama sabiranja i množenja rezultujuću promenljivu smo inicijalizovali na neutralni element odgovarajuće operacije i tako smo postigli da je rezultat definisan i u slučaju da je serija prazna, kao i da se svi elementi serije obrađuju na isti način. Pitanje je da li nešto slično možemo da uradimo i za minimum i maksimum. Potrebno je da pronađemo vrednost `x` tako da za bilo koji broj `a` važi da je `min(x, a) = min(a, x) = a`, tj. broj koji nije manji ni od jednog drugog broja. To može da bude plus beskonačno ($+\infty$), ali tu vrednost ne možemo da zapišemo kao podatak celobrojnog tipa (tipovi brojeva u pokretnom zarezu dopuštaju beskonačnu vrednost, `numeric_limits<double>::infinity()`, koja je definisana u zaglavlju `<limits>`). Umesto toga, možemo upotrebiti najveći broj koji se može zapisati u opsegu tipa `int`. Taj broj se može u jeziku C++ dobiti izrazom `numeric_limits<int>::max()` definisanim u zaglavlju `<limits>` (a u jeziku C izrazom `INT_MAX` definisanim u zaglavlju `<climits>`).

```
vector<int> niz;
int minimum = numeric_limits<int>::max();
for (int a : niz)
    if (a < minimum)
       minimum = a;
cout << minimum << endl;
```

Neutralni element za operaciju maksimuma je $-\infty$, koju za tipove brojeva u pokretnom zarezu možemo zapisati kao `-numeric_limits<double>::infinity()`. Kod celobrojnih tipova ta vrednost se ne može zapisati, ali umesto nje možemo upotrebiti najmanji mogući ceo broj, koji daje izraz `numeric_limits<int>::min()`. Sa druge strane, ako znamo da među brojevima čiji ćemo maksimum računati neće biti negativnih, onda za početnu vrednost možemo uzeti nulu.

Nekada nas ne zanima vrednost maksimuma (ili minimuma), već pozicija na kojoj se taj maksimum (ili minimum) nalazi. Da bismo to odredili, potrebno je da uz tekuću vrednost maksimuma (ili minimuma) pamtimo i tekuću vrednost njegove pozicije. Na primer, ako znamo vrednost dobijenog džeparca tokom svih 5 dana u nekoj nedelji, možemo odrediti dan u kom je dobijen najveći džeparac.

```
int maksDzeparac = numeric_limits<int>::min();
int maksDan;
for (int dan = 1; dan <= 5; dan++) {
   int dzeparac;
   cin >> dzeparac;
   if (dzeparac > maksDzeparac) {
      maksDzeparac = dzeparac;
```

```
        maksDan = dan;
    }
}
cout << maksDan << endl;
```

Nekada nas ne zanima samo najveća, već nekoliko najvećih vrednosti. Kada je broj vrednosti koje tražimo veliki, potrebno je koristiti neke malo komplikovanije algoritme. Međutim, dve najveće vrednosti možemo odrediti jednostavnom modifikacijom algoritma za određivanje maksimuma (ili minimuma). Pritom je bitno precizirati šta se dešava kada su dozvoljene ponovljene vrednosti. Na primer, da li su dve najveće vrednosti u nizu `83 94 94` vrednosti `94` i `94` ili vrednosti `83` i `94`. Pretpostavimo da očekujemo prvi odgovor (ako bi ovo bili poeni studenata, tražili bismo poene dva najbolja studenta). Održavamo dve promenljive: `maks1` čuva najveću, a `maks2` drugu po veličini od vrednosti koje su do tog trenutka obrađene. Obe promenljive se mogu inicijalizovati na vrednost $-\infty$ (tj. najmanju celobrojnu vrednost). Obrađujemo jedan po jedan element niza.

- Ako trenutni element nije manji od dosadašnje najveće vrednosti `maks1`, tada je on najveća vrednost, dok je dotadašnja najveća vrednost `maks1` sada druga po veličini.
- U suprotnom, element ne može biti najveći, ali može biti drugi po veličini. Zato ga poredimo sa vrednošću `maks2`, pa ažuriramo tu vrednost, ako je tekući element niza veći od nje.

```
int maks1, maks2;
maks1 = maks2 = numeric_limits<int>::min();
for (int a : niz) {
    if (a >= maks1) {
        maks2 = maks1;
        maks1 = a;
    } else if (a > maks2) {
        maks2 = a;
    }
}
```

6.5.3 *Linearna pretraga*

Pretragom možemo proveriti da li u seriji postoji element koji zadovoljava neki uslov (na primer, da li među brojevima postoji neki broj koji je paran ili da li među rečima postoji neka koja počinje samoglasnikom). Veoma slični problemi tome su da se proveri da li svi elementi serije zadovoljavaju neki uslov (na primer, da li su svi brojevi pozitivni),

da li postoji neki element koji ne zadovoljava uslov ili da li nijedan od elemenata ne zadovoljava uslov. Moguće je određivati i na kojoj se poziciji nalazi prvi element koji zadovoljava uslov, poslednji element koji zadovoljava uslov i slično.
Ako nemamo nikakve dodatne pretpostavke o redosledu elemenata serije (na primer, ne znamo da li su elementi zadati u nekom sortiranom redosledu), tada primenjujemo algoritam *linearne pretrage*, koji podrazumeva da seriju analiziramo redom, jedan po jedan element.
Provera da li postoji element neke serije koji zadovoljava dati uslov zahteva izračunavanje logičke disjunkcije. Na primer, da bismo proverili da li je neki od tri filma u internet bazi filmova ocenjen kao „izvanredan“ (ima ocenu veću od 9,00), treba da izračunamo disjunkciju tri uslova.

```
double ocena1 = 9.18;
double ocena2 = 10.00;
double ocena3 = 9.56;
if (ocena1 > 9.00 || ocena2 > 9.00 || ocena3 > 9.00)
   cout << "Barem jedan od tri filma je izvanredan" << endl;
```

Poželjno je uopštiti ovo rešenje na seriju sa proizvoljnim brojem elemenata. Na primer, pretpostavimo da je dat niz `ocene` koji sadrži ocene filmova i da u petlji želimo da obrađujemo elemente ovog niza. Primetimo sličnost sa svim prethodnim operacijama (sabiranjem, množenjem, minimumom/maksimumom): ponovo imamo binarnu, asocijativnu operaciju (ovaj put je to disjunkcija) koja se primenjuje na seriju elemenata. Ponovo možemo krenuti od neutralne vrednosti, a zatim iterativno obrađivati jedan po jedan element. Rezultat je ovaj put tipa `bool`.

```
bool postoji_izvanredan = false;
postoji_izvanredan = postoji_izvanredan || ocena1 > 9.00;
postoji_izvanredan = postoji_izvanredan || ocena2 > 9.00;
postoji_izvanredan = postoji_izvanredan || ocena3 > 9.00;
if (postoji_izvanredan)
   cout << "Barem jedan od tri filma je izvanredan" << endl;
```

Uopštenje je sada neposredno.

```
bool postoji_izvanredan = false;
for (double ocena : ocene)
    postoji_izvanredan = postoji_izvanredan || ocena > 9.00;
...
```

U svakom koraku petlje važi da promenljiva `postoji_izvanredan` ima vrednost `false` ako i samo ako nijedan do tog trenutka obrađen film nije bio izvanredan. Promenljiva, dakle, započinje sa vrednošću `false` i ta vrednost se menja samo ako se pojavi neki film koji je izvanredan. Stoga se isti algoritam može implementirati i malo jednostavnije.

```
// do sada nismo nasli nijednog koji je izvanredan
bool postoji_izvanredan = false;
for (double ocena : ocene)
    // nasli smo film koji je izvanredan
    if (ocena > 9.00)
       postoji_izvanredan = true;
...
```

Čim se pronađe jedan film koji je izvanredan, nema potrebe proveravati dalje, već se petlja može odmah prekinuti (ovo odgovara skraćenom izračunavanju operatora `||`).

```
bool postoji_izvanredan = false;
for (double ocena : ocene)
    if (ocena > 9.00) {
       postoji_izvanredan = true;
       break;
    }
...
```

Provera da li svi elementi zadovoljavaju dato svojstvo je dualna. Rezultat se inicijalizuje na vrednost `true`, što je neutral za konjunkciju (za sada nije pronađen element koji ne zadovoljava uslov), a zatim se menja na `false` kada se (ako se) naiđe na element koji ne zadovoljava traženo svojstvo. Na primer, provera da li su svi dati filmovi izvanredni može se uraditi na sledeći način.

```
// do sada nismo nasli nijedan koji nije izvanredan
bool svi_izvanredni = true;
for (double ocena : ocene)
    // nasli smo jedan koji nije izvanredan
    if (ocena <= 9.00) {
       svi_izvanredni = false;
       break;
    }
```

```
...
```

Na sličan način možemo odrediti prvu i poslednju poziciju na kojoj se javlja element koji zadovoljava dato svojstvo. Ako su elementi smešteni u vektor (ili niz), tada za određivanje poslednje pozicije obradu elemenata možemo vršiti zdesna nalevo. Međutim, ako se elementi učitavaju i ne želimo da ih sve istovremeno pamtimo, tada moramo da ih obrađujemo redom, sleva nadesno. Na primer, naredni kôd određuje poziciju prve i poslednje učitane nule (brojanje pozicija kreće od 0).

```
int pozicijaPrveNule = -1;
int pozicijaPoslednjeNule = -1;
for (int i = 0; i < n; i++) {
   int x;
   cin >> x; // ucitavamo novi element serije
   if (x == 0) {
      if (pozicijaPrveNule == -1) {
         // ovo je prva ucitana nula
         pozicijaPrveNule = i;
      }
      // ovo je poslednja do sada ucitana nula
      pozicijaPoslednjeNule = i;
   }
}
cout << pozicijaPrveNule << " " << pozicijaPoslednjeNule <<
 ↪  endl;
```

Mnogi algoritmi su zasnovani na algoritmu linearne pretrage. Na primer, provera da li je prirodan broj n prost se može svesti na ispitivanje da li je deljiv nekim brojem iz intervala $[2, n-1]$.

```
int n;
cin >> n;
bool prost = true;
// prosti brojevi su prirodni brojevi veci od 1: broj 1 nije
// ni prost ni slozen, a 0 i negativni brojevi nisu prosti
if (n <= 1) prost = false;
// proveravamo sve moguce delioce iz intervala [2, n-1]
for (int d = 2; d < n; d++)
   if (n % d == 0)
      prost = false;
```

```
cout << (prost ? "prost" : "nije prost") << endl;
```

Delioci broja n iz intervala $[2, n-1]$ uvek se javljaju u parovima: za svaki takav delilac d koji nije manji od vrednosti $\sqrt{n}$, i broj n/d je delilac iz tog intervala i on nije veći od vrednosti $\sqrt{n}$ (ova dva delioca se poklapaju jedino ako je $d = \sqrt{n}$). Zato, ako broj nema delilaca iz intervala $[2, \sqrt{n}]$, ne može imati ni delilaca iz intervala $[\sqrt{n}, n-1]$. Dakle, umesto provere svih brojeva iz intervala $[2, n-1]$, dovoljno je proveriti sve kandidate iz intervala $[2, \sqrt{n}]$, što je mnogo manje brojeva i program je značajno efikasniji. O efikasnosti programa i ovim vrstama optimizacije biće mnogo više reči u drugom tomu ove knjige.

```
int n;
cin >> n;
bool prost = true;
// prosti brojevi su prirodni brojevi veci od 1: broj 1 nije
// ni prost ni slozen, a 0 i negativni brojevi nisu prosti
if (n <= 1) prost = false;
// proveravamo sve moguce delioce iz intervala [2, sqrt(n)]
for (int d = 2; d * d <= n; d++)
   if (n % d == 0)
      prost = false;
// ako nema delilaca u intervalu [2, sqrt(n)], nema ni u
 ↪  [sqrt(n), n-1]
cout << (prost ? "prost" : "nije prost") << endl;
```

Linearnu pretragu možemo upotrebiti, na primer, i da proverimo da li su dve niske jednake. Podaci tipa `string` se mogu porediti operatorom == (i to je preporučeni način njihovog poređenja), međutim, ilustracije radi i sami bismo mogli da implementiramo funkciju koja proverava da li su dve niske jednake, tako što prvo proverava da li su niske iste dužine, a zatim, ako jesu, da li postoji neka pozicija na kojoj im se nalaze različiti karakteri:

```
bool jednakeNiske(string a, string b)
{
  if (a.length() != b.length())
    return false;

  for (size_t i = 0; i < a.length(); i++) {
    if (a[i] != b[i])
      return false;
```

```
  }

  return true;
}
```

Napomenimo da ćemo u poglavlju 7 (o funkcijama) videti i efikasniji način da se dugačke niske proslede funkcijama (to nam nije u trenutnom fokusu).

6.5.4 *Sortiranost niza*

Ako je zadata relacija poretka (ili strogog poretka), onda se može proveriti da li je niz *uređen* (ili *sortiran*) u skladu sa tom relacijom. Na primer, naredna funkcija proverava da li je vektor elemenata tipa `int` uređen u skladu sa relacijom `<=`. Naravno, uslov `!(a[i] <= a[i + 1])` može da se zameni jednostavnijim `a[i] > a[i + 1]`, ali je ovde naveden jer je iskazan u terminima zadate relacije `<=`.

```
bool sortiran(const vector<int>& a)
{
  size_t i;
  if (a.size() > 0) {
    for (i = 0; i < a.size() - 1; i++)
      if (!(a[i] <= a[i + 1]))
        return false;
  }
  return true;
}
```

Navedena funkcija vraća `true` ako i samo ako je niz `a` uređen u skladu sa relacijom `<=` i tada kažemo da je on uređen ili sortiran *neopadajuće*. Slično, ako je niz uređen u skladu sa relacijom `<`, kažemo da je uređen *rastuće*, ako je niz uređen u skladu sa relacijom `>=`, kažemo da je uređen *nerastuće*, i ako je niz uređen u skladu sa relacijom `>`, kažemo da je uređen *opadajuće*.

Niz nekog brojevnog tipa, dakle, može biti uređen na različite načine. Niz elemenata tipa `int` može, na primer, biti uređen i neopadajuće po zbiru svojih cifara. Sledeća funkcija proverava da li je niz uređen na takav način (podrazumeva se da funkcija `int zbir_cifara(int)` vraća zbir cifara svog parametra):

```
bool sortiran(const vector<int>& a)
{
  size_t i;
```

```
  if (a.size() > 0) {
    for (i = 0; i < a.size() - 1; i++)
      if (!(zbir_cifara(a[i]) <= zbir_cifara(a[i + 1])))
        return false;
  }
  return true;
}
```

6.5.5 *Filtriranje, preslikavanje*

Sličan problem linearnoj pretrazi je *filtriranje serije*, tj. određivanje svih elemenata koji zadovoljavaju neki dati uslov. Takvi elementi serije čine novu seriju koja se dalje može obrađivati (na primer, možemo prebrojati takve elemente, odrediti njihov zbir, prosek, smestiti ih u neki niz ili vektor i slično). Na primer, lako možemo odrediti koliko elemenata niza je deljivo brojem 3.

```
int broj = 0;
for (int x : niz)
   if (x % 3 == 0)
      broj++;
cout << broj << endl;
```

Ako je potrebno sačuvati elemente, onda upotreba vektora može biti pogodnija nego upotreba niza, jer je vektor proširiv, tj. možemo mu efikasno dodavati jedan po jedan element.

```
vector<int> niz;
...

vector<int> deljivi_sa_3;
for (int x : niz)
   if (x % 3 == 0)
      deljivi_sa_3.push_back(x);

cout << "Brojevi deljivi sa 3 su: " << endl;
for (int x : deljivi_sa_3)
   cout << x << endl;
```

Ako bismo elemente čuvali u nizu, taj niz bi morao da bude deklarisan tako da može da sačuva potencijalno sve elemente polazne serije, tj. dužina mu mora biti jednaka

dužini polazne serije (ili veća od nje). Potrebna nam je i promenljiva koja će čuvati broj izdvojenih elemenata.

```
int niz[10];
...

int deljivi_sa_3[10];
int broj_deljivih_sa_3 = 0;
for (int i = 0; i < 10; i++)
   if (niz[i] % 3 == 0)
      deljivi_sa_3[broj_deljivih_sa_3++] = niz[i];

cout << "Brojevi deljivi sa 3 su: " << endl;
for (int i = 0; i < broj_deljivih_sa_3; i++)
   cout << deljivi_sa_3[i] << endl;
```

Primećujemo da je rešenje sa korišćenjem vektora dosta jednostavnije.

Preslikavanje serije podrazumeva primenu neke funkcije na svaki element serije. Na primer, naredni kôd ispisuje vrednost kvadratnog korena za svaki od n učitanih elemenata.

```
for (int i = 0; i < n; i++) {
   double x;
   cin >> x;
   cout << x << " " << sqrt(x) << endl;
}
```

6.5.6 *Pozicioni zapis*

Razmotrimo sada neke osnovne algoritme za rad sa pozicionim zapisom brojeva. Pretpostavićemo da se radi o dekadnim brojevima (da je osnova zapisa $b = 10$), mada se isti algoritmi mogu primeniti i na druge brojevne osnove.

Jedan od prvih zadataka je da odredimo cifre pomoću kojih je broj zapisan. Određivanje prve cifre sleva zahteva da znamo broj cifara broja, što ne znamo. Međutim, određivanje prve cifre zdesna se može jednostavno uraditi određivanjem ostatka pri deljenju sa 10. Ta se cifra može ukloniti iz broja celobrojnim deljenjem sa 10 i na taj način se problem svodi na manji, koji se dalje rešava po istom principu. Dakle, ponavljanjem ovih operacija (čitanja poslednje cifre, određivanjem ostatka pri deljenju sa 10 i uklanjanja poslednje cifre, celobrojnim deljenjem sa 10), dobijamo jednu po jednu cifru broja, zdesna nalevo.

```
do {
   int cifra = n % 10;
   n = n / 10;
   cout << cifra << endl;
} while (n > 0);
```

Razmotrimo sada obratni problem u kom želimo da na osnovu poznatih cifara odredimo vrednost broja. Pretpostavimo prvo da su cifre broja date u nizu, sleva nadesno (od cifre najveće težine ka ciframa manje težine). Potrebno je, dakle, u svakom koraku iterativnog postupka na postojeći broj zdesna dopisati cifru. To se može uraditi množenjem tekuće vrednosti broja sa 10 i sabiranjem sa vrednošću tekuće cifre. Ovaj algoritam je poznat pod nazivom *Hornerova shema*.

```
int broj = 0;
for (int cifra : cifre)
   broj = 10 * broj + cifra;
```

Zadatak se može rešiti i ako su cifre u nizu date zdesna nalevo, od cifre najmanje ka cifri najveće težine. Tada je u svakom koraku iterativnog postupka potrebno dopisati cifru na levu stranu tekućeg broja, što možemo uraditi samo ako znamo stepen broja 10, koji odgovara težini cifre koja se dodaje. Zato u algoritmu, pored vrednosti broja koji se gradi, čuvamo i težinu tekuće cifre (1 za jedinice, 10 za desetice, 100 za stotine itd.) i u svakom koraku tu težinu množimo sa 10.

```
int broj = 0;
int stepen10 = 1;
for (int cifra : cifre) {
   broj = cifra * stepen10 + broj;
   stepen10 *= 10;
}
```

6.5.7 *Leksikografsko poređenje*

Za niske (potencijalno različitih dužina) definisan je leksikografski poredak koji je zasnovan na poređenju karaktera. Pored operatora <, >, <= i >=, za leksikografsko poređenje je definisana i metoda `compare` tipa `string`. Leksikografsko poređenje se, ilustracije radi, može implementirati i samostalno.

```
int porediNiske(const string& a, const string& b)
{
  size_t minDuzina = min(a.length(), b.length());
  for (size_t i = 0; i < minDuzina; ++i) {
    if (a[i] < b[i])
      return -1;
    else if (a[i] > b[i])
      return 1;
  }
  if (a.length() < b.length())
    return -1;
  else if (a.length() > b.length())
    return 1;

  return 0;
}
```

U navedenoj funkciji, niske se porede karakter po karakter do kraja kraće niske. Ako na jednoj poziciji postoji razlika, konstatuje se da li je manja prva ili druga niska. Ako se došlo do kraja kraće niske i nije pronađena razlika ni na jednoj poziciji, onda se proverava koja je niska od dve kraća i ta se smatra manjom.

6.6 Ugnežđene petlje

Telo petlje može biti bilo koja naredba, pa i druga petlja. Takve petlje nazivamo *višestruke petlje* ili *ugnežđene* (jer se jedna petlja „ugnezdila“ u drugu). Dubina ugnežđavanja može biti i veća od 2. Na primer, naredni program ispisuje sve vremenske trenutke u jednom danu.

```
for (int sat = 0; sat < 24; sat++)
    for (int minut = 0; minut < 60; minut++)
        for (int sekund = 0; sekund < 60; sekund++)
           cout << sat << ":" << minut << ":" << sekund <<
 ↪  endl;
```

Primetimo da se unutrašnja petlja (sekunde) menja najbrže, zatim srednja (minute), dok se spoljašnja petlja (sati) menja najsporije (što je upravo željeno ponašanje).
U svakom novom koraku spoljašnje petlje, unutrašnja petlja se izvršava iznova, što može dovesti do zaista velikog broja koraka izvršavanja tela unutrašnje petlje (pogotovo kada je dubina ugnežđavanja velika).

Ugnežđene petlje se često koriste kod algoritama grube sile, gde je potrebno nabrojati sve moguće parove ili trojke elemenata. Na primer, naredna petlja nabraja sve parove elemenata sa različitih pozicija datog vektora.

```
vector<int> a;
...
for (int i = 0; i < a.size(); i++)
   for (int j = i+1; j < a.size(); j++)
       cout << a[i] << " " << a[j] << endl;
```

Redom se nabrajaju `a[0] a[1]`, `a[0] a[2]` itd. sve do `a[n-2] a[n-1]`.
Sličan efekat se može postići i na sledeći način

```
vector<int> a;
...
for (int i = 1; i < a.size(); i++)
   for (int j = 0; j < i; j++)
       cout << a[i] << " " << a[j] << endl;
```

Redom se nabrajaju `a[1] a[0]`, `a[2] a[0]`, `a[2] a[1]`, itd. sve do `a[n-1] a[n-2]`.

6.6.1 *Elementarni algoritmi sortiranja*

Česta ilustracija ugnežđenih petlji su algoritmi sortiranja. Sortiranje je obično najbolje vršiti primenom bibliotečke funkcije. U jeziku C++, to je funkcija `sort`. Funkcija `sort` se može primeniti da se sortira i deo niza, pa joj je na neki način potrebno proslediti koji se deo niza sortira. Ako želimo da se sortira ceo niz `a`, reći ćemo da se sortira deo od početka niza `begin(a)` do kraja niza `end(a)`. U poglavlju 8.2.3 o pokazivačima i iteratorima opisaćemo kog tipa su `begin(a)` i `end(a)` i proučićemo načine da se ograniči samo deo niza ili vektora koji se obrađuje.

```
vector<int> a{5, 3, 4, 2, 1};
// sortiramo vektor od pocetka do kraja
sort(begin(a), end(a));
```

Slično se može uraditi i ako se sortira niz:

```
int a[] = {5, 3, 4, 2, 1};
sort(begin(a), end(a));
```

Sortiranje celog niza je moguće uraditi i na sledeći način:

```
int a[] = {5, 3, 4, 2, 1};
sort(a, a+5); // od pocetka niza a, pa narednih 5 elemenata
```

Zbog raspoloživosti bibliotečkih funkcija, algoritmi prikazani u nastavku nemaju značajnu praktičnu primenu, pogotovo zato što su veoma neefikasni i što postoje mnogo bolji algoritmi od njih. Vremenska složenost svih ovih algoritama u najgorem slučaju je kvadratna, što znači da broj koraka koje algoritam izvršava kvadratno zavisi od broja elemenata niza koji se sortira (za sortiranje duplo dužeg niza potrebno je četiri puta više vremena). Zbog toga oni ne mogu da efikasno sortiraju nizove duže od nekoliko desetina hiljada elemenata. Ipak, oni se smatraju delom opšte programerske kulture i dobra programerska vežba je da se na osnovu njihovog opisa samostalno napravi implementacija.

6.6.1.1 *Algoritam* selection sort

Algoritam *selection sort* se ukratko može opisati na sledeći način: ako niz ima više od jednog elementa, zameni početni element sa najmanjim elementom niza i zatim analogno sortiraj ostatak niza (elemente iza početnog). U svakoj iteraciji se na svoju poziciju dovodi sledeći po veličini element niza, tj. u i-toj iteraciji se i-ti po veličini element dovodi na poziciju i. Ovo se može realizovati tako što se pronađe pozicija m najmanjeg elementa od pozicije i do kraja niza i zatim se razmene element na poziciji i i element na poziciji m. Algoritam se zaustavlja kada se pretposlednji po veličini element dovede na pretposlednju poziciju u nizu.

Primer 6.6.1. *Prikažimo rad algoritma na primeru sortiranja niza* 5 3 4 2 1.

- ***5*** 3 4 2 *1*, $i = 0$, $m = 4$, *razmena elemenata 5 i 1.*
- 1 ***3*** 4 *2* 5, $i = 1$, $m = 3$, *razmena elemenata 3 i 2.*
- 1 2 ***4*** *3* 5, $i = 2$, $m = 3$, *razmena elemenata 4 i 3.*
- 1 2 3 ***4*** 5, $i = 3$, $m = 3$, *razmena elemenata 4 i 4.*
- 1 2 3 4 5, $i = 4$

Prikažimo jednu moguću implementaciju ovog algoritma.

```
vector<int> a{5, 3, 4, 2, 1};
int n = a.size();
for (int i = 0; i < n-1; i++) {
    // pozicija minimuma u segmentu [i, n)
    int m = i;
    for (int j = i+1; j < n; j++)
```

```
        if (a[j] < a[m])
            m = j;
    // razmena elementa na poziciji i i poziciji m
    swap(a[i], a[m]);
}
```

Moguća je i naredna implementacija.

```
vector<int> a{5, 3, 4, 2, 1};
int n = a.size();
for (int i = 0; i < n-1; i++)
   for (int j = i+1; j < n; j++)
       if (a[i] > a[j])
           swap(a[i], a[j]);
```

Ova implementacija je malo jednostavnija, jer se ne traži pozicija minimuma, već se minimum dovodi na mesto `i` tako što se tekući element menja sa onim na poziciji `i` kada god je manji od njega. Ovim se može dobiti mnogo veći broj razmena nego kada se koristi prva implementacija (druga implementacija može vršiti razmenu u svakom koraku unutrašnje petlje, a prva garantovano vrši samo jednu razmenu u svakom koraku spoljašnje petlje).

6.6.1.2 *Algoritam* bubble sort

Algoritam *bubble sort* u svakom prolazu kroz niz poredi uzastopne elemente i razmenjuje im mesta ukoliko su u pogrešnom poretku. Prolasci kroz niz ponavljaju se sve dok se ne napravi prolaz u kojem nije bilo razmena, što znači da je niz sortiran.

Primer 6.6.2. *Prikažimo rad algoritma na primeru sortiranja niza* (6 1 4 3 9)*:*
Prvi prolaz:

- (***6 1*** 4 3 9) → (***1 6*** 4 3 9), *razmena jer je* $6 > 1$
- (1 ***6 4*** 3 9) → (1 ***4 6*** 3 9), *razmena jer je* $6 > 4$
- (1 4 ***6 3*** 9) → (1 4 ***3 6*** 9), *razmena jer je* $6 > 3$
- (1 4 3 ***6 9***) → (1 4 3 ***6 9***)

Drugi prolaz:

- (***1 4*** 3 6 9) → (***1 4*** 3 6 9)
- (1 ***4 3*** 6 9) → (1 ***3 4*** 6 9), *razmena jer je* $4 > 3$
- (1 3 ***4 6*** 9) → (1 3 ***4 6*** 9)
- (1 3 4 ***6 9***) → (1 3 4 ***6 9***)

Treći prolaz:

- $(\ \mathbf{1}\ \mathbf{3}\ 4\ 6\ 9\ ) \rightarrow (\ \mathbf{1}\ \mathbf{3}\ 4\ 6\ 9\ )$
- $(\ 1\ \mathbf{3}\ \mathbf{4}\ 6\ 9\ ) \rightarrow (\ 1\ \mathbf{3}\ \mathbf{4}\ 6\ 9\ )$
- $(\ 1\ 3\ \mathbf{4}\ \mathbf{6}\ 9\ ) \rightarrow (\ 1\ 3\ \mathbf{4}\ \mathbf{6}\ 9\ )$
- $(\ 1\ 3\ 4\ \mathbf{6}\ \mathbf{9}\ ) \rightarrow (\ 1\ 3\ 4\ \mathbf{6}\ \mathbf{9}\ )$

Primetimo da je niz bio sortiran već nakon drugog prolaza, međutim, da bi se to utvrdilo, bilo je potrebno napraviti još jedan prolaz.

Naredni kôd primenom algoritma *bubble sort* sortira vektor `a`.

```
vector<int> a{5, 3, 4, 2, 1};
int n = a.size();
bool bilo_razmena;
do {
   bilo_razmena = false;
   for (int i = 0; i < n - 1; i++) {
      if (a[i] > a[i + 1]) {
         swap(a[i], a[i+1]);
         bilo_razmena = true;
      }
   }
} while (bilo_razmena);
```

Nakon k-te iteracije spoljašnje petlje, k-ti najveći element je na svojoj finalnoj, ispravnoj poziciji. *Bubble sort* je na osnovu ovog svojstva i dobio ime (jer veliki elementi kao mehurići „isplivavaju“ ka kraju niza). Zbog toga u unutrašnjoj petlji (`for`) nije potrebno uvek ići do pozicije $n-1$, već po jedan manje u svakoj iteraciji. Štaviše, unutrašnja petlja može se izvršavati samo do pozicije poslednje razmene u prethodnoj iteraciji. Postoje i mnoge druge varijante ovog algoritma, ali sve imaju lošu vremensku složenost. Najgori slučaj nastupa kada je niz sortiran u obratnom redosledu.
Algoritam *bubble sort* smatra se veoma lošim algoritmom i ne treba ga koristiti u praksi.

6.6.1.3 *Algoritam* insertion sort

Algoritam *insertion sort* sortira niz tako što jedan po jedan element niza umeće na odgovarajuće mesto u do tada sortirani deo niza.

Primer 6.6.3. *Prikažimo rad algoritma na primeru sortiranja niza*

- $5\ 3\ 4\ 2\ 1$
- $\mathbf{5}\ 3\ 4\ 2\ 1$

- ***3*** *5 4 2 1*
- *3* ***4*** *5 2 1*
- ***2*** *3 4 5 1*
- ***1*** *2 3 4 5*

Podebljanim slovima prikazani su elementi umetnuti na svoju poziciju.

Algoritam *insertion sort* može se opisati na sledeći način: ako niz ima više od jednog elementa, sortiraj sve elemente ispred poslednjeg, a zatim umetni poslednji u taj sortirani podniz. U nastavku je data jedna moguća implementacija (u unutrašnjoj petlji se element menja sa svojim prethodnikom sve dok je prethodnik veći od njega).

```
vector<int> a{5, 3, 4, 2, 1};
int i, n = a.size();
for (i = 1; i < n; i++) {
   for(int j = i; j > 0 && a[j] < a[j-1]; j--)
      swap(a[j], a[j-1]);
}
```

Efikasnija verzija može se dobiti ukoliko se ne koriste razmene, već se zapamti element koji treba da se umetne, zatim se pronađe pozicija na koju treba da se umetne, svi elementi od te pozicije pomere se za jedno mesto udesno i na kraju se zapamćeni element upiše na svoje mesto:

```
vector<int> a{5, 3, 4, 2, 1};
int i, n = a.size();
for (i = 1; i < n; i++) {
   int j, tmp = a[i];
   for (j = i; j > 0 && a[j-1] > tmp; j--)
      a[j] = a[j-1];
    a[j] = tmp;
}
```

I algoritam *insertion sort* ima kvadratnu vremensku složenost, a najgori slučaj nastupa kada je niz sortiran u obratnom redosledu. Iako je algoritam *insertion sort* neefikasan prilikom sortiranja dugačkih nizova, često je kod kratkih nizova (nizova sa nekoliko desetina elemenata) brži od naprednijih algoritama.

7. Funkcije

7.1 Modularnost i razlaganje problema na potprobleme

Svaki C++ program sačinjen je od funkcija. Funkcija `main` mora da postoji i, pojednostavljeno rečeno, izvršavanje programa uvek počinje izvršavanjem ove funkcije. Iz funkcije `main` (ali i drugih) pozivaju se druge funkcije, bilo bibliotečke (na primer, `sqrt`, kojom se izračunava kvadratni koren), bilo korisnički definisane.
Veliki program veoma je teško napisati ili razumeti ako nije podeljen na celine, tj. module. Podela programa na celine (na primer, datoteke i funkcije) neophodna je za razumevanje programa i nametnula se veoma rano u istoriji programiranja. Svi savremeni programski jezici su dizajnirani tako da je podela na manje celine ne samo moguća, već i tipična, a način podele često određuje i sam stil programiranja (na primer, u objektno-orijentisanim jezicima neki podaci i metode za njihovu obradu se grupišu u *klase*). Podela programa na celine utiče na više njegovih osobina:

- **Kraći kôd**: ukoliko se isti kôd ne ponavlja na više mesta u programu, već je izdvojen u zasebnu celinu, program će biti jednostavniji i koncizniji tj. kraći.
- **Lakše održavanje**:
  ukoliko je u delu koda koji je izdvojen u funkciju otkrivena greška, treba je ispraviti na samo jednom mestu (a ne na više, što bi bio slučaj da taj deo nije izdvojen u funkciju).
- **Čitljivost i razumljivost**: podela programa na celine popravlja njegovu čitljivost i razumljivost i omogućava i onome ko piše i onome ko čita program da se usredsredi na ključna pitanja jedne celine, zanemarujući u tom trenutku i iz te perspektive funkcionalnosti podržane drugim celinama. Pored dobre podele na funkcije, na čitljivost utiče i dobro imenovanje funkcija (tako da je jasno šta koja radi).
- **Šira upotrebljivost**: ukoliko je kôd kvalitetno podeljen na celine, pojedine celine tj. funkcije biće moguće upotrebiti u nekom drugom kontekstu u programu. Ako

je neko izračunavanje skriveno u jednoj celini, ona može da se koristi i ako se ne zna kako je tačno implementirana, već je dovoljno znati šta radi, tj. kakav je rezultat njenog rada za zadate argumente. Dodatno, pojedine funkcije mogu da se koriste i u drugim programima.
Na primer, proveravanje da li neki trinaestocifreni kôd predstavlja mogući JMBG (jedinstveni matični broj građana) može se izdvojiti u zasebnu funkciju koja je onda upotrebljiva u različitim programima. Srodne funkcije mogu da se grupišu u *biblioteke* (koje mogu da se koriste u drugim programima).

- **Lakša podela zadataka članovima tima**: ukoliko je program osmišljen tako da je sačinjen od logičnih celina, lakše ga je razvijati u timu – pojedinačni članovi tima rade na zasebnim funkcijama.

Za većinu jezika, celine na koje se deli kôd su obično funkcije. Program se nikada ne deli na funkcije i datoteke tek onda kada je kompletno završen. Naprotiv, podela koda u funkcije vrši se još u ranim fazama pisanja programa i predstavlja jedan od najvažnijih aspekata dizajna programa. Za pisanje funkcija postoje mnoge smernice (koje nisu stroga pravila), poput:

- Jedna funkcija u programu, u principu, treba da obavlja samo jedan zadatak.
- Tekst jedne funkcije ne treba da bude previše dug i poželjno je da staje na jedan ili dva ekrana (tj. da ima manje od pedesetak redova), radi dobre preglednosti. Duge funkcije poželjno je podeliti na manje funkcije.
- Ukoliko funkcija ima više od, na primer, 10 lokalnih promenljivih, verovatno je funkciju poželjno podeliti na nekoliko manjih. Slično važi i za broj parametara funkcije.

Ukoliko je brzina izvršavanja kritična, kompilatoru se može naložiti da *inlajnuje* kratke funkcije (da prilikom kompilacije umetne kôd kratkih funkcija na pozicije gde su pozvane)[1].

7.2 Primeri korišćenja funkcija

Ilustrujmo prednosti definisanja i korišćenja pomoćnih funkcija kroz nekoliko primera.

Primer 7.2.1. *Napišimo program koji izračunava da li dva uneta broja imaju isti zbir cifara i to prvo bez, a onda uz korišćenje funkcija.*

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

[1] Inlajnovanje u nekim situacijama kompilatori primenjuju i bez eksplicitnog zahteva programera.

```
int main() {
   int a, b;
   cin >> a >> b;

   // izracunavamo zbir cifara broja a
   int zbir_cifara_a = 0;
   do {
      int cifra = a % 10;
      zbir_cifara_a += cifra;
      a /= 10;
   } while (a > 0);

   // izracunavamo zbir cifara broja b
   int zbir_cifara_b = 0;
   do {
      int cifra = b % 10;
      zbir_cifara_b += cifra;
      b /= 10;
   } while (b > 0);

   if (zbir_cifara_a == zbir_cifara_b)
      cout << "da" << endl;
   else
      cout << "ne" << endl;
}
```

Primećujemo da se u prethodnom programu kôd koji izračunava zbir cifara broja nepotrebno ponavlja dva puta. Mnogo bolje rešenje se dobija ako se taj kôd izdvoji u zasebnu funkciju, koja se onda dva puta koristi: jednom da se izračuna zbir cifara broja a, a drugi put da se izračuna zbir cifara broja b.

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

int zbir_cifara(int n) {
   int zbir = 0;
   do {
      int cifra = n % 10;
      zbir += cifra;
      n /= 10;
```

```
    } while (n > 0);
    return zbir;
}

int main() {
    int a, b;
    cin >> a >> b;
    if (zbir_cifara(a) == zbir_cifara(b))
        cout << "da" << endl;
    else
        cout << "ne" << endl;
}
```

Primer 7.2.2. *Razmotrimo još jedan primer u kome korišćenje funkcija značajno skraćuje program. Zadatak je da se prirodan broj* `n` *(manji od 4000) zapiše pomoću rimskih cifara. Na primer, broj 1283 se rimski zapisuje kao* `MCCLXXXIII`*. Zašto je to tako? Zato što se cifra jedinica 3 zapisuje kao* `III`*, cifra desetica 8 se zapisuje kao* `LXXX`*, cifra stotina 2 se zapisuje kao* `CC`*, dok se cifra hiljada 1 zapisuje kao* `M`*. Dakle, treba odrediti dekadne cifre i svaku od njih pojedinačno prevesti u rimski zapis. Rešenje se može značajno pojednostaviti ako se primeti da su pravila za određivanje rimskog zapisa na osnovu vrednosti cifre jedinica, desetica i stotina praktično ista, jedino se razlikuju simboli (slova) pomoću kojih se zapis gradi. Na primer, ciframa jedinica 1 do 9 odgovaraju redom rimski zapisi* `I`*,* `II`*,* `III`*,* `IV`*,* `V`*,* `VI`*,* `VII`*,* `VIII` *i* `IX`*, dok ciframa desetica od 1 do 9 odgovaraju redom rimski zapisi* `X`*,* `XX`*,* `XXX`*,* `XL`*,* `L`*,* `LX`*,* `LXX`*,* `LXXX` *i* `XC`*. Dakle, u oba slučaja se koriste 3 simbola (simbol čija je vrednost 1, simbol čija je vrednost 5 i simbol čija je vrednost 10 i to su u prvom slučaju* `I`*,* `V` *i* `X`*, a u drugom slučaju* `X`*,* `L` *i* `C`*) i pravila povezivanja ovih simbola potpuno su ista (za cifru od 1 do 3 simbol vrednosti 1 se ponovi onoliko puta kolika je cifra, za cifru 4 se nadovežu simbol vrednosti 1 i simbol vrednosti 5 itd.). Ista pravila važe i za cifre stotina (tada se koriste simboli* `C`*,* `D` *i* `M`*). Zato je poželjno definisati funkciju koja dobija vrednost cifre (od 1 do 9) i tri simbola i na osnovu toga gradi odgovarajuću nisku.*
Pošto se niska gradi tako što se neki simboli ponavljaju, potrebno je pronaći način da se izgradi niska dobijena ponavljanjem datog karaktera `c` *dati broj puta* `k`*. U jeziku C++ za to se može upotrebiti poziv funkcije (konstruktora)* `string(k, c)` *(ova funkcija i druge slične mogu se pronaći pregledom dokumentacije jezika).*

```
// zapis jedne rimske cifre - simbol1, simbol5 i simbol10
 ↪  odredjuju
// da li se radi o cifri jedinica, desetica ili stotina
string rimska_cifra(int c, char simbol1, char simbol5, char
 ↪  simbol10) {
```

```
  if (c < 4) // npr. "", "I", "II", "III"
    return string(c, simbol1);
  else if (c == 4) // npr. "IV"
    return string(1, simbol1) + string(1, simbol5);
  else if (c < 9) // npr. "V", "VI", "VII", "VIII"
    return string(1, simbol5) + string(c-5, simbol1);
  else // npr. "IX"
    return string(1, simbol1) + string(1, simbol10);
}

// prevodi dati broj u rimski zapis
string arapski_u_rimski(int n) {
  string rezultat = "";
  // cifra jedinica
  rezultat = rimska_cifra(n % 10, 'I', 'V', 'X');
  n /= 10; // uklanjamo cifru jedinica
  // cifra desetica
  rezultat = rimska_cifra(n % 10, 'X', 'L', 'C') + rezultat;
  n /= 10; // uklanjamo cifru desetica
  // cifra stotina
  rezultat = rimska_cifra(n % 10, 'C', 'D', 'M') + rezultat;
  n /= 10; // uklanjamo cifru stotina
  // dodajemo hiljade na pocetak rezultata i vracamo ga
  return string(n, 'M') + rezultat;
}
```

7.3 Parametri funkcije

Funkcija može imati parametre koje obrađuje i oni se navode u okviru definicije funkcije, iza imena funkcije i između zagrada. Termini *parametar funkcije* i *argument funkcije* se ponekad koriste kao sinonimi. Ipak, pravilno je termin *parametar funkcije* koristiti za promenljivu koja je deo deklaracije funkcije, a termin *argument funkcije* za izraz naveden u pozivu funkcije na mestu parametra funkcije. Ponekad se argumenti funkcija nazivaju i *stvarni argumenti*, a parametri funkcija *formalni argumenti*. U primeru 7.2.1, `n` je parametar funkcije `int zbir_cifara(int n)`, a `a` i `b` su njeni argumenti u pozivima `zbir_cifara(a)` i `zbir_cifara(b)`.

Pre imena svakog parametra funkcije (u početnom delu koji deklariše funkciju) neophodno je navesti njegov tip. Kao i imena promenljivih, imena parametara treba da oslikavaju njihovo značenje i ulogu u programu. Ukoliko funkcija nema parametara, onda se između zagrada ne navodi ništa (na primer, `int main()`). Moguće je između zagrada

navesti i ključnu reč `void`, ali je to stil karakterističan za programski jezik C i u jeziku C++ se obično ne koristi.
Parametri funkcije mogu se u telu funkcije koristiti kao lokalne promenljive te funkcije koje imaju početnu vrednost određenu vrednostima argumenata u pozivu funkcije.
Promenljive koje su deklarisane kao parametri funkcije lokalne su za tu funkciju i njih ne mogu da koriste druge funkcije. Štaviše, bilo koja druga funkcija može da koristi isto ime za neki svoj parametar ili za neku svoju lokalnu promenljivu.
Kao što je već ranije rečeno, kvalifikatorom `const` mogu, kao i sve promenljive, biti označeni parametri funkcije, čime se obezbeđuje da neki parametar ili sadržaj na koji ukazuje neki parametar neće biti menjan u funkciji.
Funkcija `main` može biti bez parametara ili može imati dva parametra unapred određenog tipa (videti poglavlje 9.5.6).
Prilikom poziva funkcije, vrši se *prenos argumenata*, što će biti opisano u poglavlju 7.6.

7.4 Povratna vrednost funkcije

Funkcija može da vraća rezultat i tip rezultata se zapisuje na samom početku definicije funkcije (pre njenog imena). Na primer, tip povratne vrednosti funkcije `zbir_cifara` je `int`. Funkcija rezultat vraća naredbom `return r;` gde je `r` izraz zadatog tipa ili tipa koji se može konvertovati u taj tip. Naredba `return r;` ne samo da vraća vrednost `r` kao rezultat rada funkcije, nego i prekida njeno izvršavanje. Na primer, algoritam linearne pretrage se često može implementirati u posebnoj funkciji. Naredna funkcija proverava da li je dati broj prost, prekidajući pretragu i vraćajući rezultat `false` čim naiđe na neki delilac broja.

```
bool prost(unsigned n) {
   if (n <= 1) return false;
   if (n == 2) return true;
   if (n % 2 == 0) return false;
   for (int d = 3; d*d <= n; d += 2)
      if (n % d == 0)
         return false;
   return true;
}
```

Ako funkcija ne treba da vraća rezultat, onda se kao tip povratne vrednosti navodi specijalan tip `void` i tada naredba `return` nema argumenata (tj. navodi se `return;`). Štaviše, u tom slučaju nije neophodno navoditi naredbu `return` iza poslednje naredbe u funkciji (`return` se koristi jedino kada želimo da ranije prekinemo tok izvršavanja funkcije). S druge strane, ako funkcija koja treba da vrati vrednost dođe do kraja svog tela bez izvršavanja naredbe `return`, kompilator može da prijavi upozorenje, a u fazi izvršavanja

dolazi do nedefinisanog ponašanja programa. Jedini izuzetak je funkcija `main`: ako se njeno izvršavanje završi bez naredbe `return`, podrazumeva se da je vraćena vrednost 0 (zato u nekim primerima u ovoj knjizi funkcija `main` nema naredbu `return 0;`).
Funkcija koja je pozvala neku drugu funkciju može da ignoriše, tj. da ne koristi vrednost koju je ova vratila. Naime, svaki poziv funkcije je izraz, a vrednost bilo kog izračunatog izraza se može ignorisati nakon njegovog izračunavanja (što ima smisla kada nam je jedini cilj ostvarivanje propratnog efekta izračunavanja tog izraza).
Kvalifikator `const` može se primeniti i na tip povratne vrednosti funkcije. To nema mnogo smisla (osim u kombinaciji sa pokazivačima, čime se ova knjiga ne bavi) i retko se koristi.
Funkcija `main` mora da ima `int` kao tip povratne vrednosti (jer okruženje iz kojeg je program pozvan uvek kao povratnu vrednost očekuje tip `int`). Definicija oblika `void main()`, koju neki prevodioci za jezik C dopuštaju, u jeziku C++ dovodi do greške prilikom prevođenja.

7.5 Funkcije i metode

Pored funkcija koje smo do sada definisali, često koristimo i *metode*, tj. funkcije koje su definisane u okviru neke klase ili strukture (na primer, `size`, `push_back` ili `substr`). Metoda se uvek poziva za neki konkretan objekat, navođenjem imena objekta, tačke i imena metode (na primer, `s.size()` ili `v.push_back(x)`), i ima direktan pristup podacima tog objekta. Obična funkcija ne pripada nijednom objektu i sve podatke dobija preko argumenata (na primer, `sqrt(x)` ili `swap(a, b)`). Neke operacije su u biblioteci dostupne na oba načina: dužina vektora `v` može se dobiti i metodom, `v.size()`, i funkcijom, `size(v)`, a početak vektora i sa `v.begin()` i sa `begin(v)`. Sve što je u ovom poglavlju rečeno o parametrima, povratnoj vrednosti i prenosu argumenata važi i za metode. Kako se metode definišu, biće prikazano u poglavljima 8.1.2 (metode struktura) i 8.1.3 (metode klasa).

7.6 Prenos argumenata

Telo svake funkcije sadrži neke naredbe kojima se obrađuju neki podaci i dobijaju neki rezultati. Obično je funkciji potrebno na neki način preneti podatke koje treba da obradi. Takođe, nakon što završi obradu dobijenih podataka, funkcija obično treba da dobijene rezultate nekako vrati pozivaocu. U slučaju jednostavnih funkcija, kakve smo do sada sretali, za prenos podataka u funkciju koriste se argumenti funkcije, a rezultat rada funkcije vraća se u vidu povratne vrednosti. Međutim, postoje scenariji u kojima su potrebna komplikovanija rešenja. Na primer, u nekim situacijama funkcija kao rezultat treba da vrati više podataka (a povratna vrednost funkcije je uvek jedinstvena vrednost). Jedan način da se to uradi je da se više podataka upakuje u neku celinu (na primer, da se napravi torka elemenata, ili da se definiše poseban strukturni tip). Drugi način je da se

parametri funkcije upotrebe i za vraćanje vrednosti pozivaocu (a ne samo za primanje vrednosti). U programiranju su uobičajene tri vrste parametara funkcija:

- *ulazni parametri* (služe samo da se funkciji predaju podaci koje treba da obradi);
- *izlazni parametri* (služe da funkcija vrati vrednost);
- *ulazno-izlazni* parametri (služe da funkcija primi podatke koje treba da obradi, zatim da modifikuje i tako modifikovane vrati pozivaocu).

Dok neki programski jezici imaju mehanizme kojima se razlikuju sve tri vrste parametara, u jeziku C++ ne postoje čisti izlazni parametri, već se za vraćanje vrednosti preko parametara koriste ulazno-izlazni parametri.

- Ulazni parametri (tj. parametri koji služe da se funkciji predaju podaci koje treba da obradi) zadaju se tako što se prilikom poziva funkcije napravi kopija argumenata (u memorijskom prostoru namenjenom izvršavanju te funkcije) i funkcija nakon toga pristupa kopijama podataka. Funkcija može i da menja te kopije, ali će originalni podaci ostati nepromenjeni. U jeziku C++, ulazni parametri realizuju se kroz prenos argumenta u funkciju koji zovemo:
  - prenos po vrednosti (engl. *pass by value*)
- Ulazno-izlazni parametri (tj. parametri koji služe da funkcija primi podatke koje treba da obradi, zatim da modifikuje i tako modifikovane vrati pozivaocu) zadaju se tako što se ne pravi kopija, već funkcija dobija mogućnost da pristupa originalnim podacima (tj. argumentima koje je dobila) i da ih menja. Ovo se ostvaruje tako što se funkciji omogući pristup memorijskoj lokaciji na kojoj se nalaze originalni podaci. Ako je to adresa neke promenljive iz funkcije pozivaoca, kada pozvana funkcija upiše neke podatke na tu adresu, biće izmenjena odgovarajuća promenljiva unutar funkcije pozivaoca (čime je pozvana funkcija vratila nekakav rezultat svog rada). U jeziku C++, ulazno-izlazni parametri realizuju se kroz dve vrste prenosa argumenta u funkciju koje zovemo:
  - prenos po referenci (engl. *pass by reference*)
  - prenos po adresi, tj. prenos po pokazivaču (engl. *pass by address*, *pass by pointer*)

Vrste prenosa argumenta biće detaljnije objašnjene u nastavku.

7.6.1 Prenos argumenata po vrednosti

Za ulazne parametre najčešće se koristi *prenos po vrednosti*. Prilikom prenosa po vrednosti, vrednost koja se koristi kao argument funkcije *kopira se* kada počne izvršavanje funkcije (u memorijski prostor namenjen izvršavanju te funkcije) i onda funkcija radi samo sa tom kopijom, ne menjajući original.

Razmotrimo, kao primer, funkciju `zbir_cifara` koja je u primeru 7.2.1 deklarisana sa `int zbir_cifara(int n)` i pozvana sa `zbir_cifara(a)`, gde je `a` promenljiva učitana u funkciji `main`. Ta promenljiva će nakon izvršenja funkcije `zbir_cifara` ostati nepromenjena, ma kako da je funkcija `zbir_cifara` definisana, tj. i ako menja vrednost svog parametra `n`. Naime, kada počne izvršavanje funkcije `zbir_cifara`, vrednost promenljive `a` biće iskopirana u lokalnu promenljivu `n` koja je navedena kao parametar funkcije i funkcija će koristiti samo tu kopiju u svom radu. U ovom konkretnom primeru, funkcija `zbir_cifara` zaista menja vrednost promenljive `n` (deli je sa 10, sve dok joj vrednost ne postane 0), ali promenljiva `a` ostaje nepromenjena (pošto je `n` samo kopija promenljive `a`).

Naglasimo da imena promenljivih u ovom slučaju nisu relevantna: čak i da se promenljiva u funkciji `zbir_cifara` takođe zvala `a` i da je ona menjana, u programu bi zapravo postojale dve različite promenljive `a` (jedna u funkciji `main` i njena kopija u funkciji `zbir_cifara`). Dakle, moguće je i da se ime parametra funkcije poklapa sa imenom promenljive koja je prosleđena kao stvarni argument, na primer:

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

void f(int a) {
  a = 3;
  cout << "f: a = " << a << endl;
}

int main() {
  int a = 5;
  f(a);
  cout << "main: a = " << a << endl;
}
```

I u ovom slučaju radi se o dve različite promenljive (promenljiva u pozvanoj funkciji je kopija promenljive iz funkcije u kojoj se poziv nalazi).

```
f: a = 3
main: a = 5
```

U pozivu funkcije, argument koji se prenosi po vrednosti može biti promenljiva, ali i bilo koji izraz istog tipa (ili izraz čija vrednost može da se konvertuje u taj tip). Na primer, funkcija `zbir_cifara` iz pomenutog primera može biti pozvana sa `zbir_cifara(12345)`, ali i sa `zbir_cifara(12345+67890)`.

Prenos argumenata po vrednosti ilustruje i funkcija `swap` kojoj je zadatak da razmeni vrednosti dve promenljive. Njena naredna definicija je, usled prenosa po vrednosti, pogrešna, tj. ne razmenjuje vrednosti svojih argumenata.

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

void swap(int a, int b) {
   int temp = a;
   a = b;
   b = temp;
}

int main() {
   int x = 3, y = 5;
   swap(x, y);
   cout << "x = " << x << ", y = " << y << endl;
}
```

U funkciji `swap` promenljive `a` i `b` razmenjuju vrednosti, no ako je funkcija pozvana iz neke druge funkcije sa `swap(x, y)`, onda će vrednosti promenljivih `x` i `y` ostati nepromenjene nakon ovog poziva, te navedeni program daje naredni ispis.

```
x = 3, y = 5
```

Na isti način kao i osnovni tipovi (npr. `int`, `double`, `bool`) po vrednosti se prenose i objekti (npr. `string`, `vector`) i strukture. Naglasimo da podaci ovih tipova mogu biti veliki i njihovo kopiranje zahteva dodatno vreme i zahteva dodatne memorijske resurse. Stoga se, u cilju optimizacije, čak i kada se ovi tipovi podataka koriste kao ulazni parametri funkcije, ne savetuje njihov prenos po vrednosti (već najčešće prenos po konstantnoj referenci, o čemu će biti više reči u nastavku).

7.6.2 *Prenos argumenata po referenci*

U prethodnom poglavlju ilustrovano je da, ako je neka promenljiva kao argument preneta po vrednosti u neku funkciju, onda će njena vrednost biti prekopirana, ta kopija će biti korišćena u funkciji, možda i promenjena, ali originalna promenljiva ostaće neizmenjena. Ukoliko je unutar funkcije potrebno promeniti neku promenljivu koja joj je poslata kao argument, onda se ona prenosi po *referenci*, a taj se parametar označava simbolom &. U ovoj vrsti prenosa, funkciji se prosleđuje *referenca na originalnu promenljivu*. Argument koji se prenosi po referenci mora da bude promenljiva. Izuzetak je slučaj kada odgovarajući parametar ima kvalifikator `const` – tada argument može

biti i konstanta ili proizvoljan izraz (prevodilac tada pravi privremenu vrednost na koju referenca ukazuje). Referenca se može smatrati drugim imenom za originalni argument i sve promene nad referencom odražavaju se na originalnu promenljivu. Štaviše, promenljiva koja kao parametar figuriše u funkciji nema ni svoj memorijski prostor (u fazi izvršavanja), već koristi prostor promenljive koja je argument. Dakle, u pozadini ovog prenosa se, zapravo, funkciji ne prenosi vrednost originalne promenljive, već samo njena adresa. O detaljima realizacije tog prenosa stara se kompilator i programer ne mora da razmišlja o njima.
Razmotrimo ponovo funkciju `swap` i implementirajmo je tako da ona zaista razmenjuje vrednosti promenljivih za koje je pozvana. Poziv funkcije ne mora da se promeni, ali njen prototip mora – njeni parametri će sada biti reference:

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

void swap(int &a, int &b) {
   int temp = a;
   a = b;
   b = temp;
}

int main() {
   int x = 3, y = 5;
   swap(x, y);
   cout << "x = " << x << ", y = " << y << endl;
}
```

Funkcija `swap` preko svojih parametara (koji su reference) ima pristup promenljivim `x` i `y`, te navedeni program daje naredni ispis.

```
x = 5, y = 3
```

Naglasimo da standardna biblioteka jezika C++ već sadrži funkciju `swap` koja razmenjuje vrednosti promenljivih (da bi se koristila, potrebno je uključiti zaglavlje `<utility>`), pa je nema potrebe definisati u programu.
Prenos po referenci može se koristiti i da bi se definisale funkcije koje mogu da vrate više vrednosti (što je situacija koja je objašnjena ranije). Na primer, naredna funkcija ugao zadat u sekundama prevodi u stepene, minute i sekunde.

```
// ugao od S sekundi prevodi u stepene, minute i sekunde
void od_sekundi(int S, int& stepeni, int& minuti, int&
 ↪  sekundi) {
   sekundi = S % 60;
   minuti  = (S / 60) % 60;
   stepeni = S / 3600;
}
```

Naravno, više vrednosti se može vratiti i definisanjem zasebne strukture.

```
struct Ugao {
   int stepeni, minuti, sekundi;
};

Ugao od_sekundi(int S) {
   Ugao rezultat;
   rezultat.sekundi = S % 60;
   rezultat.minuti  = (S / 60) % 60;
   rezultat.stepeni = S / 3600;
   return rezultat;
}
```

Alternativa je i da se vrati torka.

```
tuple<int, int, int> od_sekundi(int S) {
   int sekundi = S % 60;
   int minuti = (S / 60) % 60;
   int stepeni = S / 3600;
   return make_tuple(stepeni, minuti, sekundi);
}
```

Kao što je rečeno, kada se vrši prenos argumenta po referenci, ne vrši se njegovo kopiranje, već se pristupa direktno originalnom argumentu. Ukoliko je argument objekat ili struktura koji zauzima veliki broj bajtova (na primer, `string` ili `vector`), to donosi značajnu prostornu i vremensku efikasnost jer nema kopiranja (koje postoji u prenosu po vrednosti). Zato prenos po referenci može biti koristan i kada nije potrebno izmeniti neki argument funkcije. Ali, ukoliko se neki parametar funkcije ne menja u njoj, tada je poželjno to obezbediti i naglasiti i samom deklaracijom, navođenjem reči `const` ispred deklaracije tog parametra. Na primer, funkcija koja izračunava broj razmaka u tekstu bi trebalo da bude definisana na sledeći način.

```
int broj_razmaka(const string& tekst) {
   int broj = 0;
   for (char c : tekst)
       if (c == ' ')
          broj++;
   return broj;
}
```

U slučaju da je tekst dugačak, ovim se dobija značajna ušteda u odnosu na prenos po vrednosti, naročito ako se ova funkcija često poziva u nekom programu.

7.6.3 *Prenos argumenata po adresi*

Prenos argumenta po adresi (tj. prenos po pokazivaču) sličan je prenosu po referenci, s tim što se u ovom slučaju funkciji dostavlja adresa neke promenljive, tj. *pokazivač* na nju, pa će odgovarajući parametar da bude pokazivačkog tipa. Pokazivač je promenljiva koja (u fazi izvršavanja) ima svoj memorijski prostor i može da u njemu čuva adresu neke druge promenljive. Raspolaganje pokazivačem na neku promenljivu omogućava funkciji da pristupa toj promenljivoj, pa i da je menja ukoliko je to potrebno. Efekat će biti praktično isti kao u slučaju korišćenja prenosa po referenci, ali će postupak prenosa adrese biti eksplicitan: u pozivu funkcije, programer mora da navede da u funkciju kao argument šalje adresu neke promenljive (navođenjem simbola &), a ne njenu vrednost. Funkcija `swap` može biti implementirana i korišćenjem prenosa po adresi, ali će to zahtevati ne samo izmenu početne definicije funkcije, nego i načina na koji se ona poziva. Parametri funkcije biće *pokazivačkog tipa* (na primer, `int*`), argumenti će biti adrese promenljivih (koje se dobijaju primenom *operatora referenciranja*, &), a za pristup promenljivim na koje ukazuju pokazivači primenjuje se *operator dereferenciranja* `*`.

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

void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

int main() {
   int x = 3, y = 5;
   swap(&x, &y);
```

```
    cout << "x = " << x << ", y = " << y << endl;
}
```

Funkcija `swap` preko svojih parametara (koji su pokazivači) ima pristup promenljivim `x` i `y`, te navedeni program daje naredni ispis.

```
x = 5, y = 3
```

Zvezdice u deklaraciji označavaju pokazivački tip `int *` (pokazivač na promenljivu koja je tipa `int`), dok zvezdice u kodu označavaju dereferenciranje, tj. pristup promenljivoj na koju ukazuje pokazivač (na primer, `int temp = *a;` uzrokuje da celobrojna promenljiva `temp` dobije vrednost promenljive na koju ukazuje pokazivač `a`, dok `*a = *b` uzrokuje da promenljiva na koju ukazuje pokazivač `a` dobije vrednost one promenljive na koju ukazuje pokazivač `b`).

Slično kao kod prenosa po referenci, prenos po adresi zahteva kopiranje samo pokazivača, što je obično daleko manje nego kopiranje promenljivih koje su argumenti. Ponovo kao kod prenosa po referenci, ukoliko se u funkciji ne menja vrednost na koju ukazuje parametar koji je pokazivač, poželjno je to naglasiti i samom deklaracijom, navođenjem reči `const` ispred deklaracije tog parametra. U pozivu funkcije, argument koji se prenosi po adresi mora da bude samo adresa promenljive (pa, dakle, kao argument po adresi ne može biti preneta konstanta ili neki izraz).

Efekat prenosa po adresi je, dakle, isti kao u slučaju prenosa po referenci, ali je sintaksa napisanog programa znatno komplikovanija (zbog eksplicitnog referenciranja i dereferenciranja). Prenos po adresi (tj. preko pokazivača) tipičan je za programski jezik C, koji ne poseduje mehanizam prenosa po referenci.[2] U jeziku C++ se pokazivači koriste u tehnikama programiranja koje prevazilaze domet ove knjige (na primer, u radu sa dinamički alociranom memorijom i implementaciji dinamičkih struktura podataka), tako da ćemo se njima baviti tek u njenim narednim tomovima.

7.7 Konverzije tipova argumenata funkcije

Prilikom poziva funkcije, vrši se implicitna konverzija tipova argumenata u tipove parametara (ako se oni razlikuju). Slično, prilikom vraćanja vrednosti funkcije (putem `return` naredbe) vrši se konverzija vrednosti koja se vraća u tip povratne vrednosti funkcije.

[2] Pošto se prenos argumenta po adresi svodi na prenos adrese promenljive po vrednosti, može se smatrati da je prenos po adresi vrsta prenosa po vrednosti. Zbog toga se često insistira da u jeziku C postoji samo prenos argumenata po vrednosti.

7.8 Anonimne funkcije

U jeziku C++ moguće je definisanje tzv. anonimnih, tj. *lambda-funkcija*[3]. Te funkcije nemaju svoje ime (mada se mogu dodeliti promenljivoj i na taj način imenovati). Te funkcije obično se koriste za samo jedan kontekst, tj. za izračunavanja potrebna samo u okviru nekog šireg izračunavanja (često se kao parametri prosleđuju funkcijama standardne biblioteke, kao što je prikazano u poglavlju 9). Korišćenje anonimnih funkcija često omogućava pojednostavljivanje i skraćivanje koda, kao i smanjivanje broja identifikatora (što dalje vodi ka čistijem i bolje organizovanom kodu). Na primer, promenljiva `kvadrat` čuva lambda-funkciju (objekat koji se može pozivati kao funkcija) koja izračunava kvadrat datog celog broja.

```
int main() {
   auto kvadrat = [] (int x) {
      return x * x;
   };
   cout << kvadrat(3) << endl;
}
```

Tip povratne vrednosti lambda-funkcije `kvadrat` nije eksplicitno naveden (što je često slučaj prilikom definisanja lambdi), već ga kompilator sam zaključuje na osnovu naredbe `return`. Ključna reč `auto` u deklaraciji označava da kompilator sam određuje tip promenljive `kvadrat` (svaki lambda-izraz ima svoj poseban tip, čije ime se ne može zapisati, pa je korišćenje `auto` ovde neophodno). Umesto tipa povratne vrednosti i imena funkcije, definicija lambda-funkcije počinje spiskom *uhvaćenih promenljivih* (engl. captured variables) navedenim u sklopu uglastih zagrada, koji je u ovom primeru prazan (`[]`). Nakon toga sledi spisak parametara funkcije i telo funkcije, koji se ni po čemu ne razlikuju od običnih funkcija. Tačka-zapeta na kraju dolazi od deklaracije sa inicijalizacijom (završetak deklaracije promenljive je uvek tačka-zapeta). Moguće je i eksplicitno navesti tip povratne vrednosti lambda-funkcije.

```
int main() {
   ...
   auto kvadrat = [] (int x) -> int {
      return x * x;
   };
   ...
```

[3]Naziv lambda dolazi od grčkog slova λ koje je Alonzo Čerč koristio u svojoj definiciji pojma funkcije i izračunavanja – tzv. λ-račun. Ovaj formalizam predstavlja teorijsku osnovu savremenog funkcionalnog programiranja.

```
}
```

Promenljiva `kvadrat` (koja se koristi kao funkcija) je lokalna u funkciji `main` i ne može se koristiti van nje.
Lambda-funkcija može pristupati samo onim lokalnim promenljivim okolne funkcije koje su eksplicitno navedene u spisku uhvaćenih promenljivih. U narednom primeru, lambda-funkcija `veci_od_praga` proverava da li je data vrednost `x` veća od datog praga i mora da ima pristup promenljivoj `prag`. Lambda-funkciju `veci_od_praga` prosleđujemo bibliotečkoj funkciji `count_if` (ova i njoj slične funkcije su opisane u poglavlju 9), koja broji elemente date kolekcije (u ovom primeru vektora) za koje data funkcija vraća vrednost `true`.

```
int main() {
    vector<int> a = {3, 7, 1, 4, 2, 5, 9};
    int prag;
    cin >> prag;
    auto veci_od_praga = [prag] (int x) { return x > prag; };
    cout << count_if(begin(a), end(a), veci_od_praga) <<
 ↪  endl;
}
```

Lambda je u prethodnom primeru upotrebljena samo da bi se prosledila kao parametar funkcije `count_if`. Da bi se to uradilo, nije neophodno uvoditi promenljivu `veci_od_praga`.

```
int main() {
    vector<int> a = {3, 7, 1, 4, 2, 5, 9};
    int prag;
    cin >> prag;
    cout << count_if(begin(a), end(a),
                     [prag] (int x) { return x > prag; })
         << endl;
}
```

Prethodni stil programiranja koji se zasniva na korišćenju bibliotečkih funkcija koje kao argumente primaju funkcije inspirisan je funkcionalnom paradigmom i tipičan je za savremeni C++ (o čemu govori i poglavlje 9). Naravno, ista funkcionalnost može se postići i bez korišćenja lambda-funkcija i biblioteke, i u ovom konkretnom primeru dobija se rešenje koje nije lošije od prethodnog.

```
int main() {
    vector<int> a = {3, 7, 1, 4, 2, 5, 9};
    int prag;
    cin >> prag;
    int broj_vecih_od_praga = 0;
    for (int x : a)
       if (x > prag)
          broj_vecih_od_praga++;
    cout << broj_vecih_od_praga << endl;
}
```

Za razliku od navedenog primera, naprednije funkcije (poput funkcija sortiranja ili efikasne pretrage sortiranog niza) je teže samostalno implementirati i u tim situacijama je poželjno poznavati i koristiti lambda-funkcije.
Ako je potrebno da se iz anonimne funkcije promeni vrednost neke okolne lokalne promenljive, ona se u grupi uhvaćenih promenljivih navodi uz simbol &, koji označava prenos po referenci. Na primer, grupa `[x, &y]` označava da lambda ima pristup promenljivoj `x` po vrednosti (može da je čita, ali ne i da je menja), a promenljivoj `y` po referenci (može da je i čita i menja).
Grupa `[=]` označava da lambda ima pristup po vrednosti svim okolnim promenljivim, a `[&]` da lambda ima pristup po referenci svim okolnim promenljivim.

7.9 Složeni tipovi i funkcije

Parametri funkcija mogu biti i strukture, drugi korisnički definisani tipovi, kao i složeni tipovi poput `pair`, `vector`, `map`, itd. Funkcije mogu takve tipove imati i kao tip povratne vrednosti. Prenos argumenta se i u ovom slučaju vrši kao i za osnovne tipove. Funkcija `kreiraj_razlomak` od dva cela broja kreira i vraća objekat tipa `razlomak`:

```
struct razlomak {
   int brojilac, imenilac;
};

razlomak kreiraj_razlomak(int brojilac, int imenilac) {
   razlomak rezultat;
   rezultat.brojilac = brojilac;
   rezultat.imenilac = imenilac;
   return rezultat;
}
```

Navedeni primer pokazuje i da ne postoji konflikt između imena parametara i istoimenih članova strukture. Naime, imena članova strukture su uvek vezana za ime promenljive (u ovom primeru `rezultat`).
Sledeći primer ilustruje funkcije sa parametrima i povratnim vrednostima koji su tipa strukture:

```
razlomak saberi_razlomke(const razlomak& a, const razlomak&
 ↪  b) {
   razlomak c;
   c.brojilac = a.brojilac*b.imenilac +
 ↪  a.imenilac*b.brojilac;
   c.imenilac = a.imenilac*b.imenilac;
   return c;
}
```

Primećujemo da smo, efikasnosti radi, ulazne parametre umesto po vrednosti, preneli po konstantnoj referenci.
Prenos (statičkih) nizova u funkciju je veoma specifičan. Naime, pošto nizovi po pravilu zauzimaju veću količinu memorije od primitivnih tipova podataka, još od programskog jezika C odlučeno je da se prilikom prenosa nizova u funkciju oni ne kopiraju, već da se funkciji samo prenese adresa početka niza. Pokazivač na početak niza se ne koristi eksplicitno (već je zadat imenom niza), ali prenos nizova suštinski jeste prenos po adresi, tj. preko pokazivača. Razmotrimo naredni primer.

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

// za dati broj n > 0, u niz stepeni upisuje vrednosti [2^0,
 ↪  2^1, ..., 2^{n-1}]
void stepeni_dvojke(int stepeni[], int n) {
   stepeni[0] = 1;
   for (int i = 1; i < n; i++)
       stepeni[i] = 2 * stepeni[i-1];
}

int main() {
   const int N = 20;
   int stepeni[N];
   stepeni_dvojke(stepeni, N);
   for (int i = 0; i < N; i++)
```

```
        cout << stepeni[i] << " ";
    cout << endl;
}
```

Iako se u deklaraciji ne koristi ni simbol & koji bi ukazao na prenos po referenci, ni simbol * koji bi ukazao na prenos po adresi, u funkciju stiže adresa početka niza definisanog u funkciji `main`, ne pravi se kopija niza i funkcija `stepeni_dvojke` sve vreme radi sa originalnim nizom, što znači da će stepene dvojke upisati u originalni niz, pa će prethodni program ispravno ispisati sve stepene dvojke od 2^0 do 2^{19}. Ovaj način definisanja funkcija koje vraćaju niz vrednosti je veoma tipičan za jezik C: funkciji se prosledi niz (unapred alociran u pozivaocu) i broj elemenata, a funkcija onda popunjava sadržaj tog niza.
Alternativa korišćenju nizova je upotreba vektora, pri čemu vektor (kao i bilo koji drugi tip) može biti vraćen kao povratna vrednost.

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

// za dati broj n > 0 vraca niz stepena [2^0, 2^1, ...,
 ↪  2^{n-1}]
vector<int> stepeni_dvojke(int n) {
    vector<int> stepeni(n);
    stepeni[0] = 1;
    for (int i = 1; i < n; i++)
        stepeni[i] = 2 * stepeni[i-1];
    return stepeni;
}

int main() {
    const int N = 20;
    vector<int> stepeni = stepeni_dvojke(N);
    for (int i = 0; i < N; i++)
        cout << stepeni[i] << " ";
    cout << endl;
}
```

Ovaj program je sporiji, jer zahteva da se unutar funkcije `stepeni_dvojke` izvrši rezervisanje memorijskog prostora za `vector`, što je sporije nego rezervisanje memorijskog prostora za statički niz, kao u prvoj verziji programa (videti poglavlje 8.2.2). Ipak, razlika u brzini nije uvek toliko značajna da bi opravdala korišćenje statičkih nizova. Prednosti

vektora su to što njihov broj elemenata ne mora biti poznat prilikom pisanja tj. prevođenja programa i dobija se program koji je mnogo fleksibilniji i u kom je mogućnost nastajanja grešaka manja. Naglasimo da je jedino kreiranje vektora sporije nego kreiranje statičkog niza – jednom kada se vektor kreira, tj. kada se odvoji potrebna memorija, dalje operacije nad vektorom se izvršavaju praktično istom brzinom kao nad nizom.
U ranijim verzijama jezika C++, vraćanje objekata (pa i vektora) iz funkcije je zahtevalo kopiranje tih objekata (vektor `stepeni` koji je kreiran u funkciji `stepeni_dvojke` bi se kopirao u vektor `stepeni` u funkciji `main`), nakon čega bi se vektor `stepeni` alociran u funkciji `stepeni_dvojke` oslobađao. Međutim, u novim verzijama jezika garantuje se da se elementi vektora neće kopirati, već će vektor u funkciji `main` praktično preuzeti već postojeće elemente. O ovom i sličnim mehanizmima kopiranja više reči će biti u narednim tomovima ove knjige.

7.10 Rekurzivne funkcije – osnovni pregled

Funkcija može da poziva druge funkcije. Funkcija može da pozove i samu sebe (u tom slučaju argumenti funkcije obično se razlikuju od argumenata u pozivu). Da bi se izvršavanje funkcije završavalo, potrebno je da postoji slučaj u kojem se ne vrši rekurzivni poziv. Naredna rekurzivna funkcija izračunava vrednost x^n, na osnovu poznatih matematičkih veza:

$$x^n = \begin{cases} 1, & n = 0 \\ x \cdot x^{n-1}, & n > 0 \end{cases}$$

```
double stepen(double x, unsigned n) {
  if (n == 0)
    return 1.0;
  else
    return x*stepen(x, n-1);
}
```

Za vrednosti argumenta n veće od nule vrši se rekurzivni poziv, a za vrednost 0 – vrednost funkcije izračunava se neposredno. Na primer, $stepen(2,3) = 2 \cdot stepen(2,2) = 4 \cdot stepen(2,1) = 8 \cdot stepen(2,0) = 8 \cdot 1 = 8$.
Funkcije koje pozivaju same sebe zovemo *rekurzivne funkcije*. Korišćenjem rekurzije može se napisati i efikasnija funkcija za stepenovanje.

```
double stepen(double x, unsigned n) {
  if (n == 0)
```

```
    return 1.0;
  else if (n % 2 == 0)
    return stepen(x*x, n / 2);
  else
    return x*stepen(x, n-1);
}
```

Na primer, $stepen(2, 12) = stepen(4, 6) = stepen(16, 3) = 16 \cdot stepen(16, 2) = 16 \cdot stepen(256, 1) = 16 \cdot 256 \cdot stepen(256, 0) = 16 \cdot 256 \cdot 1 = 4096$. Ovaj algoritam bilo bi teže implementirati bez korišćenja rekurzije.
Rekurzija je veoma važna tehnika konstrukcije algoritama i programiranja i o njoj će mnogo više reči biti u drugom tomu ove knjige.

7.11 Doseg, životni vek i organizacija memorije

U prisustvu više funkcija, postavlja se prirodno pitanje gde je poželjno deklarisati promenljive (unutar funkcija, van funkcija, na početku tela funkcija, unutar tela funkcija i slično), koliko dugo te promenljive zauzimaju memorijski prostor i kako su raspoređene po memoriji dodeljenoj našem programu.

7.11.1 Doseg identifikatora

Jedna od karakteristika dobrih programa je da se promenljive većinom deklarišu u funkcijama (ili čak nekim užim blokovima), čime je njihova upotreba ograničena na te funkcije (ili blokove). Ovim se smanjuje zavisnost između funkcija i ponašanje funkcije određeno je samo njenim ulaznim parametrima, a ne nekim globalnim stanjem programa. Time se omogućava i da se analiza rada programa zasniva na analizi pojedinačnih funkcija, nezavisnoj od konteksta celog programa. Ipak, u nekim slučajevima prihvatljivo je da funkcije međusobno komuniciraju korišćenjem zajedničkih promenljivih.
Doseg identifikatora ili *vidljivost identifikatora* (engl. *scope of identifiers*) predstavlja deo teksta programa u kojem je određeni identifikator vidljiv, tj. u kojem ga je moguće koristiti i u kojem taj identifikator identifikuje određeni objekat (na primer, promenljivu ili funkciju). Doseg je zadat načinom i mestom u izvornom kodu u kojem je identifikator uveden. Svaki identifikator ima neki doseg. Jezik C++ spada u grupu jezika sa statičkim pravilima dosega, što znači da se doseg svakog identifikatora može jednoznačno utvrditi analizom izvornog koda (bez obzira na moguće tokove izvršavanja programa). U jeziku C++ postoji nekoliko vrsta dosega, od kojih su najznačajnije:

- *doseg datoteke* (engl. *file scope*) koji podrazumeva da ime važi od tačke uvođenja do kraja datoteke;
- *doseg bloka* (engl. *block scope*) koji podrazumeva da ime važi od tačke uvođenja do kraja bloka u kojem je uvedeno.

Identifikatori koji imaju doseg datoteke najčešće se nazivaju *globalni*, dok se identifikatori koji imaju doseg bloka nazivaju *lokalni*. Na osnovu diskusije sa početka ovog poglavlja, jasno je da je poželjno koristiti identifikatore promenljivih lokalnog dosega kada god je to moguće.
Lokalne promenljive su promenljive deklarisane unutar funkcija i njih smo već koristili u funkcijama koje smo do sada prikazali. Globalne promenljive se mogu koristiti kako bi se izbegao prenos parametara u funkciju. Zamislimo, na primer, da u nekoj veb-aplikaciji sadržaj treba da se prikazuje samo ulogovanim korisnicima. Svaka funkcija proverava da li je korisnik ulogovan i ako nije – prikazuje mu informaciju da treba da se uloguje, a u suprotnom mu prikazuje odgovarajući sadržaj. To znači da svaka funkcija treba da prima parametar kroz koji dobija informaciju o tome da li je korisnik ulogovan. Jednostavnije rešenje može biti da se ta informacija čuva u globalnoj promenljivoj.

```
bool ulogovan = false;

void ulogujKorisnika(const string& korisnickoIme, const
 ↪  string& lozinka) {
   if (proveriLogovanje(korisnickoIme, lozinka))
      ulogovan = true;
}

void prvaStrana() {
  if (!ulogovan) {
     cout << "Morate biti ulogovani" << endl;
     return;
  }
  // prikaz sadrzaja prve strane
  ...
}
```

Moguće je i da globalna i lokalna promenljiva imaju isto ime. Naime, moguće je da postoji više identifikatora istog imena. Ako su njihovi dosezi jedan u okviru drugog, tada identifikator u užoj oblasti dosega sakriva identifikator u široj oblasti dosega. Na primer, u narednom programu, promenljiva `i` u petlji sakriva lokalnu promenljivu `i` inicijalizovanu na vrednost 7, koja sakriva globalnu promenljivu `i` inicijalizovanu na vrednost `10`.

```
int i = 10;

void f() {
```

```
    int i = 7;
    for (int i = 0; i < 4; i++)
      cout << i << " ";
    cout << endl;
    cout << i << endl;
}

int main() {
    f();
    cout << i << endl;
}
```

```
0 1 2 3
7
10
```

Ovim je omogućeno da prilikom uvođenja novih imena programer ne mora da brine da li je takvo ime već upotrebljeno u širem kontekstu.
Naglasimo da korišćenje globalnih promenljivih može ponekad pojednostaviti program (jer funkcije imaju manje parametara), ali dobijeni program može biti po mnogim kriterijumima lošiji. Naime, ako se ustanovi neki problem koji nastaje usled pogrešne vrednosti te promenljive, grešku treba tražiti među svim funkcijama programa, jer sve one mogu da pristupe i promene vrednost globalne promenljive. Za razliku od toga, ako se ustanovi problem usled pogrešne vrednosti lokalne promenljive, tada je dovoljno proveriti samo kôd unutar funkcije u kojoj je definisana ta lokalna promenljiva. Dodatno, funkcija koja ne koristi globalne promenljive znatno je šire upotrebljiva: ne samo za globalne promenljive, već za bilo koje argumente koji su joj prosleđeni.
Jezik C++ podržava i korišćenje objektno-orijentisanog programiranja u kom se omogućava i *doseg nivoa klase*, koji će biti ukratko opisan u poglavlju 8.

7.11.2 Životni vek objekata

Lokalna promenljiva vezana je za jedan poziv funkcije – za nju se rezerviše memorijski prostor kada je funkcija pozvana i oslobađa se čim se izvršavanje funkcije završi. Sa druge strane, globalne promenljive se koriste iz različitih funkcija i memorijski prostor za njih treba da bude rezervisan tokom čitavog izvršavanja programa (jer se ne može unapred predvideti kada će biti pozvana neka funkcija koja će koristiti tu globalnu promenljivu).
Životni vek (engl. *storage duration*) promenljive je period izvršavanja programa u kojem je za tu promenljivu rezervisan deo memorije i kada se ta promenljiva može koristiti. Postoje sledeće vrste životnog veka:

- *statički* (engl. *static*) životni vek koji znači da je objekat dostupan tokom celog izvršavanja programa;
- *automatski* (engl. *automatic*) životni vek koji najčešće imaju promenljive koje se automatski stvaraju i uklanjaju prilikom pozivanja funkcija;
- *dinamički* (engl. *dynamic*) životni vek koji imaju promenljive koje se alociraju i dealociraju na eksplicitan zahtev programera.

Životni vek nekog objekta određuje se na osnovu pozicije u kodu i načina na koji je objekat uveden. Po pravilu, lokalne promenljive imaju automatski životni vek, a globalne statički. Ipak, moguće je definisati i promenljive koje imaju lokalni doseg (mogu se koristiti samo unutar jedne funkcije), a statički životni vek (memorija za njih je odvojena i vrednost im se čuva tokom celog izvršavanja programa). To se postiže korišćenjem ključne reči `static` u sklopu deklaracije lokalne promenljive. Takve *lokalne statičke promenljive* su vezane za neku funkciju, ali čuvaju vrednost tokom različitih poziva te funkcije. Razmotrimo sledeći primer.

```
void f() {
   static int brojPoziva = 0;
   brojPoziva++;
   cout << "Broj poziva funkcije f() je " << brojPoziva <<
 ↪  endl;
}

int main() {
   f(); f(); f();
}
```

Statičkom lokalnom promenljivom postignuto je prebrojavanje poziva funkcije `f`. U trenutku prvog poziva ona je inicijalizovana na 0, a u svakom pozivu njena vrednost se uvećava za jedan i ispisuje.

Promenljive koje imaju dinamički životni vek se koriste za implementaciju tzv. dinamičkih struktura podataka i o njima će biti mnogo više reči u narednim tomovima ove knjige. Većina struktura podataka (na primer, `vector`, `string`, `map`) koristi dinamičku alokaciju memorije, međutim, to je skriveno od programera i programer ne mora da bude upoznat sa detaljima dinamičke alokacije da bi mogao da koristi ove dinamičke strukture podataka (za razliku od programskog jezika C, čija biblioteka ne sadrži implementacije ovih struktura podataka i programiranje bez eksplicitnog korišćenja dinamičke alokacije memorije je praktično nezamislivo).

7.11.3 Organizacija memorije dodeljene programu

Različit životni vek promenljivih realizuje se u fazi izvršavanja i veoma je važno pitanje kako se to tehnički realizuje. Ključna ideja je to da se promenljive različitog životnog veka smeste u različite delove memorije dodeljene programu. Iako direktan uticaj na ovo nema programer koji piše program, već kompilator i operativni sistem, razumevanje ovog mehanizma može ponekad pomoći programeru da dublje razume ponašanje programa i lakše uoči i otkloni neke greške.

Način organizovanja i korišćenja memorije u fazi izvršavanja programa može se razlikovati od jednog do drugog operativnog sistema. Tekst u nastavku odnosi se, ako to nije drugačije naglašeno, na širok spektar platformi, pa su, zbog toga, načinjena i neka pojednostavljivanja.

Kada se izvršivi program učita u radnu memoriju računara, biva mu dodeljena određena memorija i započinje njegovo izvršavanje. Dodeljena memorija organizovana je u nekoliko delova:

- *segment koda* (engl. *code segment*, *text segment*);
- *segment podataka* (engl. *data segment*);
- *stek segment* (engl. *stack segment*);
- *hip segment* (engl. *heap segment*).

U nastavku će biti opisana prva tri, dok će o hip segmentu biti više reči u narednim tomovima ove knjige, u delu posvećenom dinamičkoj alokaciji memorije.

Kao što smo najavili, podela memorije na segmente je u određenoj vezi sa životnim vekom promenljivih (o kome je bilo reči u poglavlju 7.11.2):

- promenljive statičkog životnog veka obično se čuvaju u segmentu podataka,
- promenljive automatskog životnog veka obično se čuvaju u stek segmentu,
- promenljive dinamičkog životnog veka obično se čuvaju u hip segmentu.

7.11.4 Segment koda

Fon Nojmanova arhitektura računara predviđa da se u memoriji čuvaju podaci i programi. Dok su ostala tri segmenta predviđena za čuvanje podataka, u segmentu koda se nalazi sâm izvršivi kôd programa — njegov mašinski kôd koji uključuje mašinski kôd svih funkcija programa (uključujući kôd svih korišćenih funkcija koje su povezane statički). Na nekim operativnim sistemima, ukoliko je pokrenuto više instanci istog programa, onda sve te instance dele isti prostor za izvršivi kôd, tj. u memoriji postoji samo jedan primerak koda. U tom slučaju, za svaku instancu se, naravno, zasebno čuva informacija o tome do koje naredbe je stiglo izvršavanje.

7.11.5 *Segment podataka*

U segmentu podataka čuvaju se određene vrste promenljivih koje su zajedničke za ceo program (one koje imaju statički životni vek, najčešće globalne promenljive), kao i konstantni podaci (najčešće konstantne niske). Ukoliko se istovremeno izvršava više instanci istog programa, svaka instanca ima svoj zaseban segment podataka. Na primer, u programu

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

int a;

int main() {
  int b;
  static double c;
  cout << "Zdravo" << endl;
  return 0;
}
```

u segmentu podataka će se nalaziti promenljive `a` i `c`, kao i konstantna niska `"Zdravo"` (bez navodnika). Promenljiva `b` je lokalna automatska i ona će se čuvati u segmentu steka. Ukoliko se ista konstantna niska javlja na više mesta u programu, standard jezika ne definiše da li će za nju postojati jedna ili više kopija u segmentu podataka.

7.11.6 *Stek segment*

U stek segmentu (koji se naziva i *stek poziva* (engl. *call stack*) ili *programski stek*) čuvaju se svi podaci koji karakterišu izvršavanje funkcija. Podaci koji odgovaraju jednoj funkciji (ili, preciznije, jednoj instanci jedne funkcije — jer, na primer, rekurzivna funkcija može da poziva samu sebe i da tako u jednom trenutku bude aktivno više njenih instanci) organizovani su u takozvani *stek okvir* (engl. *stack frame*). Stek okvir jedne instance funkcije obično, između ostalog, sadrži:

- argumente funkcije;
- lokalne promenljive (promenljive deklarisane unutar funkcije);
- međurezultate izračunavanja;
- adresu povratka (koja ukazuje na to odakle treba nastaviti izvršavanje programa nakon povratka iz funkcije);
- adresu stek okvira funkcije pozivaoca.

Stek poziva je struktura tipa *LIFO* (engl. last-in-first-out)[4]. To znači da se stek okvir

[4]Ime *stek* (engl. stack) je zajedničko ime za strukture podataka koje su okarakterisane ovim načinom

može dodati samo na vrh steka i da se sa steka može ukloniti samo okvir koji je na vrhu. Stek okvir za instancu funkcije kreira se onda kada funkcija treba da se izvrši i taj stek okvir se oslobađa (preciznije, smatra se nepostojećim) onda kada se završi izvršavanje funkcije.
Kako izvršavanje programa počinje izvršavanjem funkcije `main`, prvi stek okvir se kreira za ovu funkciju. Ako funkcija `main` poziva neku funkciju `f`, na vrhu steka, iznad stek okvira funkcije `main`, kreira se novi stek okvir za ovu funkciju. Ukoliko funkcija `f` poziva neku treću funkciju, onda će za nju biti kreiran stek okvir na novom vrhu steka. Kada se završi izvršavanje funkcije `f`, onda se vrh steka vraća na prethodno stanje i prostor koji je zauzimao stek okvir za `f` se smatra slobodnim (iako on neće biti zaista obrisan).
Veličina stek segmenta obično je ograničena. Zbog toga je poželjno izbegavati smeštanje jako velikih podataka na segment steka. Na primer, sasvim je moguće da u prvom programu u nastavku niz `a` neće biti uspešno alociran i doći će do greške prilikom izvršavanja programa, dok će u drugom programu niz biti smešten u segment podataka i sve će teći očekivano. Podrazumevana veličina steka zavisi od operativnog sistema i može se promeniti.

```
int main() {
    int a[1000000];
    ...
}
```

```
int a[1000000];
int main() {
    ...
}
```

Opisana organizacija steka omogućava jednostavan mehanizam međusobnog pozivanja funkcija, kao i rekurzivnih poziva.

7.11.7 *Implementacija rekurzije*

Navedeno je da je rekurzija situacija u kojoj jedna funkcija poziva sebe samu direktno ili indirektno. Razmotrimo, kao primer, funkciju koja rekurzivno izračunava faktorijel:[5]

pristupa.

[5] Vrednost faktorijela se, naravno, može izračunati i iterativno, bez korišćenja rekurzije.

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

int faktorijel(int n) {
  if (n <= 0)
    return 1;
  else
    return n*faktorijel(n-1);
}

int main() {
   int n;
   cout << "Unesi prirodan broj: " << endl;
   cin >> n;
   cout << n << "! = " << faktorijel(n) << endl;
   return 0;
}
```

Ukoliko je funkcija `faktorijel` pozvana za argument 5, onda će se na steku poziva formirati šest stek okvira (za vrednosti argumenta 5, 4, 3, 2, 1, 0), za šest nezavisnih instanci funkcije. U svakom stek okviru je drugačija vrednost argumenta `n`. No, iako u jednom trenutku ima šest aktivnih instanci funkcije `faktorijel`, postoji i koristi se samo jedan primerak izvršivog koda ove funkcije (u segmentu koda), a svaki stek okvir pamti za svoju instancu dokle je stiglo izvršavanje funkcije, tj. koja je naredba tekuća u segmentu koda.

7.12 Deklaracija i definicija funkcije

Da bi kompilator mogao da proveri ispravnost poziva funkcije (da li je naveden dobar broj argumenata i da li su argumenti i povratna vrednost odgovarajućeg tipa), on mora da ima neke informacije o funkciji u trenutku obrade njenog poziva. Mnogi savremeni programski jezici imaju kompilatore koji više puta čitaju tekst programa koji prevode (kažemo da su *višeprolazni*) i u prvom čitanju mogu da prikupe informacije o svim funkcijama, pa da u drugom čitanju obrade sve pozive funkcija. To znači da redosled definisanja funkcija može biti proizvoljan. Međutim, pravila jezika C++ (kao i njegovog prethodnika, jezika C) zahtevaju da se u tekstu programa pre svakog poziva funkcije nađu informacije o toj funkciji koje su potrebne za proveru ispravnosti poziva i prevođenje njenog poziva u mašinski kôd (ovo pravilo je istorijski nastalo zbog toga što su prvi kompilatori za jezik C tekst programa čitali samo jednom). U većini slučajeva u kojima pišemo kratke programe i to u sklopu jedne datoteke, kôd možemo organizovati

tako da se prvo navede definicija funkcije, a zatim da se u narednim funkcijama ranije definisana funkcija poziva (tako je urađeno u svim dosadašnjim primerima). Međutim, to rešenje nije uvek moguće.

7.12.1 Uzajamna rekurzija

Kada postoji više funkcija u programu i kada postoje njihove međuzavisnosti, može biti veoma teško ili nemoguće poređati njihove definicije na način koji omogućava prevođenje (sa proverom tipova argumenata). Na primer, moguće je zamisliti situaciju u kojoj je dopušteno da funkcija `A` koristi i poziva funkciju `B`, a da funkcija `B` koristi i poziva funkciju `A` (kažemo da su funkcije `A` i `B` uzajamno rekurzivne). U tom slučaju, ni redosled definisanja `A`, pa `B`, ni redosled definisanja `B`, pa `A` nisu ispravni, jer se u oba slučaja unutar prve funkcije poziva druga funkcija koja nije još definisana.
Razmotrimo, kao primer, program koji obrađuje spisak datoteka na disku. Na disku su podaci smešteni u datotekama, koje se grupišu u direktorijume. Direktorijumi mogu da sadrže datoteke, ali i druge direktorijume. Stoga je pogodno uvesti pojam stavke koja će istovremeno predstavljati i datoteke i direktorijume. Možemo definisati tip podataka `Stavka`:[6]

```
const int DATOTEKA = 1;
const int DIREKTORIJUM = 2;

struct Stavka {
   int tip;
   string ime;
   vector<Stavka> sadrzaj;
};
```

Prirodno je onda definisati zasebne funkcije koje obrađuju datoteke i direktorijume, ali potrebno je definisati i funkciju koja obrađuje stavku tako što analizira njen tip i na osnovu toga poziva odgovarajuću funkciju za obradu. Pretpostavimo da će se obrada datoteka vršiti samo tako što će se ispisati ime datoteke, a da će se kod direktorijuma dodatno obrađivati sve stavke koje taj direktorijum sadrži. Definišimo ove funkcije na sledeći način.

```
void obradiDatoteku(const Stavka& datoteka) {
   cout << "Datoteka: " << datoteka.ime << endl;
}
```

[6] Konstante `DATOTEKA` i `DIREKTORIJUM`, umesto kao nezavisne vrednosti `int`, bolje je uvesti kao nabrojivi tip, korišćenjem ključne reči `enum`, o čemu će biti reči u daljem tekstu.

```
void obradiDirektorijum(const Stavka& direktorijum) {
   cout << "Direktorijum: " << direktorijum.ime << endl;
   for (const Stavka& stavka : direktorijum.sadrzaj)
       obradiStavku(stavka);
}

void obradiStavku(const Stavka& stavka) {
   if (stavka.tip == DATOTEKA)
      obradiDatoteku(stavka);
   else if (stavka.tip == DIREKTORIJUM)
      obradiDirektorijum(stavka);
}
```

Međutim, prethodni program nije ispravan, jer se u funkciji za obradu direktorijuma poziva funkcija `obradiStavku` koja još nije definisana. Pomeranje definicije funkcije `obradiStavku` na početak ne bi pomoglo, jer se u njoj poziva funkcija `obradiDirektorijum`.
Da bi se ova situacija mogla razrešiti, potrebno je primetiti da prevodiocu nije potrebno da poznaje celokupnu definiciju funkcije da bi mogao da proveri ispravnost njenog pozivanja. Dovoljno je da zna njeno ime, broj i tipove parametara i tip povratne vrednosti. To je sve sadržano u *deklaraciji* funkcije (kažemo i *prototip funkcije*). U prethodnom kodu je moguće prvo navesti deklaracije svih funkcija, nakon čega njihove definicije mogu biti navedene u proizvoljnom redosledu (dovoljno bi bilo i dodati samo prototip funkcije `obradiStavku`, jer se samo ona poziva pre nego što je definisana).

```
void obradiDatoteku(const Stavka& datoteka);
void obradiDirektorijum(const Stavka& direktorijum);
void obradiStavku(const Stavka& stavka);

void obradiDatoteku(const Stavka& datoteka) {
   cout << "Datoteka: " << datoteka.ime << endl;
}

void obradiDirektorijum(const Stavka& direktorijum) {
   cout << "Direktorijum: " << direktorijum.ime << endl;
   for (const Stavka& stavka : direktorijum.sadrzaj)
       obradiStavku(stavka);
}
```

```
void obradiStavku(const Stavka& stavka) {
   if (stavka.tip == DATOTEKA)
      obradiDatoteku(stavka);
   else if (stavka.tip == DIREKTORIJUM)
      obradiDirektorijum(stavka);
}
```

Deklaracija funkcije ima sledeći opšti oblik:

```
tip ime_funkcije(niz_deklaracija_parametara);
```

Definicija funkcije ima sledeći opšti oblik:

```
tip ime_funkcije(niz_deklaracija_parametara) {
   naredbe
}
```

Definicija funkcije mora da bude u skladu sa navedenim prototipom, tj. moraju da se podudaraju tipovi povratne vrednosti i tipovi parametara. Deklaracija ukazuje prevodiocu da će u programu biti korišćena funkcija sa određenim tipom povratne vrednosti i parametrima određenog tipa. Zahvaljujući tome, kada prevodilac (na primer, u okviru funkcije `main`) naiđe na poziv funkcije, može da proveri da li je njen poziv ispravan (čak i ako je definicija funkcije nepoznata u trenutku te provere). Pošto prototip služi samo za proveravanje tipova u pozivima, nije neophodno navoditi imena parametara, već je dovoljno navesti njihove tipove (mada dobro odabrana imena parametara često oslikavaju njihovu namenu i doprinose čitljivosti). U navedenom primeru, dakle, prototip je mogao da bude i

```
void obradiStavku(const Stavka&);
```

Deklaracija ili definicija funkcije mora biti navedena u kodu pre prvog poziva te funkcije.
Postojanje dve iste deklaracije iste funkcije u okviru jednog programa je dozvoljeno, pa i postojanje dve deklaracije funkcije istog imena, a različitih lista parametara. Postojanje dve definicije funkcije istog imena i sa istom listom parametara u jednom programu dovodi do greške tokom prevođenja ili povezivanja.

7.12.2 *Razdvojena kompilacija i povezivanje*

Još jedna prednost navođenja deklaracija funkcije je odvojena kompilacija programa koji su podeljeni u više datoteka. Lako je zamisliti scenario u kom želimo da neku grupu funkcija koristimo u više programa. Jedna mogućnost je da definicije tih funkcija

izdvojimo u zasebnu datoteku koju onda uključujemo (direktivom `#include`) u svaki program u kome je ona potrebna. Na primer, možemo napraviti datoteku `cifre.cpp` koja će sadržati definiciju funkcije koja izračunava zbir cifara (uz neke druge funkcije koje rade sa ciframa broja).

```
int zbir_cifara(int n) {
   int cifra, zbir = 0;
   do {
      cifra = n % 10;
      zbir += cifra;
      n /= 10;
   } while (n > 0);
   return zbir;
}
```

Tada program koji koristi ovu funkciju može izgledati ovako (pretpostavljamo da će datoteka `cifre.cpp` biti smeštena u isti direktorijum kao i datoteka u kojoj je sačuvan naredni program).

```
#include <iostream>
#include "cifre.cpp"
using namespace std;

int main() {
   int a;
   cin >> a;
   cout << zbir_cifara(a) << endl;
}
```

Direktiva `#include` prouzrokuje da se linija `#include "cifre.cpp"` zameni celokupnim sadržajem datoteke `cifre.cpp`, tj. da kompilator prevodi tekst u kom su navedene prvo definicija funkcije `zbir_cifara`, a zatim definicija funkcije `main`. Ovo rešenje će funkcionisati, ali problemi nastaju u slučaju programa koji na ovaj način koriste ogroman broj funkcija. Prilikom izmene bilo glavnog programa, bilo jedne od mnogih funkcija koje su ovako uključene u program, kompilator mora da prevodi celokupan tekst programa u kom se nalaze definicije svih funkcija iznova. To je neefikasno i kompilacija takvih programa bi trajala nedopustivo dugo (to ne znači da bi se programi sporo izvršavali, nego bi njihova kompilacija bila dugotrajna). Zbog toga je uveden mehanizam *odvojene kompilacije* i *povezivanja*.

Datoteka koja sadrži glavni program (recimo da se ona zove `test_zbir_cifara.cpp`) treba da sadrži prototip funkcije `zbir_cifara` i to pre funkcije `main`. Ovo se obično

realizuje tako što se uz datoteku `cifre.cpp` koja sadrži definicije funkcija za rad sa ciframa broja kreira i *datoteka zaglavlja* `cifre.hpp` (ili `cifre.h`) koja sadrži samo deklaracije tih funkcija. Ta datoteka bi sadržala naredni kôd:

```
int zbir_cifara(int n);
```

Ta datoteka onda se uključuje i u datoteku `test_zbir_cifara.cpp` i u datoteku `cifre.cpp`. U datoteku `cifre.cpp` uključuje se da bi se obezbedilo da je ova deklaracija u skladu sa definicijom funkcije `zbir_cifara` (a i jer neka druga funkcija u `cifre.cpp` možda koristi funkciju `zbir_cifara`, a definisana je pre nje). Datoteka `cifre.hpp` se uključuje u datoteku `test_zbir_cifara.cpp` da bi prevodilac mogao da proveri i prevede poziv funkcije `zbir_cifara` unutar funkcije `main`.
Sadržaj datoteke `cifre.cpp`.

```
#include "cifre.hpp"

int zbir_cifara(int n) {
   int cifra, zbir = 0;
   do {
      cifra = n % 10;
      zbir += cifra;
      n /= 10;
   } while (n > 0);
   return zbir;
}
```

Sadržaj datoteke `test_zbir_cifara.cpp`.

```
#include <iostream>
#include "cifre.hpp"
using namespace std;

int main() {
   int a;
   cin >> a;
   cout << zbir_cifara(a) << endl;
   return 0;
}
```

Datoteke zaglavlja se ne prevode direktno, već samo posredno (uključivanjem u `*.cpp` datoteke). Sada se i datoteka `cifre.cpp` i datoteka `test_zbir_cifara.cpp` mogu ispravno prevesti, ali nijedna od njih nije sama za sebe dovoljna da bi se dobio izvršivi program. Izvršivi program možemo dobiti ako obe datoteke prevedemo zajedno:

```
g++ test_zbir_cifara.cpp cifre.cpp -o test_zbir_cifara
```

Ovim, međutim, nismo rešili polazni problem jer se uvek istovremeno prevode obe datoteke, iako je možda bilo izmena samo u jednoj od njih. Bolje rešenje je da se prvo prevede samo biblioteka `cifre.cpp`, zatim datoteka `test_zbir_cifara.cpp` i da se nakon toga dobijene objektne datoteke povežu. Prevođenje ovih datoteka treba da bude samo do nivoa objektnih datoteka (to su datoteke koje sadrže mašinski kôd funkcija koje su definisane u `cpp` datotekama od kojih su nastale, ali nisu još spremne za izvršavanje, jer tek treba da se povežu sa drugim objektnim datotekama), što se može postići tako što se prilikom prevođenja navede opcija `-c`. Time se dobijaju objektne datoteke koje imaju ekstenziju `*.o`. Kada se one navedu prilikom pozivanja kompilatora, vrši se njihovo povezivanje i dobija se izvršivi program:

```
g++ -c cifre.cpp
g++ -c test_zbir_cifara.cpp
g++ cifre.o test_zbir_cifara.o -o test_zbir_cifara
```

Kada se sadržaj neke od `cpp` datoteka promeni, dovoljno je samo nju ponovo prevesti i ponovo povezati program.
Opisani mehanizam pokazuje da nije ni potrebno ni poželjno `*.cpp` datoteke uključivati u druge `*.cpp` datoteke.
Opisani proces razdvojene kompilacije i povezivanja obično se automatizuje korišćenjem pomoćnih alatki. Jedna od njih je program `make`. Ako se koriste integrisana okruženja za izgradnju programa, tada se obično kreiraju *projekti* u kojima se navode `cpp` i `hpp` datoteke od kojih se program gradi, a okruženje automatski određuje postupak kojim se od njih odvojenom kompilacijom i povezivanjem dobija izvršivi program.
I prototipovi funkcija iz standardne biblioteke dati su u datotekama zaglavlja. One obično nemaju nikakvu ekstenziju, a navode se unutar zagrada oblika `<...>` poput datoteka `<algorithm>`, `<vector>`, `<string>` itd. Dakle, da bi se funkcije iz standardne biblioteke mogle ispravno koristiti, dovoljno je samo uključiti odgovarajuće zaglavlje. Međutim, nekada program uspešno prolazi kompilaciju, iako potrebno zaglavlje nije uključeno, jer ga neko drugo uključeno zaglavlje (na primer, `<iostream>`) indirektno uključuje. Na to ne treba računati, jer se takav program možda neće prevesti drugim prevodiocem ili novijom verzijom istog prevodioca. Zato ovakav kôd treba izbegavati i zaglavlja bi uvek trebalo eksplicitno uključiti (tj. pre svakog poziva funkcije trebalo bi osigurati da kompilator poznaje njen prototip).
Mnogo više reči o programima koji se sastoje iz većeg broja datoteka i odvojenoj kompilaciji biće u narednim tomovima ove knjige.

8. Strukture podataka

8.1 Korisnički definisani tipovi: nabrojivi tip, strukture, klase

Postoji svega nekoliko ugrađenih osnovnih tipova (na primer, `int`, `char`, `double`). Već nizovi, vektori, niske predstavljaju složene tipove podataka. U jeziku C++ korisnik može definisati nove tipove i to na nekoliko načina. Mogu se koristiti i nabrojivi tipovi, sa konačnim skupom vrednosti. Podaci se mogu organizovati u *strukture* (tj. *slogove*), pogodne za specifične potrebe. Na taj način se povezane vrednosti (ne nužno istog tipa) tretiraju kao jedna celina i, za razliku od nizova gde se pristup pojedinačnim vrednostima vrši na osnovu brojevnog indeksa, pristup pojedinačnim vrednostima vrši se na osnovu imena polja strukture. Moguće je definisati i *klase*, koje pored podataka sadrže i funkcije koje obrađuju te podatke (tzv. metode). Pored definisanja novih tipova, već definisanim tipovima se može pridružiti i novo ime.

8.1.1 Nabrojivi tipovi (`enum`)

U nekim slučajevima korisno je definisati tip podataka koji ima mali skup dopuštenih vrednosti. Ovakvi tipovi se nazivaju *nabrojivi tipovi*. U jeziku C++ nabrojivi tipovi se definišu korišćenjem ključne reči `enum`. Na primer:

```
enum znak_karte {
  KARO,
  PIK,
  HERC,
  TREF
};
```

Nakon navedene definicije, u programu se mogu koristiti imena `KARO`, `PIK`, `HERC`, `TREF`, umesto nekih konkretnih konstantnih brojeva, što popravlja čitljivost programa. Pri

tome, obično nije važno koje su konkretne vrednosti pridružene imenima KARO, PIK, HERC, TREF, već je dovoljno znati da su one sigurno međusobno različite i celobrojne. U navedenom primeru, KARO ima vrednost 0, PIK vrednost 1, HERC vrednost 2 i TREF vrednost 3. Moguće je i eksplicitno navođenje celobrojnih vrednosti. Na primer:

```
enum znak_karte {
  KARO = 1,
  PIK  = 2,
  HERC = 4,
  TREF = 8
};
```

Moguće je navesti i vrednosti samo za neka imena, dok se narednim automatski dodeljuju vrednosti uvećane za 1.

```
enum mesec {
  JAN = 1, FEB, MAR, APR, MAJ, JUN,
  JUL, AVG, SEP, OKT, NOV, DEC
};
```

ili

```
enum vrednost_karte {
  AS = 1, DVA, TRI, CETIRI, PET,
  SEST, SEDAM, OSAM, DEVET, DESET,
  ZANDAR = 12, KRALJICA, KRALJ
};
```

Vrednosti nabrojivih tipova nisu promenljive i njima se ne može menjati vrednost. S druge strane, promenljiva može imati tip koji je nabrojiv tip i koristiti se na uobičajene načine.
Sličan efekat – uvođenja imena sa pridruženim celobrojnim vrednostima – može se postići i pretprocesorskom direktivom `#define`, ali tada ta imena ne čine jedan tip (kao kada se koristi `enum`) i lakše je napraviti grešku. (Direktiva `#define` karakteristična je za programski jezik C i u ovoj knjizi koristićemo je u veoma malom obimu.) Grupisanje u tip je pogodno zbog provera koje se vrše u fazi prevođenja.
Uz definiciju nabrojivog tipa moguće je odmah deklarisati i promenljive tog tipa (isto će važiti i za strukture, o kojima će biti reči u nastavku). Promenljive se mogu i naknadno deklarisati. Na primer,

```
  znak_karte znak;
```

Nabrojivi tipovi se često koriste da zamene konkretne brojeve u programu, na primer, povratne vrednosti funkcija. Mnogo je bolje, u smislu čitljivosti programa, ukoliko funkcije vraćaju (različite) vrednosti koje su opisane nabrojivim tipom (i imenima koja odgovaraju pojedinim povratnim vrednostima) nego konkretne brojeve. Tako, na primer, tip povratne vrednosti neke funkcije može da bude nabrojiv tip definisan na sledeći način:

```
enum status {
  USPEH,
  GRESKA_DATOTEKA,
  GRESKA_MEMORIJA,
  ISTEKLO_VREME
};
```

8.1.2 *Strukture*

Osnovni tipovi jezika C++ često nisu dovoljni za pogodno opisivanje svih podataka u programu. Ukoliko je neki podatak složene prirode, tj. sastoji se od više delova, ti njegovi pojedinačni delovi mogu se čuvati nezavisno (u zasebnim promenljivim), ali to često vodi programima koji su nejasni i teški za održavanje. Umesto toga, pogodnije je koristiti *strukture*. Za razliku od nizova, vektora i listi, koji objedinjuju jednu ili više promenljivih istog tipa, struktura objedinjuje jednu ili više promenljivih, ne nužno istih tipova. Definisanjem strukture uvodi se novi tip podataka i nakon toga mogu da se koriste promenljive tog novog tipa, na isti način kao i za druge tipove. Termin struktura se nekada koristi i za tip podataka i za konkretne instance, tj. objekte tog tipa.
Korišćenje struktura biće ilustrovano na primeru razlomaka. U jeziku C++ ne postoji tip koji opisuje razlomke, ali može se definisati struktura koja opisuje razlomke. Razlomak može da bude opisan parom koji čine brojilac i imenilac, na primer, celobrojnog tipa. Brojilac (svakog) razlomka zvaće se `brojilac`, a imenilac (svakog) razlomka zvaće se `imenilac`. Struktura `razlomak` može se definisati na sledeći način:

```
struct razlomak {
    int brojilac;
    int imenilac;
};
```

Ključna reč `struct` započinje definiciju strukture. Nakon nje, navodi se ime strukture, a zatim, između vitičastih zagrada, opis njenih *članova* (ili *polja*, *atributa*). Članovi

strukture ne mogu se koristiti kao samostalne promenljive – oni postoje samo kao deo složenijeg objekta. Prethodnom definicijom strukture uveden je samo novi tip pod imenom `razlomak`, ali ne i promenljive tog tipa.
Strukture mogu sadržati promenljive proizvoljnog tipa. Na primer, moguće je definisati strukturu koja sadrži i nisku.

```
struct student {
  string ime;
  float prosek;
};
```

Strukture mogu sadržati i nabrojive tipove.

```
struct karta {
  vrednost_karte vrednost;
  znak_karte znak;
} mala_dvojka = {DVA, TREF};
```

Definicija strukture uvodi novi tip i nakon nje se ovaj tip može koristiti kao i bilo koji drugi. Definicija strukture se obično navodi van svih funkcija. Ukoliko je navedena u okviru funkcije, onda se može koristiti samo u okviru te funkcije.

```
razlomak a, b, c;
```

Definicijom strukture je opisano da se razlomci sastoje od brojioca i imenioca, dok se navedenom deklaracijom uvode tri konkretna razlomka koja se nazivaju `a`, `b` i `c`.
Moguća je i deklaracija sa inicijalizacijom, pri čemu se inicijalne vrednosti za članove strukture navode između vitičastih zagrada:

```
razlomak a = {1, 2};
```

Redosled navođenja inicijalizatora odgovara redosledu navođenja članova strukture. Dakle, navedenom deklaracijom je uveden razlomak `a` čiji je brojilac 1, a imenilac 2.
Definisanje strukture i deklarisanje i inicijalizacija promenljivih može se (isto važi i za nabrojive tipove) uraditi istovremeno (otuda i neuobičajeni simbol `;` nakon zatvorene vitičaste zagrade prilikom definisanja strukture):

```
struct razlomak {
  int brojilac;
  int imenilac;
} a = {1, 2}, b, c;
```

Članu strukture se pristupa preko imena promenljive (čiji je tip struktura) iza kojeg se navodi tačka, a onda ime člana, na primer:

```
a.imenilac
```

Na primer, vrednost promenljive a tipa `razlomak` može biti ispisana na sledeći način:

```
cout << a.brojilac << "/" << a.imenilac << endl;
```

Naglasimo da je operator `.`, iako binaran, operator najvišeg prioriteta (istog nivoa kao male zagrade i unarni postfiksni operatori).
Ne postoji konflikt između imena polja strukture i istoimenih promenljivih, pa je naredni kôd korektan.

```
int brojilac = 5, imenilac = 3;
a.brojilac = brojilac; a.imenilac = imenilac;
```

Strukture mogu biti ugnežđene, tj. članovi struktura mogu biti druge strukture. Na primer:

```
struct dvojni_razlomak {
    razlomak gore;
    razlomak dole;
};
```

Od ranije prikazanih operacija, nad promenljivim tipa strukture dozvoljene su operacije dodele, a nisu dozvoljeni aritmetički i relacijski operatori, osim ako ih programer sam ne definiše (što će biti prikazano u nastavku).
Operator `sizeof` se može primeniti i na ime tipa i na promenljive tog tipa i u oba slučaja dobija se broj bajtova koje taj tip ili promenljiva zauzimaju u memoriji. Ovaj operator može se primeniti i na složene tipove, pa i na strukture. Napomenimo da tada broj može nekada biti i veći od zbira veličina pojedinačnih polja, jer se zbog uslova poravnanja

(engl. alignment), o kojem će više biti reči u narednim tomovima ove knjige, ponekad između dva uzastopna polja strukture ostavlja prazan prostor.
Često postoji povezana skupina složenih podataka. Umesto da se oni čuvaju u nezavisnim nizovima (što bi vodilo programima teškim za održavanje), bolje je koristiti vektore (ili nizove) struktura. Na primer, ako je potrebno imati podatke o imenima i broju dana meseci u godini, moguće je te podatke čuvati u vektoru sa brojevima dana i u (nezavisnom) vektoru imena meseci. Bolje je, međutim, opisati strukturu `opis_meseca` koja sadrži broj dana i ime:

```
struct opis_meseca {
    string ime;
    int broj_dana;
};
```

i koristiti vektor ovakvih struktura, koji se može odmah i inicijalizovati[1]:

```
vector<opis_meseca> meseci = {
    {"", 0},
    {"januar", 31},
    {"februar", 28},
    {"mart", 31},
    ...
    {"decembar", 31}
};
```

(na početnoj poziciji nalazi se element koji se ne koristi, da bi se meseci mogli referisati po svojim rednim brojevima). Nakon navedene deklaracije, ime prvog meseca u godini se može dobiti sa `meseci[1].ime`, njegov broj dana sa `meseci[1].broj_dana` itd., a broj elemenata vektora sa `meseci.size()`.
Umesto vektora, moguće je koristiti i statički niz struktura. On se deklariše na uobičajen način, na primer `opis_meseca meseci[13];`, a može se i inicijalizovati isto kao vektor (tada nije neophodno navesti broj elemenata niza):

```
opis_meseca meseci[] = {
    {"", 0},
    {"januar", 31},
    ...
    {"decembar", 31}
```

[1] U ovom primeru se zanemaruje činjenica da februar može imati i 29 dana.

```
};
```

Elementima niza se pristupa na isti način (na primer, `meseci[1].ime`), a broj elemenata ovako inicijalizovanog niza može se izračunati izrazom

```
sizeof(meseci)/sizeof(opis_meseca)
```

ili, od verzije C++17 kraće i jasnije, pozivom `size(meseci)`.
Strukture mogu da sadrže i funkcije, koje se onda nazivaju *metode* (to su funkcije članice struktura ili klasa, koje služe za rad sa podacima koje se čuvaju u strukturi ili klasi). Na primer, struktura za predstavljanje razlomka može sadržati i metodu za ispis razlomka.

```
struct Razlomak {
   int brojilac;
   int imenilac;

   void ispisiSe() {
        cout << brojilac << "/" << imenilac << endl;
   }
};
```

Kada definišemo konkretan razlomak, možemo pozvati metodu za ispis (kažemo da na taj način šaljemo poruku razlomku da se ispiše).

```
Razlomak a = {1, 2};
a.ispisiSe();
```

Posebna je metoda koja se zove isto kao i struktura i koja služi da inicijalizuje polja strukture. Nju nazivamo *konstruktor*. Pored postavljanja vrednosti polja, u konstruktoru možemo, na primer, izvršiti skraćivanje razlomka (za to možemo upotrebiti funkciju `gcd` iz zaglavlja `<numeric>`, koja izračunava najveći zajednički delilac dva data broja). Pretpostavljamo da imenilac nije nula, a znak razlomka čuvamo u brojiocu, tj. imenilac je uvek pozitivan.

```
struct Razlomak {
   int brojilac;
   int imenilac;

   Razlomak(int b, int i) {
```

```
        // imenilac cuvamo kao pozitivan broj
        if (i < 0) {
            b = -b;
            i = -i;
        }
        int nzd = gcd(b, i);
        brojilac = b / nzd;
        imenilac = i / nzd;
    }

    void ispisiSe() {
         cout << brojilac << "/" << imenilac << endl;
    }
};
```

Tada se struktura može kreirati i inicijalizovati pozivom konstruktora, na sledeći način.

```
Razlomak a(2, 4);
a.ispisiSe();
```

Ovaj program ispisuje tekst `1/2`.
Posebna vrsta metoda su operatori. Na primer, možemo definisati operatore `<` i `==`, kojima poredimo vrednosti razlomaka. Operatori se definišu slično kao i obične metode, jedino im ime mora biti `operatorXYZ`, gde je `XYZ` simbol operatora (npr. `operator+`, `operator*`, `operator<`, `operator==`). Poređenje realizujemo svođenjem na zajednički imenilac, pronalaženjem NZS imenilaca (bibliotečkom funkcijom `lcm` iz zaglavlja `<numeric>`), pretpostavljajući da pritom neće doći do prekoračenja.

```
struct Razlomak {
    ...

    // provera da li je ovaj razlomak jednak drugom
    bool operator==(const Razlomak& drugi) const {
        // nzs dva imenioca
        int nzs = lcm(imenilac, drugi.imenilac);
        // poredimo brojioce prosirenih razlomaka
        return brojilac*(nzs/imenilac) ==
               drugi.brojilac*(nzs/drugi.imenilac);
    }
```

```
    // provera da li je ovaj razlomak manji od drugog
    bool operator<(const Razlomak& drugi) const {
        // nzs dva imenioca
        int nzs = lcm(imenilac, drugi.imenilac);
        // poredimo brojioce prosirenih razlomaka
        return brojilac*(nzs/imenilac) <
               drugi.brojilac*(nzs/drugi.imenilac);
    }
};

int main() {
    Razlomak a(3, 4), b(6, 8);
    if (a == b) cout << "Jednaki su" << endl;
    else cout << "Nisu jednaki" << endl;
}
```

Ključna reč `const` iza liste parametara operatora označava da operator ne menja objekat nad kojim je pozvan. Inače se operatori ne mogu koristiti nad `const` objektima, npr. u `set<Razlomak>`.

8.1.3 *Klase*

Struktura ima svojstvo da se svim poljima i svim metodama svake njene instance (promenljive tipa strukture), ukoliko nije drugačije rečeno (korišćenjem ključne reči `private`), može pristupati iz bilo kojeg dela programa. To može da olakša korišćenje struktura, ali može da omogući neželjene upotrebe instanci strukture i time omogući loš dizajn programa. Dizajn programa može biti kvalitetniji ako se interna reprezentacija podataka koji čine neki tip može sakriti od korisnika tog tipa. Na primer, nije potrebno da korisnik strukture `Tacka` zna da li je tačka interno predstavljena pomoću Dekartovih ili polarnih koordinata. To omogućava da se u nekom trenutku interna reprezentacija podataka i algoritmi za rad sa njima promene (na primer, optimizuju), bez uticaja na kôd u kojem se tip podataka koristi. Ograničen pristup podacima može i da čuva neka njihova svojstva. Na primer, ako korisnik ne može da menja polja razlomka nakon što je razlomak konstruisan, tada bismo bili sigurni da su brojilac i imenilac uvek skraćeni (jer se u konstruktoru skraćuju) i tada bi se poređenje dva razlomka moglo izvršiti jednostavnim poređenjem brojilaca i imenilaca. Dodatno, može biti pogodno da su neka polja i neke metode dostupni korisniku definisanog tipa, a neka polja i neke metode nisu. Takve mogućnosti (i još mnoge druge) pružaju *klase*. Uobličavanje podataka i metoda koji ih obrađuju u vidu klasa jedna je od centralnih ideja objektno-orijentisanih jezika kao što je C++. U ovoj knjizi, međutim, neće biti prezentovana ni korišćena mnoga svojstva objektno-orijentisanog programiranja (kao što su *nasleđivanje*, *polimorfizam*, *apstrakcija*), već će se koristiti (i to u ograničenoj

meri) samo *enkapsulacija* – upravo opisano svojstvo objedinjavanja podataka i metoda koji ih obrađuju u celine, a kojima pristup može biti ograničen na različitim nivoima. Dodatno, u ovoj knjizi nećemo se baviti ni definisanjem *destruktora*, operatora dodele ni drugim sličnim konceptima objektno-orijentisanog programiranja.
U jeziku C++ identifikatori mogu imati i *doseg nivoa klase*, koji predstavlja kompromis između globalnih i lokalnih promenljivih. Metode koje pripadaju klasi mogu da pristupe svim promenljivim koje su deo te klase. Moguće je definisanje više objekata (instanci klase) i svaki od njih ima svoju zasebnu kopiju tih promenljivih, a koristi iste funkcije za rad sa njima.
Za razliku od struktura u kojima je, ukoliko nije traženo drugačije (korišćenjem ključne reči `private`), pristup svim poljima i svim metodama javan, u klasama je pristup svim poljima i svim metodama privatan, ukoliko nije traženo drugačije (korišćenjem ključne reči `public`).
Atributi i metode navedeni u sekciji `private` su *privatni* i može im se pristupiti samo iz koda koji se nalazi unutar klase, dok su oni navedeni u sekciji `public` javni i može im se pristupiti iz bilo kog dela koda. Na primer, klasa `Razlomak` može sakriti pristup brojiocu i imeniocu, kao i metodi za skraćivanje razlomka koja će biti pozivana svaki put kada se razlomak konstruiše.

```
class Razlomak {
   private:
      int brojilac, imenilac;

      void skrati() {
         // imenilac cuvamo kao pozitivan broj
         if (imenilac < 0) {
            brojilac = -brojilac;
            imenilac = -imenilac;
         }
         int nzd = gcd(brojilac, imenilac);
         brojilac /= nzd;
         imenilac /= nzd;
      }

   public:
      Razlomak(int b, int i) {
          brojilac = b;
          imenilac = i;
          skrati();
      }
```

```
    void ispisiSe() {
        cout << brojilac << "/" << imenilac;
    }

    bool operator==(const Razlomak& drugi) const {
       return brojilac == drugi.brojilac &&
              imenilac == drugi.imenilac;
    }
};
```

Za navedeni kôd, moguće je u programu deklarisati instancu klase `Razlomak`, tj. promenljivu tipa `Razlomak`, na sledeći način (korišćenjem *konstruktora*, analogno strukturama):

```
Razlomak r(3,4);
```

Nad promenljivom `r` može se onda koristiti metoda `ispisiSe` jer je javna:

```
r.ispisiSe();
```

ali ne i metoda `skrati` (sem u okviru metoda koje čine klasu `Razlomak`). U okviru klasa mogu se definisati i operatori, što omogućava davanje specifičnog značenja operatorima kada se primene na instance klase. U navedenom primeru, definisan je operator `==`, čime je omogućeno da se dve instance tipa `Razlomak` porede na elegantan način: `r1 == r2`, kao da se radi o osnovnim tipovima (za koje je ovaj operator predefinisan).
Primetimo da se u prethodnom tekstu govori o *metodama*, a ne o *funkcijama*. Ovi pojmovi su, zapravo, vrlo bliski – i metode i funkcije vrše nekakva izračunavanja i nekakve obrade podataka. Razlika je u tome što se metoda definiše u okviru neke klase i može da se primenjuje samo na instancama te klase (što znači da se ime metode navodi iza imena instance i simbola tačke), pri čemu može da ima i dodatne parametre. S druge strane, funkcije se definišu van klasa i ne primenjuju se neposredno na instance klase, mada, naravno, mogu da imaju i takve parametre. Postoji očigledna sintaksička razlika u pozivanju metoda i funkcija. Na primer, metoda `ispisiSe` koristi se tako što se primenjuje neposredno na objekat `r`:

```
r.ispisiSe();
```

dok bi se neka funkcija `f` koja kao parametar ima razlomak pozivala na sledeći način:

```
f(r);
```

Kao što je već rečeno u poglavlju 7.5, većina složenih tipova koje smo do sada koristili (na primer, `string` i `vector`) definisana je u vidu klasa, a funkcije koje smo uz njih koristili (na primer, `size()`) zapravo su metode vezane za ove klase, te ih pozivamo u obliku `a.size()`.

8.1.4 *Parovi i torke (tipovi `pair<T1, T2>` i `tuple<T1, ..., Tn>`)*

Često je potrebno da u programu upamtimo uređen par ili neku malo širu *n*-torku elemenata (kraće se kaže samo „torku“, engl. tuple), ne obavezno istog tipa. Jedan način da se to uradi je da se definiše novi tip strukture isključivo za trenutne potrebe. Sa druge strane, jezik C++ daje bibliotečku podršku za parove i torke i nekada je jednostavnije iskoristiti je, pogotovo što bibliotečki parovi i torke imaju definisane i neke korisne operacije (poput poređenja).

Par se realizuje tipom `pair<T1, T2>`, gde je `T1` tip prvog, a `T2` tip drugog elementa uređenog para (na primer, `pair<string, int>` označava uređen par u kome je prvi element tipa `string`, a drugi tipa `int`). Da bi se parovi mogli koristiti, potrebno je uključiti zaglavlje `<utility>` ili neko drugo zaglavlje koje uključuje i ovo zaglavlje (na primer, zaglavlje `<map>`, koje nudi podršku za mape, o čemu će više reči biti u poglavlju 8.5). Inicijalizacija se može izvršiti na isti način kao i kod inicijalizacije strukture. Par se može izgraditi od pojedinačnih elemenata funkcijom `make_pair`. Kada je par definisan, pojedinačnim podacima možemo pristupiti korišćenjem polja `first` i `second`. Na primer,

```
pair<string, double> student = {"Petar Petrovic", 9.38};
cout << student.first << " " << student.second << endl;
```

Izdvajanje pojedinačnih elemenata para se može vršiti korišćenjem polja `first` i `second`, ali postoje i drugi načini da se to uradi.

```
pair<string, double> student = {"Petar Petrovic", 9.38};
string ime = student.first;
double prosek = student.second;
```

Drugi način je da se upotrebi funkcija `tie`, raspoloživa u zaglavlju `<tuple>`.

```
pair<string, double> student = {"Petar Petrovic", 9.38};
string ime;
```

```
double prosek;
tie(ime, prosek) = student;
```

Još elegantniji način, podržan od verzije C++17, je sledeći:

```
pair<string, double> student = {"Petar Petrovic", 9.38};
auto [ime, prosek] = student;
```

Parovi se mogu dodeljivati jedan drugom (ako su istog tipa), ali i porediti. Poređenje jednakosti se vrši operatorom `==`, a različitosti operatorom `!=`. Dva para su jednaka ako i samo ako su sve odgovarajuće komponente jednake. Definisan je i poredak parova, korišćenjem operatora `<`, `>`, `<=` i `>=`. Parovi se porede leksikografski (prvo se, podrazumevanom relacijom, porede prvi elementi, pa ako su oni jednaki, porede se drugi elementi).
Torke su predstavljene tipom `tuple<T1, ..., Tn>`, za čije je korišćenje potrebno uključiti zaglavlje `<tuple>`. Inicijalizacija se može vršiti na isti način kao i kod parova. Pojedinačnim elementima torke se može pristupiti korišćenjem funkcija `get<i>`. Na primer,

```
tuple<int, int, int> datum = {2024, 3, 29};
int godina = get<0>(datum);
```

Izdvajanje svih polja možemo uraditi funkcijom `tie`.

```
int godina, mesec, dan;
tie(godina, mesec, dan) = datum;
```

Promenljive dobijaju vrednosti redom, prema poziciji elemenata u torci.
Od verzije C++17 može se koristiti i udobnija sintaksa.

```
auto [godina, mesec, dan] = datum;
```

I torke podržavaju poređenje jednakosti i leksikografsko poređenje.

```
tuple<int, int, int> datum1 = {2024, 3, 29};
tuple<int, int, int> datum2 = {2024, 4, 17};
if (datum1 < datum2)
   cout << "prvi datum je raniji" << endl;
```

```
else if (datum2 < datum1)
   cout << "drugi datum je raniji" << endl;
else
   cout << "datumi su jednaki" << endl;
```

8.1.5 *Imenovanje tipova – `typedef`*

Moguće je kreirati nova imena postojećih tipova koristeći ključnu reč `typedef`. Na primer, deklaracija

```
typedef tuple<int, int, int> Datum;
```

uvodi ime `Datum` kao sinonim za tip `tuple<int, int, int>`. Ime tipa `Datum` se onda može koristiti u deklaracijama, eksplicitnim konverzijama i slično, na isti način kao što se koristi ime `tuple<int, int, int>`:

```
Datum prviMaj = {2024, 5, 1};
```

Novo ime tipa se navodi kao poslednje, na poziciji na kojoj se u deklaracijama obično navodi ime promenljive, a ne neposredno nakon ključne reči `typedef`. Obično se novouvedena imena tipova pišu velikim početnim slovima da bi se istakla.
Od verzije C++11 novo ime tipa može se uvesti i deklaracijom `using Datum = tuple<int, int, int>;`, koja ima isto značenje, a mnogima je čitljivija, jer se novo ime navodi na početku, slično kao kod dodele.
Deklaracijom `typedef` se ne kreira novi tip, već se samo uvodi novo ime za postojeći tip. Staro ime za taj tip se može koristiti i dalje.
Deklaracija `typedef` se može koristiti za imenovanje prilično naprednih tipova (na primer, tip pokazivača na funkcije, o kojima će više reči biti u narednim tomovima ove knjige) i u tim situacijama se novo ime tipa ne navodi na kraju, već je sastavni deo deklaracije.

```
typedef int (*PFI)(char *, char *);
```

Postoje dva osnovna razloga za korišćenje ključne reči `typedef` i imenovanje tipova. Prvi je skraćivanje koda i popravljanje čitljivosti programa, naročito u slučaju dugačkih imena tipova (na primer, imenovani tip `Datum` je mnogo čitljiviji nego tip `tuple<int, int, int>`). Drugi razlog je parametrizovanje tipova u programu da bi se dobilo na njegovoj prenosivosti i lakoj izmeni. Naime, ukoliko se `typedef` koristi za uvođenje novih imena za tipove koji su mašinski zavisni, u slučaju da se program prenosi na

drugu mašinu, potrebno je promeniti samo `typedef` deklaracije. Na primer, u zavisnosti od konkretnog računara, za imenovanje pogodnog celobrojnog tipa može se koristiti `typedef short CeoBroj`, `typedef int CeoBroj` ili `typedef long CeoBroj` i u nastavku se onda može koristiti samo tip `CeoBroj`.

8.2 Strukture podataka sa sekvencijalnim pristupom

U mnogim programima potrebno je u memoriji čuvati neku seriju elemenata, čiji se elementi učitavaju ili izračunavaju, a zatim i obrađuju redom, jedan za drugim, korišćenjem petlji. Jezik C++ nam na raspolaganje stavlja različite oblike *sekvencijalnih kolekcija* podataka tj. *sekvencijalnih kontejnera* (engl. sequential container) koji nam pružaju ovu mogućnost.
Bibliotečke strukture podataka sa sekvencijalnim pristupom (na primer, `vector<T>` i `list<T>`) su *šablonske* strukture (engl. templates), te mogu da čuvaju elemente proizvoljnog tipa `T`, koji se navodi između uglastih zagrada. Elementi proizvoljnog tipa mogu se čuvati i u statički alociranim nizovima, ali se kod njih tip elemenata navodi na drugačiji način (na primer, `int a[10];`).
U nastavku ćemo proučiti:

- statički alocirane nizove,
- tip `vector<T>` (vektore),
- tip `list<T>` (liste).

Pored nabrojanih struktura, koriste se i dinamički alocirani nizovi, koji će detaljno biti opisani u narednim tomovima ove knjige.

Kolekcija	`T[]`	`array<T>`	`vector<T>`	`list<T>`	`forward_list<T>`
veličina poznata u fazi kompilacije	da	da	ne	ne	ne
promena veličine u fazi izvršavanja	ne	ne	da	da	da
memorijski segment	stek	stek	hip	hip	hip
indeksni pristup	da	da	da	ne	ne
iteracija	oba smera	oba smera	oba smera	oba smera	udesno
ubacivanje	na kraj (dok ima mesta)	na kraj (dok ima mesta)	na kraj	da	da
izbacivanje	sa kraja	sa kraja	sa kraja	da	da
dodela	ne	da	da	da	da
prenos	adresa	kopija	kopija	kopija	kopija

Slika 8.1: Poređenje sekvencijalnih struktura podataka

Ove kolekcije se razlikuju po nekim svojim svojstvima (pre svega po tome da li zahtevaju da broj elemenata koji se mogu smestiti u kolekciju bude unapred poznat, ali i po brzini i efikasnosti nekih operacija). Na slici 8.1 rezimirane su neke karakteristike različitih sekvencijalnih kolekcija podataka koje su ukratko opisane u nastavku:

- Kod nekih kolekcija veličina (maksimalni broj elemenata koji mogu biti smešteni u kolekciju) mora biti poznata u fazi pisanja tj. prevođenja programa, a kod nekih ne.
- Neke kolekcije mogu tokom rada programa menjati veličinu (dodavanjem ili uklanjanjem elemenata).
- Kada su kolekcije definisane kao lokalne promenljive, elementi nekih kolekcija se čuvaju u stek segmentu memorije (gde se memorija zauzima i oslobađa brže, ali je ima manje), a nekih delom u stek segmentu, a delom u hip segmentu (gde se memorija zauzima i oslobađa sporije, ali je ima više). Malo preciznije, za kolekcije čija se veličina može menjati u fazi izvršavanja, u stek segmentu čuvaju se informacije o toj kolekciji koje je opisuju (na primer, koliko elemenata je alocirano i koliko elemenata se koristi), a sami elementi kolekcije se čuvaju u hip segmentu.

- Neke kolekcije dopuštaju efikasan indeksni pristup elementu (efikasan pristup elementu na datoj poziciji), a neke ne (elementu se može pristupiti samo ako se redom obilaze svi elementi od početka kolekcije, pa do traženog).
- Neke kolekcije dopuštaju nabrajanje elemenata u oba smera (od prvog ka poslednjem i od poslednjeg ka prvom), a neke samo od prvog ka poslednjem (udesno).
- Neke kolekcije dopuštaju efikasno dodavanje elemenata u kolekciju (neke na proizvoljno mesto, neke samo na kraj, a neke na kraj, ali samo dok u kolekciji ima dovoljno prostora za smeštanje tog novog elementa).
- Neke kolekcije se mogu dodeljivati novim promenljivim (pri čemu se dodelom kopira celokupan sadržaj kolekcije), a neke ne.
- Neke kolekcije se prenose u funkciju po vrednosti (pravi se kopija čitave kolekcije), osim ako programer eksplicitno ne zahteva da se prenos vrši po referenci ili preko pokazivača, a neke (nizovi) se prenose u funkciju tako što se funkciji dostavi samo adresa početka kolekcije (pokazivač na njen početak).

8.2.1 *Statički alocirani nizovi*

Najjednostavniji oblik niza u jeziku C++ je statički alociran niz, nasleđen iz programskog jezika C. Statički alocirani niz koristimo kada unapred (u trenutku pisanja i u fazi kompilacije programa) znamo tačan broj potrebnih elemenata niza ili makar gornju granicu tog broja (na primer, znamo da se neće koristiti više od 100 elemenata). Pristup elementima ovakvih nizova je veoma brz, ali se oni ne mogu proširivati, porediti, dodeljivati jedan drugom i slično.

Razmotrimo problem učitavanja 10 brojeva i njihovog ispisa u obratnom redosledu. Ja-

sno je da je potrebno upamtiti sve elemente istovremeno u memoriji da bismo mogli da ih ispišemo unazad. Za to možemo upotrebiti 10 pojedinačnih promenljivih, ali mnogo bolje od toga je upotrebiti niz. Pošto znamo da će biti učitano tačno 10 elemenata, definisaćemo statički niz dužine 10.

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

int main() {
  int brojevi[10];
  for (int i = 0; i < 10; i++)
    cin >> brojevi[i];
  for (int i = 9; i >= 0; i--)
    cout << brojevi[i] << " ";
  return 0;
}
```

Nizovi u programskom jeziku C++ deklarišu se u obliku:

`tip ime_niza[dimenzija];`

Broj elemenata niza zadat je vrednošću `dimenzija`. Na primer, deklaracija

```
int a[10];
```

uvodi niz `a` od 10 celih brojeva. Prvi element niza ima indeks 0, pa su elementi niza:

`a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9]`

Indeksi elemenata niza obično su nenegativni celi brojevi (mada je u nekim prilikama, kada se nizovi koriste u kombinaciji sa pokazivačima, dopušteno koristiti i negativne indekse, o čemu će biti reči u narednim tomovima ove knjige).
Veličina memorijskog prostora potrebnog za niz određuje se u fazi prevođenja programa, pa broj elemenata niza (koji se navodi u deklaraciji) mora biti konstantan izraz. U narednom primeru, deklaracija niza `a` je ispravna, dok su deklaracije nizova `b` i `c` neispravne:

```
int x;
char a[100+10];
int b[];
```

```
float c[x];
```

Prilikom deklaracije može se izvršiti i inicijalizacija:

```
int a[5] = { 1, 2, 3, 4, 5 };
```

Nakon ove deklaracije, sadržaj niza a jednak je:

```
1 2 3 4 5
```

Ako se koristi inicijalizacija, moguće je navesti veću dimenziju od broja navedenih elemenata (kada se početni elementi deklarisanog niza inicijalizuju zadatim vrednostima, a preostali elementi podrazumevanim vrednostima – ako postoje za odgovarajući tip). Na primer, narednom deklaracijom uvodi se statički niz od 100 elemenata od kojih je prvi jednak 3 (a preostali elementi su inicijalizovani na vrednost nula).

```
int a[100] = { 3 };
```

Svi elementi globalnih brojevnih nizova inicijalizuju se na nulu.
Navođenje dimenzije manje od broja elemenata inicijalizatora je neispravno i ne prolazi kompilaciju. Dimenziju niza je moguće izostaviti samo ako je prilikom deklaracije izvršena i inicijalizacija niza i tada se dimenzija određuje na osnovu broja elemenata u inicijalizatoru. Na primer, nakon deklaracije

```
int b[] = { 1, 2, 3 };
```

niz b ima tri elementa i njegov sadržaj jednak je:

```
1 2 3
```

Broj elemenata niza se ne čuva u okviru niza, te se ne može jednostavno pročitati iz samog niza, ali se može dobiti korišćenjem operatora `sizeof`. Kada se operator `sizeof` primeni na ime niza, rezultat je veličina niza u bajtovima. Broj elemenata može se izračunati na sledeći način:

```
sizeof(ime niza)/sizeof(tip elementa niza)
```

Od verzije C++17 broj elemenata niza može se dobiti i pozivom funkcije `size` (na primer, `size(a)`), koja je čitljivija, a i bezbednija: ako se primeni na parametar funkcije koji je

zadat kao niz (o čemu će biti reči u nastavku), umesto pogrešnog rezultata dovodi do greške prilikom prevođenja. Način zasnovan na operatoru `sizeof` treba ipak poznavati, jer se često sreće u postojećem kodu i u jeziku C.
Kada se pristupa elementu niza, indeks može da bude proizvoljan izraz celobrojne vrednosti, na primer:

```
a[i+1]
a[bk*v + k];
```

U fazi prevođenja (pa ni u fazi izvršavanja[2]) ne vrši se nikakva provera da li je indeks u granicama niza i moguće je bez ikakve prijave greške ili upozorenja od strane prevodioca pristupati i lokaciji koja se nalazi van opsega deklarisanog niza (na primer, moguće je koristiti element `a[13]`, pa čak i element `a[-1]` u prethodnom primeru). Ovo najčešće dovodi do fatalnih grešaka prilikom izvršavanja programa. S druge strane, ovakva (jednostavna) politika upravljanja nizovima omogućava veću efikasnost.
Iteracija kroz elemente niza može se vršiti pomoću indeksa, ali i pomoću skraćenog oblika petlje `for` (tzv. oblika `foreach`).

```
int a[] = {1, 2, 3};
int n = 3;

for (int i = 0; i < n; i++)
   cout << a[i] << endl;

for (int x : a)
   cout << x << endl;
```

Nizovi imaju ograničenja koja neke druge kolekcije nemaju. Statički alociran niz ne može se proširivati niti sužavati tokom rada programa, tj. sve vreme rada programa zauzima isti memorijski prostor, dovoljan za smeštanje navedenog broja elemenata. Pojedinačni elementi nizova se mogu menjati (na primer, korišćenjem naredbe dodele), ali nizovi (kao celine) nisu izmenljive vrednosti i nije im moguće dodeljivati vrednosti niti ih menjati. To ilustruje sledeći primer:

[2]Pristup elementu van granica niza dovodi do *nedefinisanog ponašanja* programa. Ako je prekoračenje veliko, pa se pristupa memoriji koja uopšte nije dodeljena programu, operativni sistem će obično nasilno prekinuti izvršavanje programa (na primer, uz poruku `segmentation fault`). Međutim, mala prekoračenja najčešće ostaju neprimećena i mogu neprimetno da izmene vrednosti drugih promenljivih, što dovodi do grešaka koje se teško otkrivaju.

```
int a[3] = {5, 3, 7};
int b[3];
b = a;      // Neispravno - nizovi se ne mogu dodeljivati.
a++;        // Neispravno - nizovi se ne mogu menjati.
```

Sadržaj dva niza se ne može porediti operatorima ==, !=, >, <, <=, >=. Upotreba ovih operatora nad imenima nizova je zapravo dopuštena, ali tada se ne poredi sadržaj nizova, već memorijske adrese na kojima počinju ti nizovi, što obično nije ono što programer očekuje i potencijalni je uzrok grešaka.

Sa druge strane, statički nizovi kreiraju se brže od ostalih kolekcija. Memorija za elemente statički alociranih nizova rezerviše se na *programskom steku* (za lokalne nizove[3]) ili u segmentu podataka (za globalne nizove), dok se za druge kolekcije memorija rezerviše delom na steku (za lokalne kolekcije) ili u segmentu podataka (za globalne kolekcije), a delom na tzv. hipu, što zahteva utrošak određenog vremena prilikom alokacije memorije, i to pre nego što korišćenje kolekcije započne. Često je ova razlika u brzini zanemarljiva, međutim, ima situacija kada može značajno doprineti ukupnoj efikasnosti programa (na primer, kada se niz alocira u funkciji koja se jako često poziva). Programski stek je često veoma mala količina memorije (tek nekoliko megabajta na današnjim računarima), pa nije moguće kreirati velike lokalne statički alocirane nizove.

Neki kompilatori podržavaju oblik nizova koji se naziva VLA (engl. variable length array), gde se kao dimenzija niza navodi promenljiva čija je vrednost poznata tek u fazi izvršavanja programa. VLA nikada nisu bili deo standarda jezika C++ i stoga ih nikako nije preporučljivo koristiti.

8.2.1.1 *Nizovi i funkcije*

Kada se kao argument funkcije navede ime niza, u funkciju se, u fazi izvršavanja, prenosi samo adresa početka niza, a ne kopira se sadržaj niza niti se prenosi informacija o broju elemenata niza (koja je u fazi kompilacije pridružena imenu niza). Pošto funkcija koja je pozvana dobija informaciju o adresi početka originalnog niza, ona može da neposredno menja njegove elemente (i takve izmene će biti sačuvane nakon izvršenja funkcije). Iako se opisani mehanizam može smatrati prenosom po vrednosti (ali ne niza, nego adrese početka niza), on je po duhu sličan prenosu po referenci. Ovakvo ponašanje doprinosi efikasnosti, ali može biti uzrok nekih grešaka.

Funkcija koja kao parametar ima niz može biti deklarisana na neki od narednih načina:

```
tip ime_funkcije(tip ime_niza[dimenzija]);
tip ime_funkcije(tip ime_niza[]);
```

[3] Ako nisu obeleženi kvalifikatorom `static`.

S obzirom na to da se u funkciju prenosi samo adresa početka niza, a ne i dimenzija niza, prvi oblik deklaracije nema puno smisla te se retko koristi. Iako se u deklaraciji funkcije koristi sintaksa koja podseća na nizove, tip podataka parametra funkcije zapravo nije niz, već adresa prvog elementa niza (ovaj mehanizam biće podrobnije objašnjen u narednim tomovima ove knjige). Pošto funkcija nije primila niz (sa pridruženom informacijom o broju elemenata), operator `sizeof` neće dati veličinu celog niza, već samo veličinu memorijske adrese. U funkciji, dakle, ne možemo znati veličinu niza koji je argument (osim ako je ne prosledimo pored niza), ne možemo za iteraciju koristiti petlju `foreach` i slično. Naredni primer ilustruje činjenicu da se u funkciju ne prenosi ceo niz, već samo adresa njegovog početka.

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

void f(int a[]) {
  cout << "f: " << sizeof(a) << endl;
}

int main() {
  int a[] = {1, 2, 3, 4, 5};
  cout << "main: " << sizeof(a) << endl;
  f(a);
  return 0;
}
```

Prilikom pokretanja programa na 64-bitnoj mašini na kojoj `int` zauzima 4 bajta, program ispisuje:

```
main: 20
f: 8
```

Ovo ukazuje na to da niz u funkciji `main` zauzima 20 bajtova (5 podataka veličine 4 bajta), dok niz koji je parametar funkcije `f` zauzima 8 bajtova (koliko na 64-bitnim sistemima zauzima jedna memorijska adresa).
Prilikom prenosa niza (tj. adrese njegovog početka) u funkciju, pored imena niza, programer može da eksplicitno prosledi i broj elemenata niza kao dodatni argument (da bi pozvana funkcija imala tu informaciju).
Povratni tip funkcije ne može da bude niz. Funkcija ne može da kreira niz koji bi bio vraćen kao rezultat, ali funkcija rezultate svog rada može da upisuje u niz koji joj je prosleđen kao argument.

U narednom programu, funkcija `ucitaj_broj` ne uspeva da učita i promeni vrednost broja `x`, dok funkcija `ucitaj_niz` ispravno unosi i menja elemente niza `y` (jer joj je poznata adresa početka niza `y`).

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

void ucitaj_broj(int a) {
   cout << "Unesi broj: ";
   cin >> a;
}

void ucitaj_niz(int a[], int n) {
  int i;
  cout << "Unesi niz: ";
  for (i = 0; i < n; i++)
     cin >> a[i];
}

int main() {
  int x = 0;
  int y[3] = {0, 0, 0};
  ucitaj_broj(x);
  ucitaj_niz(y, 3);
  cout << "x = " << x << endl;
  cout << "y = " << y[0] << ", " << y[1] << ", " << y[2] <<
 ↪  endl;
}
```

Kada se pokrene, program daje sledeći rezultat.

```
Unesi broj: 5
Unesi niz: 1 2 3
x = 0
y = 1, 2, 3
```

Uprkos nedostacima i ponašanju koje se, u jeziku C++, razlikuje od drugih tipova, statički alocirani nizovi se veoma često koriste jer imaju mnogo dužu tradiciju korišćenja (dolaze iz programskog jezika C, u kojem su praktično osnovna kolekcija podataka). Stoga je veoma važno dobro izučiti korišćenje statički alociranih nizova i sve njihove specifičnosti.

8.2.2 Tip `vector<T>`

Jedna od osnovnih karakteristika statički alociranih nizova je to da im je veličina fiksirana i ne menja se tokom izvršavanja programa. Jednom kada se ove kolekcije kreiraju, one se ne mogu proširivati novim elementima niti sužavati. To je značajno ograničenje u mnogim realnim aplikacijama u kojima u trenutku pisanja i prevođenja programa nemamo tačne informacije o tome koliko elemenata će biti potrebno smestiti tokom izvršavanja programa. Zbog toga je poželjno koristiti kolekcije čija se veličina može menjati tokom izvršavanja programa, tj. koje se mogu *dinamički realocirati*.
Osnovna sekvencijalna kolekcija koja ovo podržava je `vector<T>`, gde je `T` tip podataka koji se čuva. Dakle, vektor je parametrizovan tipom elemenata koje čuva, tako da može da se koristi `vector<int>`, `vector<double>`, `vector<string>`, ali, na primer, i `vector<vector<int>>`.
Osnovni oblik deklaracije uvodi prazan vektor. Na primer,

```
vector<int> a;   // prazan vektor
```

Može se zadati početni broj elemenata vektora. On ne mora biti konstantan (tj. ne mora biti poznat u fazi kompilacije), već može postati poznat u toku izvršavanja programa. Ukoliko u toku izvršavanja programa, a pre deklaracije vektora, potreban broj elemenata postane poznat, onda je vektor najbolje deklarisati uz navođenje dimenzije u sklopu deklaracije, jer se na taj način odmah odvaja potrebna količina memorije. Na primer,

```
int n;
cin >> n;
vector<double> a(n);  // vektor od n elemenata
```

Naravno, da bi ovo uspelo, potrebno je da broj `n` ne bude toliko veliki da smeštanje `n` elemenata prevazilazi količinu raspoložive memorije računara na kom se program izvršava[4]. U suprotnom, proizvodi se izuzetak (u fazi izvršavanja). Ako alokacija uspe, veličina vektora je `n`, a vrednost svakog elementa jednaka je 0.
Kao drugi argument deklaracije vektora moguće je navesti početnu vrednost svih elemenata.

```
int n;
cin >> n;
```

[4] Ako je vektor deklarisan kao lokalna promenljiva, opšte informacije o vektoru čuvaju se na programskom steku, a sami elementi vektora na hipu.

```
vector<double> a(n, 1.0);  // vektor od n elemenata
 ↪  postavljenih na vrednost 1.0
```

Inicijalizacija se može vršiti na isti način kao i kod nizova.

```
vector<int> brojevi = {1, 2, 3};
```

Nakon deklaracije u kojoj je naveden broj elemenata, vektor se koristi na potpuno isti način kao i niz – elementu na poziciji `i` pristupa se sa `a[i]`, pri čemu se pozicije broje od 0. Ukoliko indeks `i` nije unutar granica vektora, u fazi izvršavanja dolazi do nedefinisanog ponašanja, pa i prekida rada programa. Elementu vektora može se pristupiti i primenom metode `at`, na primer `a.at(i)`. Ako se elementima pristupa korišćenjem ove metode, u fazi izvršavanja proverava se da li je indeks `i` unutar granica vektora i, ukoliko nije, aktivira se izuzetak.
Vektori se često koriste na sledeći način:

```
int n;
cin >> n;
vector<int> a(n);
for (int i = 0; i < n; i++)
  cin >> a[i];
...
```

U navedenom kodu, prvo se učitava dimenzija (ona je, dakle, poznata na početku izvršavanja programa, ali ne i u fazi prevođenja), zatim se vektor deklariše tako da ima odgovarajući broj elemenata, nakon čega se učitavaju pojedinačni elementi.
Dimenzija (broj elemenata) vektora `a` može se odrediti izrazom `a.size()`.

```
vector<int> a = {1, 2, 3, 4};
for (int i = 0; i < a.size(); i++)
  // obradjuje se a[i]
```

Iteraciju kroz sve elemente vektora možemo vršiti i takozvanom petljom `for-each` (ili `foreach`).

```
vector<int> a = {1, 2, 3, 4};
for (int x : a)
  ...
```

Proširivanje vektora `a` dodavanjem elementa `x` na njegov kraj moguće je pozivom metode `a.push_back(x)` (time se uvećava broj elemenata za 1). Uklanjanje elementa sa kraja (tj. uklanjanje poslednjeg elementa) vrši se pozivom metode `a.pop_back()` (što, naravno, ima smisla samo kada vektor nije prazan, a u suprotnom je ponašanje nedefinisano i izvršavanje programa obično biva prekinuto). Čitanje poslednjeg elementa nepraznog vektora se vrši izrazom `a.back()`. Sve ove operacije su veoma efikasne. Vektor nema operacije za dodavanje ni za uklanjanje elementa sa početka. Čitanje prvog elementa nepraznog vektora vrši se izrazom `a.front()` (ili, naravno, `a[0]`).
Ako ne znamo unapred broj elemenata vektora, možemo ih učitavati i ubacivati u vektor jedan po jedan. U narednom programu, deklarišemo vektor koji je inicijalno prazan, učitavamo `n` brojeva i u vektor smeštamo samo parne.

```
int n;
cin >> n;
vector<int> parni;
for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    if (x % 2 == 0)
       parni.push_back(x);
}
```

Umesto metode `push_back`, može se koristiti i metoda `emplace_back`. Njoj se ne prosleđuje već napravljen element, već podaci od kojih se element gradi, i on se gradi direktno na svom mestu u vektoru (bez pravljenja privremene kopije). Ovo radi za elemente bilo kog tipa koji se može napraviti od navedenih podataka. Na primer, ako se u vektoru čuvaju uređeni parovi, umesto `v.push_back(make_pair(x, y))` može se pisati `v.emplace_back(x, y)`.
Vektor funkcioniše tako što se u startu rezerviše (*alocira*) određena količina memorije (koja može biti i veća od trenutnog broja popunjenih elemenata). Kada se sva ta memorija popuni, tada se vrši *realokacija*, koja podrazumeva da se alocira nova količina memorije (često duplo veća od prethodne) i da se prepišu elementi na novu memorijsku lokaciju. Ovo može biti sporo, međutim, zahvaljujući tome što količina memorije raste geometrijskom progresijom, realokacije su sve ređe i ređe i većina poziva metode `push_back` funkcioniše veoma brzo (jer samo upisuje element u već rezervisanu memoriju).
Eksplicitna promena dimenzije vektora se može sprovesti i nezavisno od inicijalizacije, metodom `resize` — nakon poziva `a.resize(n)` vektor `a` ima dimenziju `n` (pod uslovom da postoji dovoljno memorije). Ako se vektor proširuje, postojeći elementi se ne menjaju, a novi elementi dobijaju podrazumevanu vrednost (za brojevne tipove to je 0). Vrednost novih elemenata može se navesti kao drugi parametar metode `resize` (na

primer, ako je vektor `a` prazan, `a.resize(100, 17)` uzrokuje da vektor sadrži 100 elemenata jednakih 17).
Vektori se u funkcije prenose po vrednosti (osim ako se eksplicitno ne navede da se prenose po referenci), što znači da se u funkciju prenosi kopija vektora koji je naveden kao argument u pozivu. Ako želimo da funkcija modifikuje sadržaj vektora, neophodno ga je preneti po referenci (navođenjem simbola `&`). Tada se u funkciju šalje samo memorijska adresa tog vektora i sve operacije se sprovode nad originalnim vektorom (ne vrši se kopiranje). Kopiranjem se nepotrebno troši memorija i vreme, pa ima smisla vektore proslediti po referenci i funkcijama koje samo analiziraju njihov sadržaj i ne menjaju ih. Tada se obično navodi ključna reč `const`, čime se obezbeđuje da se vektor ne može promeniti u funkciji, iako je prenet po referenci. Na primer,

```
int zbir(const vector<int>& a) {
   int z = 0;
   for (int x : a)
      z += x;
   return z;
}
```

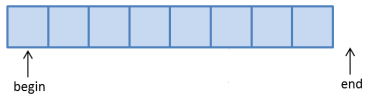
Funkcija može da vrati vektor kao svoju rezultujuću vrednost. Zahvaljujući optimizacijama koje savremene verzije jezika C++ garantuju, ne vrši se kopiranje elemenata vektora, pa se ovim ne umanjuju performanse programa.

8.2.3 *Pokazivači i iteratori*

Jedan način da se u strukturama podataka sa sekvencijalnim pristupom izdvoji neki konkretan element je upotreba indeksa, tj. pozicije tog elementa unutar kolekcije. Međutim, videćemo uskoro da kod nekih kolekcija (na primer, skupova i mapa) elementima ne pristupamo na osnovu pozicije, već na osnovu ključa, i da redosled elemenata u njima ne određuje programer, već ga kolekcija sama održava (na primer, uređujući elemente po ključu). Takođe, kod nekih sekvencijalnih kolekcija (pre svega lista, o kojima će više reči biti u poglavlju 8.2.4) pristup na osnovu indeksa je veoma neefikasan, jer elementi ne zauzimaju susedne memorijske lokacije i da bi se našao element na poziciji `n`, potrebno je obići redom sve elemente od prvog do tog traženog.
Stoga se pored indeksa za pristup elementima nizova i drugih kolekcija koriste *pokazivači* i *iteratori*. Veza između pokazivača i nizova je napredna tema, jako važna za programski jezik C i sistemsko programiranje i pokazivači će detaljno biti obrađeni u narednim tomovima ove knjige. U nastavku ćemo prikazati samo osnove korišćenja pokazivača i iteratora, u meri koja je dovoljna za korišćenje bibliotečkih kolekcija i funkcija.
Pokazivači su promenljive koje sadrže memorijske adrese. Razmotrimo, kao jedan primer, iteraciju kroz niz korišćenjem pokazivača.

```
int a[] = {1, 2, 3, 4};
int n = 4;
for (int* p = a; p < a + n; p++)
   cout << *p << endl;
```

Pošto niz a sadrži elemente tipa int, pokazivač p koji pokazuje na njegove elemente biće tipa int*. Pokazivač p se inicijalizuje tako da sadrži adresu prvog elemenata niza (inicijalizacijom p = a). Nakon toga se kao uslov petlje proverava da li je pokazivač p stigao do adrese a+n, što je adresa koja je za n elemenata niza pomerena od početka niza, tj. adresa koja se nalazi tačno iza poslednjeg elemenata niza — kada p stigne do te adrese, obrađeni su svi elementi niza i petlja može da se prekine.
U svakom koraku petlje ispisuje se element na adresi na koju pokazuje pokazivač p (do tog elementa se dolazi tzv. dereferenciranjem pokazivača, tj. izrazom *p) i zatim se pokazivač uvećava, čime se sa adrese jednog, pomera na adresu narednog elementa niza. O pokazivačima će mnogo više reči biti u narednim tomovima ove knjige.
Primetimo da smo u prethodnom primeru sabiranjem pokazivača a (adresa početka niza se može smatrati nekim vidom pokazivača) i broja n dobili novi pokazivač, koji je udaljen od polaznog za n elemenata niza (a ne za n bajtova). Slično, oduzimanjem dva pokazivača koji ukazuju na neka dva elementa niza dobija se broj elemenata niza između njih (uključujući prvi i ne uključujući poslednji). To je takozvana *pokazivačka aritmetika*.
Umesto pokazivača koji se koriste u radu sa nizovima, u radu sa bibliotečkim kolekcijama koriste se *iteratori*. To su posebni objekti koji se koriste na skoro isti način kao pokazivači (u svakom trenutku pokazuju na jedan element kolekcije).



Slika 8.2: Iteratori begin i end

Tip iteratora je određen tipom kolekcije na čije elemente taj iterator pokazuje. Na primer, tip vector<int>::iterator označava iterator koji ukazuje na elemente vektora tipa vector<int>, dok tip vector<int>::const_iterator označava iterator koji ukazuje na elemente vektora tipa vector<int> (dakle, na podatke tipa int), ali tako da se preko njega ti elementi ne mogu menjati. Sam iterator pri tome nije konstantan i može se pomerati kroz vektor. Videćemo da se, zahvaljujući ključnoj reči auto, u programima često može izbeći navođenje konkretnog tipa iteratora i zaključivanje o njegovom tipu prepustiti kompilatoru.
Neke od najčešće korišćenih funkcija deklarisanih u zaglavlju <iterator> su:

- `begin`, `end` – vraćaju iteratore koji ograničavaju opseg date kolekcije (na primer, `vector` i slično). Mnoge kolekcije podržavaju ove dve metode. Funkcija `begin` vraća iterator koji ukazuje na prvi element, a `end` vraća iterator koji ukazuje neposredno iza poslednjeg elementa (na primer, `begin(v)` vraća iterator koji ukazuje na početak vektora `v`).
- `distance` – vraća rastojanje (broj elemenata) u opsegu ograničenom sa dva iteratora koja se prosleđuju kao argumenti funkcije (prvi iterator pokazuje na početak, tj. na prvi element opsega, a drugi neposredno iza poslednjeg elementa opsega). Na primer, ako je `it` iterator koji ukazuje na neki element unutar vektora `v`, tada se njegov indeks može odrediti pomoću `distance(begin(v), it)`;
- `next` (`prev`) – vraća iterator koji pokazuje na element date kolekcije koji je posle (pre) prosleđenog iteratora, na datom rastojanju; ako se kao drugi argument ne prosledi rastojanje, podrazumevano se traži iterator na naredni (prethodni) element kolekcije. Na primer, `next(begin(v))` je iterator koji ukazuje na drugi element vektora `v` (ako takav postoji), dok je `prev(end(a), 2)` iterator koji ukazuje na pretposlednji element niza `a` (ako takav postoji).
- Nad pokazivačima i nad iteratorima se mogu primenjivati i aritmetičke operacije: `it + n` odgovara iteratoru koji se dobija kada se iterator `it` pomeri za `n` mesta unapred (isto kao i `next(it, n)`). Razlika dva iteratora određuje broj elemenata između njih (uključujući prvi i ne uključujući poslednji), isto kao i funkcija `distance`. Ove operacije su, međutim, dozvoljene samo za iteratore kod kojih je moguć efikasan indeksni pristup (iteratori nizova, vektora, niski i redova sa dva kraja). Za liste, skupove i mape (o kojima će biti reči u nastavku) aritmetičke operacije nad iteratorima nisu dozvoljene i umesto njih se koriste funkcije `next`, `prev` i `distance`. Za pokazivače i nizove važi slično: ako niz `a` ima `n` elemenata, tada se njegov opseg može zadati pokazivačima `a` i `a+n`.

Iteracija kroz vektor se može izvršiti korišćenjem iteratora, na sledeći način.

```
vector<int> a = {1, 2, 3, 4};
for (auto it = a.begin(); it != a.end(); it++)
    cout << *it << endl;
```

8.2.4 Tip `list<T>`

Vektori dopuštaju efikasno dodavanje elemenata na kraj i brisanje elemenata sa kraja, ali za rešavanje nekih zadataka su nam potrebne sekvencijalne kolekcije koje omogućavaju efikasno dodavanje elemenata na proizvoljnu poziciju i efikasno brisanje elemenata sa proizvoljne pozicije. U takvim zadacima je umesto vektora efikasnije da koristimo

liste. Lista je predstavljena tipom `list<T>`, gde je `T` tip elemenata liste. Za razliku od nizova i vektora, gde je indeksni pristup osnovni mehanizam pristupa elementima (i on je veoma efikasan), elementima liste se obično pristupa preko iteratora. Iteratore obično koristimo da bismo izveli sledeće operacije nad listom.

- Kada je poznat iterator koji ukazuje na neki element liste, tom elementu pristupamo dereferenciranjem iteratora.

```
list<int> lista = {1, 2, 3, 4};
auto it = next(lista.begin(), 3); // 3 elementa desno od
 ↪  pocetnog
cout << *it << endl;
```

- Prolazak kroz sve elemente liste vršimo ili petljom `foreach` ili korišćenjem iteratora.

```
list<int> lista = {1, 2, 3, 4};

for (int x : lista)
    cout << x << endl;

for (auto it = lista.begin(); it != lista.end(); it++)
    cout << *it << endl;
```

- Metoda `erase` briše element na koji ukazuje dati iterator i vraća iterator na element koji sledi iza obrisanog. Nakon brisanja, iterator koji je ukazivao na obrisani element više nije validan (ne sme se koristiti), pa ga je uobičajeno zameniti povratnom vrednošću metode `erase`. Ostali iteratori ostaju validni.

```
list<int> lista = {1, 2, 3, 4};
auto it = lista.begin();
it = lista.erase(it);
// sadrzaj liste je {2, 3, 4}, a it sada ukazuje na 2
```

- Metoda `insert` dodaje element pre elementa na koji ukazuje iterator i vraća iterator na umetnuti element. Dati iterator i dalje ukazuje na isti element kao pre umetanja, a i svi ostali iteratori ostaju validni.

```
list<int> lista = {1, 2, 3, 4};
auto it = lista.begin();
auto novi = lista.insert(it, 0);
// sadrzaj liste je {0, 1, 2, 3, 4},
// novi ukazuje na 0, a it i dalje ukazuje na 1
```

Naglasimo da se ne preporučuje izmena liste tokom iteracije kroz nju petljom `foreach` (isto kao i kod svih ostalih kolekcija), jer prilikom izmena iterator koji se (implicitno) koristi za iteraciju postaje neispravan.

8.3 Višedimenzioni nizovi i kolekcije

Često je umesto jednodimenzionih kolekcija potrebno koristiti dvodimenzione (na primer, matrice), pa i višedimenzione. Ponovo postoji izbor između korišćenja statički alociranih višedimenzionih nizova i korišćenja bibliotečkih kolekcija (na primer, `vector`), koje se mogu ugnežđavati (na primer, možemo napraviti vektor čiji su elementi vektori i na taj način dobiti višedimenzionu kolekciju).
Višedimenzioni nizovi se deklarišu na sledeći opšti način:

```
tip ime_niza[dimenzija_1]...[dimenzija_n];
```

Dvodimenzioni nizovi (matrice) tumače se kao jednodimenzioni nizovi čiji su elementi nizovi. Zato se elementima dvodimenzionog niza pristupa sa:

```
ime_niza[vrsta][kolona]
```

a ne sa `ime_niza[vrsta, kolona]`.
Elementi se u memoriji smeštaju po vrstama, pa se, kada se elementima pristupa u redosledu po kojem su smešteni u memoriji, najbrže menja poslednji indeks. Ovde se podrazumeva konvencija iz matematike za zapis matrica po kojoj se prvi indeks odnosi na vrste, a drugi na kolone, kao u sledećem primeru:

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \dots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & a_{1,2} & \dots & a_{1,n-1} \\ \dots & \dots & \dots & \dots & \dots \\ a_{m-1,0} & a_{m-1,1} & a_{m-1,2} & \dots & a_{m-1,n-1} \end{pmatrix}$$

Elementi navedene matrice bili bi u memoriji poređani redom $a_{0,0}$, $a_{0,1}$, $a_{0,2}$, ..., $a_{0,n-1}$, $a_{1,0}$, $a_{1,1}$, $a_{1,2}$, ..., $a_{1,n-1}$, ..., tj. `a[0][0]`, `a[0][1]`, `a[0][2]`, ..., `a[0][n-1]`, `a[1][0]`, `a[1][1]`, `a[1][2]`, ..., `a[1][n-1]`,...

Niz se može inicijalizovati navođenjem liste inicijalizatora u vitičastim zagradama; pošto su elementi opet nizovi, svaki od njih se opet navodi u okviru vitičastih zagrada (mada je unutrašnje vitičaste zagrade moguće i izostaviti). Razmotrimo, kao primer, jedan dvodimenzioni niz:

```
int a[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Kao i u slučaju jednodimenzionih nizova, ako je naveden inicijalizator, vrednost prvog indeksa moguće je i izostaviti (jer se on u fazi kompilacije može odrediti na osnovu broja inicijalizatora):

```
int a[][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

U memoriji su elementi dvodimenzionog niza poređani na sledeći način: `a[0][0]`, `a[0][1]`, `a[0][2]`, `a[1][0]`, `a[1][1]`, `a[1][2]`, tj. vrednosti elemenata niza poređane su na sledeći način: `1`, `2`, `3`, `4`, `5`, `6`. U ovom primeru, element `a[v][k]` je *i*-ti po redu, pri čemu je *i* jednako `3*v+k`. Pozicija elementa višedimenzionog niza može se slično izračunati i u slučaju nizova sa tri i više dimenzija[5].
Razmotrimo, kao dodatni primer, dvodimenzioni niz koji sadrži broj dana za svaki mesec, pri čemu su u prvoj vrsti vrednosti za obične, a u drugoj vrsti za prestupne godine:

```
int broj_dana[][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};
```

Za skladištenje brojeva koristi se tip `int` (kako su u pitanju mali prirodni brojevi, ako je potrebno štedeti memoriju, umesto tipa `int` može da se koristi tip `char`), a u nultu kolonu su upisane nule da bi se podaci za mesec `m` nalazili upravo u koloni `m` (tj. da bi se mesecima pristupalo korišćenjem indeksa 1–12, umesto indeksa 0–11). Niz `broj_dana` je moguće koristiti da bi se, na primer, promenljiva `bd` postavila na broj dana za mesec `mesec` i godinu `godina`:

[5] Nekada programeri ovu tehniku izračunavanja pozicija eksplicitno koriste da matricu smeste u jednodimenzioni niz, ali, pošto jezik dopušta korišćenje višedimenzionih nizova, za ovim nema potrebe.

```
bool prestupna = (godina % 4 == 0 && godina % 100 != 0) ||
                 godina % 400 == 0;
int bd = broj_dana[prestupna][mesec];
```

Vrednost `true` tipa `bool` se može konvertovati u `1`, a vrednost `false` u `0`, pa se vrednost `prestupna` može koristiti kao indeks pri pristupanju nizu.
Umesto statički alociranih višedimenzionih nizova možemo koristiti vektor-vektora. Na primer, matricu možemo definisati na sledeći način.

```
vector<vector<int>> A = {{1, 2}, {3, 4}};
```

Matricu dimenzije $m \times n$ popunjenu nulama možemo definisati na sledeći način.

```
int m, n;
cin >> m >> n;
vector<vector<int>> A(m, vector<int>(n, 0));
```

Vektor `A` se konstruiše tako da ima `m` elemenata, pri čemu se svaki od njih postavlja na vrednost `vector<int>(n, 0)`, što je vektor koji sadrži `n` elemenata koji se postavljaju na 0. Alternativno, alokaciju svakog pojedinačnog vektora vrste možemo napraviti i u petlji.

```
int m, n;
cin >> m >> n;
vector<vector<int>> A;
A.resize(m);
for (int i = 0; i < m; i++)
   A[i].resize(n, 0);
```

Naglasili smo već da je alokacija memorije za vektor sporija nego alokacija memorije za statički niz. U slučaju vektora vektora vrši se alokacija velikog broja vektora, pa kreiranje vektora-vektora može biti znatno sporije nego kreiranje višedimenzionog niza. Sa druge strane, za razliku od nizova kod kojih sve vrste imaju jednak broj elemenata, pomoću vektora-vektora mogu se napraviti strukture podataka u kojima svaka vrsta ima različit broj elemenata (na primer, za svakog učenika možemo imati različit broj ocena).

8.4 Rad sa matricama

Ilustrujmo upotrebu višedimenzionih nizova, tako što ćemo učitati kvadratnu matricu (dimenzije najviše 10) i proveriti da li je ona magični kvadrat. Matrica dimenzije $n \times n$ čini magični kvadrat ako su njeni brojevi od 1 do n^2 i jednaki su zbirovi elemenata svake vrste, svake kolone i obe dijagonale. Zbir svih brojeva od 1 do n^2 jednak je $n^2(n^2 + 1)/2$, pa zbir elemenata svake vrste, svake kolone i obe dijagonale treba da bude jednak $(n^2(n^2 + 1)/2)/n = n(n^2 + 1)/2$.

```
const int MAXN = 10;

// provera da li je kvadrat magican
// funkcija prima matricu, dimenziju kvadrata i njegov
 ↪  karakteristicni zbir
bool jeMagicanKvadrat(int kvadrat[MAXN][MAXN], int n, int
 ↪  zbir) {
   // proveravamo da li se svaki broj od 1 do n*n javlja
    ↪  tacno jednom
   // (vidjen[x] belezi da li smo broj x vec videli)
   bool vidjen[MAXN*MAXN + 1] = {false};
   for (int v = 0; v < n; v++)
      for (int k = 0; k < n; k++) {
         int x = kvadrat[v][k];
         // broj je van opsega ili se ponavlja, pa kvadrat
          ↪  nije magican
         if (x < 1 || x > n*n || vidjen[x])
            return false;
         vidjen[x] = true;
      }

   // proveravamo zbirove svih vrsta
   for (int vrsta = 0; vrsta < n; vrsta++) {
      int zbirVrste = 0;
      for (int kolona = 0; kolona < n; kolona++)
         zbirVrste += kvadrat[vrsta][kolona];
      // nasli smo vrstu koja nema trazeni zbir, pa kvadrat
       ↪  nije magican
      if (zbirVrste != zbir)
         return false;
   }
```

```
    // proveravamo zbirove svih kolona
    for (int kolona = 0; kolona < n; kolona++) {
       int zbirKolone = 0;
       for (int vrsta = 0; vrsta < n; vrsta++)
          zbirKolone += kvadrat[vrsta][kolona];
       // nasli smo kolonu koja nema trazeni zbir, pa kvadrat
        ↪ nije magican
       if (zbirKolone != zbir)
          return false;
    }

    // proveravamo zbir glavne dijagonale
    int zbirDijagonale = 0;
    for (int i = 0; i < n; i++)
       zbirDijagonale += kvadrat[i][i];
    // ako zbir glavne dijagonale nije odgovarajuci, kvadrat
     ↪ nije magican
    if (zbirDijagonale != zbir)
       return false;

    // proveravamo zbir sporedne dijagonale
    zbirDijagonale = 0;
    for (int i = 0; i < n; i++)
       zbirDijagonale += kvadrat[i][n-1-i];
    // ako zbir sporedne dijagonale nije odgovarajuci, kvadrat
     ↪ nije magican
    if (zbirDijagonale != zbir)
       return false;

    // zbirovi svih vrsta, kolona i obe dijagonale su ispravni
    return true;
}

int main() {
    int kvadrat[MAXN][MAXN];
    int n;
    cin >> n;
    for (int v = 0; v < n; v++)
        for (int k = 0; k < n; k++)
           cin >> kvadrat[v][k];
```

```
    // karakteristicni zbir
    int zbir = n*(n*n + 1) / 2;
    if (jeMagicanKvadrat(kvadrat, n, zbir))
       cout << "Kvadrat je magicni" << endl;
    else
       cout << "Kvadrat nije magicni" << endl;
}
```

Matrice (tj. dvodimenzioni nizovi) koriste se u mnogim oblastima računarstva: u obradi slika (za rasterski zapis slika), u računarskoj grafici (za opis transformacija), u rešavanju linearnih jednačina, itd. Kao što je rečeno ranije, matrice se obično predstavljaju kao statički dvodimenzioni nizovi ili kao vektori vektora. U nastavku su prikazane implementacije funkcija za sabiranje, množenje i štampanje matrica reprezentovanih kao vektori vektora.

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

vector<vector<int>> saberiMatrice(const vector<vector<int>>&
 ↪  A,
                                   const vector<vector<int>>&
                                    ↪  B)
{
   int vrste = A.size(), kolone = A[0].size();
   vector<vector<int>> C(vrste, vector<int>(kolone));
   for(int i = 0; i < vrste; i++)
      for(int j = 0; j < kolone; j++)
         C[i][j] = A[i][j] + B[i][j];
   return C;
}

vector<vector<int>> pomnoziMatrice(const
 ↪  vector<vector<int>>& A,
                                    const
                                     ↪  vector<vector<int>>&
                                     ↪  B)
{
   int Avrste = A.size(), Akolone = A[0].size();
   int Bvrste = B.size(), Bkolone = B[0].size();
```

```
    vector<vector<int>> C(Avrste, vector<int>(Bkolone, 0));
    if (Akolone != Bvrste) {
       cout << "Matrice ne mogu biti pomnozene" << endl;
    } else {
      for(int i = 0; i < Avrste; i++)
         for(int j = 0; j < Bkolone; j++) {
            for(int k = 0; k < Akolone; k++)
                C[i][j] += A[i][k] * B[k][j];
         }
    }
    return C;
}

void odstampajMatricu(const vector<vector<int>>& A)
{
    int vrste = A.size(), kolone = A[0].size();
    for(int i = 0; i < vrste; i++) {
       for(int j = 0; j < kolone; j++)
          cout << A[i][j] << " ";
       cout << endl;
    }
}

int main()
{
    vector<vector<int>> A = { { 1, 2, 3}, { 4, 5, 3} };
    vector<vector<int>> B = { { 2, 3, 4}, { 5, 2, 1} };
    vector<vector<int>> C = { { 1, -1}, { 0, 1}, { 1, 0} };

    cout << "Zbir matrica A i B:" << endl;
    odstampajMatricu(saberiMatrice(A,B));

    cout << "Proizvod matrica A i C:" << endl;
    odstampajMatricu(pomnoziMatrice(A,C));
    return 0;
}
```

8.5 Strukture podataka sa asocijativnim pristupom

U strukturama podataka sa sekvencijalnim pristupom, elementi su poređani u niz, te može da ih identifikuje redni broj u tom nizu. Taj redni broj, dakle, omogućava jednostavan pristup i brzo pronalaženje elementa strukture.
U strukturama sa asocijativnim pristupom, efikasan pristup elementu omogućava *ključ*. Ključevi određuju uređenje elemenata u strukturi i omogućavaju brzo pronalaženje elemenata.
Kao i bibliotečke strukture podataka sa sekvencijalnim pristupom, i strukture podataka sa asocijativnim pristupom su *šablonske* strukture (engl. templates), te mogu da čuvaju elemente proizvoljnog tipa `T`.

8.5.1 Skupovi

Često postoji potreba da održavamo skup elemenata (bez duplikata), u koji efikasno možemo da dodajemo elemente, iz koga efikasno možemo da izbacujemo elemente i za koji efikasno možemo da proveravamo da li je neka zadata vrednost element skupa. Savremeni programski jezici u svojim bibliotekama pružaju strukture podataka koje nude baš ove operacije.
U jeziku C++, *skup* je opisan klasom `set<T>`, gde je `T` tip elemenata skupa (za korišćenje ove klase potrebno je uključiti zaglavlje `<set>`). U slučaju skupa, ključ za element skupa je sam taj element.
Skupovi podržavaju sledeće osnovne operacije (za pregled svih operacija upućujemo čitaoca na dokumentaciju):

- `insert` – umeće novi element u skup (ako je element već u skupu, operacija nema efekta).
- `erase` – uklanja dati element iz skupa (ako element ne postoji u skupu, skup se ne menja).
- `find` – proverava da li skup sadrži dati element i vraća iterator na njega ako sadrži, a vrednost `end` inače. Tako se provera pripadnosti elementa `e` skupu `s` može izvršiti sa `if (s.find(e) != s.end()) ...`
- `size` – vraća broj elemenata skupa.

Moguća je i iteracija kroz elemente skupa korišćenjem petlje oblika `for (T element : skup)`, pri čemu se elementi kolekcije nabrajaju u sortiranom redosledu. Na primer, naredni program učitava brojeve i ispisuje ih u uređenom redosledu, bez duplikata.

```
int n;
cin >> n;
set<int> A;
for (int i = 0; i < n; i++) {
```

```
    int x;
    cin >> x;
    A.insert(x);
}
cout << "Broj razlicitih: " << A.size() << endl;
for (int x : A)
    cout << x << endl;
```

Uređeni skupovi (kolekcija `set`) podržavaju i metode:

- `lower_bound(x)` – pronalazi najmanji element skupa koji nije manji od date vrednosti `x` i vraća iterator koji ukazuje na njega (ili `end`, ako takav element ne postoji),
- `upper_bound(x)` – pronalazi najmanji element skupa koji je strogo veći od date vrednosti `x` i vraća iterator koji ukazuje na njega (ili `end`, ako takav element ne postoji).

U uređenim skupovima se lako mogu naći minimalni i maksimalni element (korišćenjem iteratora dobijenih metodama `begin` i `end`).
Ako uređen skup sadrži podatke tipa za koji postoji podrazumevani poredak (na primer, brojeve ili niske, koje se porede leksikografski), tada se koristi taj poredak. Da bi se definisao skup elemenata nekog tipa nad kojim nije definisan podrazumevani poredak, potrebno je posebno definisati (i zadati u okviru deklaracije) poredak koji želimo da se koristi. I za tip podataka za koji postoji podrazumevani poredak, može se deklarisati skup koji koristi neki drugi poredak. Na primer, skup u kom su elementi tipa `T` uređeni nerastuće (pri čemu na tipu podataka postoji podrazumevani neopadajući poredak) može se definisati deklaracijom `set<T, greater<T>>` (pri čemu je za upotrebu `greater` potrebno uključiti zaglavlje `<functional>`).

8.5.2 *Multiskupovi*

Skupovi, kao i u matematici, ne mogu da sadrže duplikate. Kada se element koji već postoji u skupu ubaci u taj skup metodom `insert`, skup se ne menja. Jedno uopštenje skupova daju *multiskupovi*, u kojima je dopušteno ponavljanje elemenata. Multiskupovi su podržani bibliotečkom strukturom `multiset<T>`, koja se koristi na potpuno isti način kao i `set<T>` (za njeno korišćenje je takođe dovoljno uključiti zaglavlje `<set>`).

8.5.3 *Mape*

Programski jezik C++ pruža podršku za *mape* (u drugim se jezicima one nazivaju i rečnici ili asocijativni nizovi) – kolekcije podataka u kojima se ključevima nekog tipa pridružuju vrednosti nekog (ne obavezno istog) tipa. Na primer, imenima meseci (podacima

tipa `string`) možemo dodeliti broj dana (podatke tipa `int`). Rečnici se predstavljaju objektima tipa `map<TipKljuca, TipVrednosti>`, definisanog u zaglavlju `<map>`. Na primer,

```
map<string, int> brojDana =
{
    {"januar", 31},
    {"februar", 28},
    {"mart", 31},
    ...
};
```

Primetimo da smo inicijalizaciju mape izvršili tako što smo naveli listu parova oblika `{kljuc, vrednost}`. Inicijalizaciju nije neophodno izvršiti odmah tokom kreiranja, već je vrednosti moguće dodavati (a i čitati) korišćenjem indeksnog pristupa (pomoću zagrada `[]`).

```
map<string, int> brojDana;
brojDana["januar"] = 31;
brojDana["februar"] = 28;
brojDana["mart"] = 31;
...
```

Mapu, dakle, možemo shvatiti i kao niz tj. vektor u kome indeksi nisu obavezno iz nekog celobrojnog intervala oblika $[0, n)$, već mogu biti proizvoljnog tipa.
Ako se korišćenjem operatora indeksnog pristupa `[...]` pokuša pristup ključu koji ne postoji, on se ubacuje u mapu (i pridružuje mu se podrazumevana vrednost tipa vrednosti mape). Metodom `at` se može pročitati vrednost pridružena ključu, bez ubacivanja nove vrednosti ako ključ ne postoji (ako ključ ne postoji, dolazi do izuzetka).
Pretragu ključa možemo ostvariti metodom `find`, koja vraća iterator na pronađeni element, ako element postoji, a iterator iza kraja mape (koji dobijamo metodom ili funkcijom `end`), inače. Na primer,

```
string mesec; cin >> mesec;
auto it = brojDana.find(mesec);
if (it != end(brojDana))
   cout << "Broj dana: " << it->second << endl;
else
   cout << "Mesec nije korektno unet" << endl;
```

Sve elemente rečnika moguće je ispisati korišćenjem petlje `for`. Na primer,

```
for (const pair<const string, int>& p : brojDana) {
   string mesec = p.first; int broj_dana = p.second;
   cout << mesec << ": " << broj_dana << endl;
}
```

ili kraće:

```
for (const auto& p : brojDana) {
  string mesec = p.first; int broj_dana = p.second;
  cout << mesec << ": " << broj_dana << endl;
}
```

U prethodnoj petlji promenljiva p je referenca koja se tokom petlje pomera tako da ukazuje na jedan po jedan uređen par elemenata mape (svaki element je par koji sadrži ključ i vrednost). Ključnom rečju `const` se naznačava da se tokom prolaska kroz elemente mape oni neće menjati. Primetimo da je ključ u paru konstantan (tip elementa mape je `pair<const string, int>`), jer se ključevi elemenata mape ne smeju menjati. Zato je zgodno koristiti `auto`, pa kompilator sam određuje tačan tip.
Još lepši oblik je kada se tokom iteracije odmah par razdvoji na ključ i vrednost.

```
for (const auto& [mesec, broj_dana] : brojDana)
    cout << mesec << ": " << broj_dana << endl;
```

Alternativno, možemo eksplicitno koristiti iteratore:

```
for (auto it = brojDana.begin(); it != brojDana.end(); it++)
     cout << it->first << ": " << it->second << endl;
```

Iteracija kod (sortirane, uređene) mape se uvek vrši u sortiranom redosledu ključeva. Ključevi mape mogu biti samo nekog tipa čije vrednosti se mogu porediti relacijskim operatorima. Niske, koje ćemo najčešće koristiti kao ključeve, zadovoljavaju taj uslov. Ako su ključevi mape neke korisnički definisane strukture ili objekti klasa, tada je potrebno da se u tim strukturama ili klasama definiše operator <, koji poredi dva objekta. Ilustrujmo upotrebu mapa na primeru programa koji učitava spisak mejl adresa i određuje onu koja se najčešće pojavljuje u spisku (ako ima više takvih, ispisuje bilo koju od njih). Osnovni zadatak je da se prebroji koliko puta se u spisku pojavljuje svaka od adresa, i njega možemo rešiti tako što svakom ključu mape (mejl adresi) pridružimo njen broj pojavljivanja.

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

int main() {
   // svakoj ucitanoj adresi dodeljujemo njen broj
    ↪ pojavljivanja
   map<string, int> brojPojavljivanja;
   // ukupan broj adresa koje ucitavamo
   int brojAdresa;
   cin >> brojAdresa;
   for (int i = 0; i < brojAdresa; i++) {
      string adresa;
      cin >> adresa;
      // povecavamo broj pojavljivanja upravo ucitane adrese
      brojPojavljivanja[adresa]++;
   }

   // maksimalni broj pojavljivanja neke adrese
   int maksPojavljivanja = 0;
   // adresa koja se pojavljuje najveci broj puta
   string maksAdresa;
   // obradjujemo sve adrese koje su u mapi
   for (const auto& [adresa, broj] : brojPojavljivanja)
       // azuriramo maksimum
       if (broj > maksPojavljivanja) {
           maksPojavljivanja = broj;
           maksAdresa = adresa;
       }
   // ispisujemo adresu koja se najvise pojavljuje
   cout << maksAdresa << endl;
}
```

Naglasimo da naredba `brojPojavljivanja[adresa]++` povećava broj pojavljivanja adrese u mapi, ako ona postoji, a ako ne postoji, onda je prvo ubacuje i pridružuje joj vrednost 0 i odmah zatim tu vrednost uvećava na 1.

Sličan zadatak bi bio da se odredi slovo koje se najčešće pojavljuje u reči. Brojanje pojavljivanja bismo mogli da ostvarimo na sledeći način.

```
string s;
cin >> s;
map<char, int> brojPojavljivanja;
for (char c : s)
   brojPojavljivanja[c]++;
```

Ako, na primer, znamo da se niska `s` sastoji samo od malih slova engleske abecede, umesto u mapi, brojeve pojavljivanja svakog slova možemo pamtiti u statički alociranom nizu koji ima 26 elemenata (broj pojavljivanja slova `a` na poziciji 0, slova `b` na poziciji 1, itd.). Poziciju datog slova možemo lako odrediti tako što od njegovog koda oduzmemo kôd karaktera `a` (obično se radi o kodovima u tabeli ASCII). U jeziku C++ (isto kao u jeziku C) karakteri su interno reprezentovani pomoću svojih kodova, pa se odgovarajuća pozicija u nizu može dobiti oduzimanjem karakterske konstante `'a'`. S druge strane, karakter se može dobiti od svoje redne pozicije dodavanjem `'a'` i konverzijom u tip `char`.

```
string s;
cin >> s;
// broj pojavljivanja svakog karaktera
int brojPojavljivanja[26] = {0};
for (char c : s)
   brojPojavljivanja[c - 'a']++;
// pozicija karaktera koji se najcesce pojavljuje
int maks = 0;
for (int i = 0; i < 26; i++)
   if (brojPojavljivanja[i] > brojPojavljivanja[maks])
      maks = i;
// na osnovu pozicije odredjujemo karakter koji se pojavljuje
 ↪  najcesce
cout << (char)('a' + maks) << endl;
```

Dakle, ako se u programu opisuje preslikavanje jednog skupa u drugi i ako je domen preslikavanja mali skup uzastopnih vrednosti (kao što su mala slova engleske abecede), umesto mape se može koristiti i niz. Takva rešenja su tipična za programski jezik C, u kome mape ne postoje, ali imaju smisla i u jeziku C++: pristup elementu niza je brži od pristupa elementu mape i niz zauzima manje memorije. Sa druge strane, mapa je opštija (ključevi ne moraju biti mali celi brojevi) i ne zahteva preračunavanje ključa u indeks, pa je rešenje sa mapom često jednostavnije i manje podložno greškama.

8.6 Specijalizovane strukture podataka

8.6.1 Stek

Stek (engl. stack) je kolekcija podataka sa pristupom po principu *LIFO* (engl. last-in-first-out) – element se može dodati samo na vrh steka i može se skinuti samo sa vrha steka. Kao stek ponaša se, na primer, štap na koji su naređani kompakt diskovi. Ako sa štapa može da se uklanja samo po jedan disk, da bi bio skinut disk koji je na dnu, potrebno je pre njega skinuti sve druge diskove.
U jeziku C++, stek se realizuje klasom `stack<T>`, gde `T` predstavlja tip elemenata na steku. Za njeno korišćenje potrebno je uključiti zaglavlje `<stack>`. Podržane su sledeće metode (sve su veoma efikasne):

- `push` – postavlja dati element na vrh steka
- `pop` – skida element sa vrha steka (pod pretpostavkom da stek nije prazan). Ova metoda je tipa `void` i ne vraća uklonjeni element. Poziv metode `pop` u trenutku kada je stek prazan dovodi do nedefinisanog ponašanja (obično do nasilnog prekida programa) i zadatak programera je da osigura da se to neće dešavati tokom izvršavanja programa.
- `top` – očitava element na vrhu steka (pod pretpostavkom da stek nije prazan)
- `empty` – proverava da li je stek prazan
- `size` – vraća broj elemenata na steku

Stek u jeziku C++ je zapravo samo *adapter*, omotač oko neke kolekcije podataka (obično reda sa dva kraja ili vektora) koji korisnika primorava da poštuje pravila pristupa steku i sprečava da napravi operaciju koja nad stekom nije dopuštena (poput pristupa nekom elementu ispod vrha).

8.6.1.1 Primer upotrebe steka: izrazi u postfiksnoj notaciji

Kao primer upotrebe steka razmotrimo izračunavanje vrednosti izraza zapisanih u *postfiksnoj* notaciji. U postfiksnoj notaciji, binarni operatori se ne zapisuju između operanada, nego iza njih. Na primer, izraz $3 \cdot ((1 + 2) \cdot 3 + 5)$ se zapisuje na sledeći način: $(3\ (((1\ 2\ +)\ 3\ \cdot)\ 5\ +)\ \cdot)$ Interesantno je da zagrade u postfiksnom zapisu uopšte ne moraju da se pišu i nema opasnosti od višesmislenog tumačenja zapisa. Dakle, navedeni izraz može se napisati i na sledeći način: $3\ 1\ 2\ +\ 3\ \cdot\ 5\ +\ \cdot$ (u ASCII sintaksi `3 1 2 + 3 * 5 + *`).
Vrednost navedenog izraza može se jednostavno izračunati čitanjem sleva nadesno i korišćenjem steka. Ako je pročitan broj, on se stavlja na stek. Inače (ako je pročitan znak operacije), onda se dva broja skidaju sa steka, na njih se primenjuje pročitana operacija i rezultat se stavlja na stek. Nakon čitanja čitavog izraza (ako je ispravno zapisan), na steku će biti samo jedan element, i to broj koji je vrednost izraza.
Sledeći program čita aritmetički izraz (zapisan u postfiksnom zapisu). Jedine dozvoljene operacije su `+` i `*`. Brojevi i znakovi operacija su razdvojeni razmacima. Program čita

jednu po jednu nisku, sve dok ih ima, tj. dok ne dođe do kraja ulaza. Pretpostavićemo da je svaka niska ili ispravno zapisan prirodni broj ili simbol operacije. Kada se naiđe na broj, on se postavlja na stek (nisku sastavljenu od cifara možemo konvertovati u broj korišćenjem bibliotečke funkcije `stoi`, opisane u poglavlju 9). Kada se naiđe na znak operacije, sa steka se čitaju dve vrednosti i zamenjuju se rezultatom primene pročitane operacije. Ukoliko je zadati izraz ispravno zapisan (što ćemo pretpostaviti), na kraju izvršavanja programa vrednost izraza je jedini element koji je ostao na steku i ona se ispisuje.

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

int main() {
   stack<int> operandi;
   // citamo jednu po jednu nisku do kraja ulaza
   string s;
   while (cin >> s) {
      if (s[0] == '+') {
         // operator +
         // dve vrednosti na vrhu steka menjamo njihovim
          ↪ zbirom
         int op1 = operandi.top(); operandi.pop();
         int op2 = operandi.top(); operandi.pop();
         operandi.push(op1 + op2);
      } else if (s[0] == '*') {
         // operator *
         // dve vrednosti na vrhu steka menjamo njihovim
          ↪ proizvodom
         int op1 = operandi.top(); operandi.pop();
         int op2 = operandi.top(); operandi.pop();
         operandi.push(op1 * op2);
      } else {
         // broj
         // postavljamo vrednost na stek
         operandi.push(stoi(s));
      }
   }
```

```
    // ako je izraz ispravan, njegova konacna vrednost se
     ↪  nalazi
    // na vrhu steka
    cout << operandi.top() << endl;
}
```

Na primer, ako se unesu sledeći podaci (koji se čitaju kao niske):

```
3 1 2 + 3 * 5 + *
```

nakon prekida unosa, program će ispisati vrednost 42.
Primetimo da se prilikom skidanja vrednosti sa steka ona mora prvo pročitati (metodom `top`), pa tek zatim skinuti sa steka (metodom `pop`).

8.6.2 *Red*

Red je kolekcija podataka sa pristupom po principu *FIFO* (engl. first-in-first-out) – element se uvek uzima sa početka, a dodaje na kraj reda.
U jeziku C++, red se realizuje klasom `queue<T>`, gde `T` predstavlja tip elemenata u redu. Za njeno korišćenje potrebno je uključiti zaglavlje `<queue>`. Podržane su sledeće metode:

- `push` – postavlja dati element na kraj reda
- `pop` – skida element sa početka reda (pod pretpostavkom da red nije prazan). Naglasimo da je ova metoda tipa `void` i da ne vraća uklonjeni element.
- `front` – očitava element na početku reda (pod pretpostavkom da red nije prazan)
- `empty` – proverava da li je red prazan
- `size` – vraća broj elemenata u redu

Red u jeziku C++ je zapravo samo adapter oko neke kolekcije podataka (obično reda sa dva kraja) koji korisnika primorava da poštuje pravila pristupa redu i sprečava da napravi operaciju koja nad redom nije dopuštena (poput pristupa nekom elementu koji nije na početku).

8.6.2.1 *Primer upotrebe reda: poslednjih k učitanih linija teksta*

Ilustrujmo upotrebu reda na primeru programa koji štampa poslednjih k učitanih linija teksta (pretpostavljamo da je $k \geq 1$). Ako je broj k dosta manji od ukupnog broja linija (što često može biti slučaj), tada bi rešenje koje bi učitalo sve linije u jedan vektor nepotrebno trošilo previše memorije. Umesto toga, možemo u strukturi podataka čuvati samo poslednjih k učitanih linija (ili manje, dok se još ne učita prvih k linija). Kada se učita nova linija, ona se dodaje na kraj reda. Ako je tada u redu $k + 1$ linija, uklanja se prva linija iz reda.

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

int main() {
   // broj linija koje treba ispisati
   int k;
   // sa ws osiguravamo da ce biti procitan i prelazak u novi
    ↪ red
   cin >> k >> ws;
   // red u kome cuvamo poslednjih k linija
   queue<string> poslednjiKLinija;
   // citamo sve linije do kraja ulaza
   string linija;
   while (getline(cin, linija)) {
       // osiguravamo da u redu nema nikad vise od k linija
       if (poslednjiKLinija.size() == k)
          poslednjiKLinija.pop();
       // ubacujemo procitanu liniju u red
       poslednjiKLinija.push(linija);
   }

   // ispisujemo rezultat
   while (!poslednjiKLinija.empty()) {
      cout << poslednjiKLinija.front() << endl;
      poslednjiKLinija.pop();
   }
}
```

8.6.3 *Red sa dva kraja*

Jedno uopštenje strukture red je *red sa dva kraja*, koji dopušta da se elementi i dodaju i uzimaju sa oba kraja reda (ta struktura podataka zapravo kombinuje i funkcionalnost steka i funkcionalnost reda).
U jeziku C++, red sa dva kraja raspoloživ je kao struktura `deque<T>`. Za njeno korišćenje potrebno je uključiti zaglavlje `<deque>`. Podržane su sledeće operacije (sve su veoma efikasne):

- `push_front` – dodavanje elementa na početak

- `push_back` – dodavanje elementa na kraj
- `front` – čitanje elementa sa početka (pod pretpostavkom da red nije prazan)
- `back` – čitanje elementa sa kraja (pod pretpostavkom da red nije prazan)
- `pop_front` – uklanjanje elementa sa početka (pod pretpostavkom da red nije prazan). Ova metoda je tipa `void` i ne vraća uklonjeni element.
- `pop_back` – uklanjanje elementa sa kraja (pod pretpostavkom da red nije prazan). Naglasimo da je ova metoda tipa `void` i da ne vraća uklonjeni element.
- `empty` – provera da li je red prazan
- `size` – broj elemenata u redu

Interesantno, zahvaljujući specifičnom načinu implementacije, ova struktura podataka podržava i operator indeksnog pristupa kojim se element na datoj poziciji može pročitati ili izmeniti veoma efikasno.

8.6.3.1 *Primer upotrebe reda sa dva kraja: istorija veb-pregledača*

Kao primer upotrebe strukture `deque` navešćemo program koji simulira rad istorije veb-pregledača. Pretpostavimo da se u njoj pamte adrese k prethodno posećenih veb-sajtova. Kada korisnik poseti novi veb-sajt, on se dodaje na kraj istorije. Ako u istoriji nema mesta za dodavanje novog sajta, prva dodata adresa (ona na početku reda) se briše. Ako korisnik pritisne dugme `back`, on se vraća na prethodno posećeni veb-sajt, koji se nalazi na kraju istorije. Pretpostavićemo da se sa standardnog ulaza prvo učitava broj k (pretpostavljamo da je $k \geq 1$), a zatim linije sve do kraja ulaza (tj. dok se program ne prekine). Poslednja adresa u redu je adresa sajta na kome se korisnik trenutno nalazi. Ako je sadržaj linije niska `back`, ta adresa se uklanja iz reda i ispisuje se adresa sajta na koji se korisnik vratio, tj. nova poslednja adresa u redu (ili `-`, ako je red prazan). U suprotnom, smatraćemo da je u pitanju nova adresa: ona se ispisuje (jer korisnik prelazi na taj sajt) i dodaje se na kraj reda.

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

int main() {
  // citamo duzinu istorije (sa ws osiguravamo da ce biti
   ↪ procitan
  // i prelazak u novi red)
  int k;
  cin >> k >> ws;
  // red sa dva kraja u kom cuvamo istoriju posecenih sajtova
  deque<string> istorija;
```

```
  // citamo liniju po liniju do kraja standardnog ulaza
  string linija;
  while (getline(cin, linija)) {
    if (linija == "back") {
      // treba se vratiti na prethodno posecen sajt
      // skidamo trenutni sajt iz istorije (ako postoji)
      if (!istorija.empty())
        istorija.pop_back();
      // prijavljujemo prethodni sajt (ako postoji)
      if (!istorija.empty()) {
        cout << istorija.back() << endl;
      } else {
        cout << "-" << endl;
      }
    } else {
      // ispisujemo adresu sajta na koji prelazimo
      cout << linija << endl;
      // osiguravamo da u istoriji ne moze nikada biti vise
      ↪  od k adresa
      if (istorija.size() == k)
        istorija.pop_front();
      // dodajemo trenutnu adresu na kraj istorije
      istorija.push_back(linija);
    }
  }
}
```

Ako bismo umesto reda sa dva kraja (`deque`) koristili vektor, program bi nastavio da funkcioniše, ali bi bio sporiji, zato što uklanjanje prvog elementa vektora podrazumeva da se svi ostali elementi pomeraju za jednu poziciju ulevo i veoma je neefikasna operacija.

8.6.4 *Red sa prioritetom*

Red sa prioritetom je vrsta reda u kome elementi imaju na neki način pridružen prioritet, dodaju se u red jedan po jedan, a uvek se iz reda uklanja onaj element koji ima najveći prioritet od svih elemenata u redu. Zbog načina implementacije, red sa prioritetom se nekada naziva i *hip* (engl. heap), o čemu će više reči biti u narednim tomovima knjige. U jeziku C++, red sa prioritetom se realizuje klasom `priority_queue<T>`, gde je `T` tip elemenata u redu. Za njeno korišćenje potrebno je uključiti zaglavlje `<queue>`. Red sa prioritetom podržava sledeće metode:

- `push` – dodaje dati element u red

- `pop` – uklanja element sa najvećim prioritetom iz reda (pod pretpostavkom da red nije prazan). Ova metoda je tipa `void` i ne vraća uklonjeni element.
- `top` – očitava element sa najvećim prioritetom (pod pretpostavkom da red nije prazan)
- `empty` – proverava da li je red prazan
- `size` – vraća broj elemenata u redu

Prilikom poređenja elemenata tipa `T` koristi se podrazumevani poredak. Veći elementi u tom poretku imaju veći prioritet. Na primer, ako se u red ubacuju elementi `1`, `3` i `2`, element `3` ima najveći prioritet, pa bi on bio vraćen primenom metode `top` i uklonjen primenom metode `pop`. Nekad nam je potreban red u kojem manji elementi imaju veći prioritet. Takav red može se najlakše deklarisati na sledeći način:

```
priority_queue<int, vector<int>, greater<int>> pq;
```

Navedenom deklaracijom zadato je:

- da će se u redu čuvati brojevi tipa `int`;
- da će se oni interno smeštati u kolekciju `vector<int>` (što je uslovljeno načinom interne reprezentacije podataka, koja zahteva neki oblik sekvencijalne kolekcije, tj. niza sa efikasnim indeksnim pristupom i mogućnošću proširivanja);
- da će se za poređenje koristiti struktura `greater<int>` definisana u zaglavlju `<functional>`, koja obrće podrazumevani poredak – vraća `true` ako i samo ako je prvi argument koji se poredi veći od drugog.

Drugi i treći argument deklaracije mogu biti izostavljeni i tada se podrazumeva korišćenje vektora i podrazumevani poredak nad elementima zadatog tipa. Nije moguće izostaviti drugi, a navesti treći argument.

8.6.4.1 *Primer upotrebe reda sa prioritetom: zbir najvećih k brojeva*

Ilustrujmo upotrebu reda sa prioritetom kroz program koji određuje zbir najvećih k brojeva učitanih sa ulaza. Najvećih k do sada viđenih elemenata niza možemo čuvati u strukturi podataka koja nam omogućava da pronađemo najmanji element u njoj i da ga eventualno zamenimo onim koji je trenutno učitan (ako je trenutno učitani element veći od njega). Pogodna struktura za to je red sa prioritetom. Na početku red popunjavamo sa k prvih učitanih elemenata, a zatim svaki naredni učitani element poredimo sa najmanjim u redu i, ako je veći od njega, najmanji izbacujemo, a učitani element ubacujemo.

```
#include <iostream>
#include <queue>
```

```cpp
#include <vector>
#include <functional>
using namespace std;

int main() {
   int n, k;
   cin >> n >> k;

   // red sa prioritetom koji cuva k najvecih elemenata
    ↪ koristi se
   // min-hip, koji omogucava brzo uklanjanje najmanjeg
    ↪ elementa
   priority_queue<int, vector<int>, greater<int>> pq;

   // ucitavamo prvih k elemenata i ubacujemo ih u red
   for (int i = 0; i < k; i++) {
      int x;
      cin >> x;
      pq.push(x);
   }

   // ucitavamo preostale elemente
   for (int i = k; i < n; i++) {
      int x;
      cin >> x;
      // ako je ucitani element veci od najmanjeg trenutno u
       ↪ redu
      // izbacujemo taj najmanji i menjamo ga ucitanim
      if (x > pq.top()) {
          pq.pop();
          pq.push(x);
      }
   }

   // izbacujemo elemente iz reda racunajuci njihov zbir i
    ↪ ispisujemo ga
   int zbir = 0;
   while (!pq.empty()) {
      zbir += pq.top();
      pq.pop();
```

```
    }
    cout << zbir << endl;
}
```

9. Pregled standardne biblioteke

U većini savremenih programskih jezika, kroz dodatnu biblioteku raspoložive su implementacije mnogih često korišćenih algoritama. Te biblioteke su deo standarda jezika (na primer, standard jezika C++ opisuje i njegovu standardnu biblioteku), ali njihove funkcije nisu ugrađene u sintaksu jezika poput naredbi `if` ili `for`, već su napisane u samom jeziku i da bi se koristile, potrebno je uključiti odgovarajuće zaglavlje. Mnoge od ovih funkcija (ne i sve) mogu se jednostavno implementirati, pa se programeri često odlučuju za samostalno programiranje algoritama umesto korišćenja bibliotečkih verzija (naročito ako su u pitanju programeri naviknuti na programiranje u imperativnim programskim jezicima kakvi su `C` ili `Pascal`). To se ne smatra greškom, ali korišćenje bibliotečkih funkcija smatra se boljom navikom i boljom praksom iz nekoliko razloga. Kôd je kraći i lakše se razume, a u slučaju dobrog poznavanja biblioteke – programiranje je jednostavnije i brže. Dodatno, implementacije bibliotečkih funkcija sa sigurnošću se mogu smatrati ispravnim, a često su i efikasnije nego neka samostalno napisana verzija.

Neke funkcije iz standardnih biblioteka često se koriste i gotovo je nužno poznavati ih (na primer, funkcije za sortiranje). Neke se koriste ređe i u praksi je dovoljno znati da postoje i kako pronaći njihova svojstva u dokumentaciji jezika. Iako u standardnim bibliotekama postoji mnoštvo funkcija, njihovo memorisanje (sa više ili manje detalja) nije previše teško. Naime, većina je napisana u istom duhu i deklaracije su intuitivne, kao i sama imena funkcija, na primer, `sort` za sortiranje, `copy` za kopiranje i `reverse` za obrtanje niza.

Sve navedeno važi i za jezik `C++`. Veliki broj osnovnih algoritama implementiran je u vidu funkcija u okviru standardne biblioteke i one se mogu koristiti ako se uključi zaglavlje `<algorithm>`. Te funkcije su specifične za `C++`, ali slične postoje i za većinu drugih savremenih jezika, kao što su `Python`, `JavaScript`, `Haskell`. Zbog toga poznavanje standardne biblioteke jezika `C++` prevazilazi okvire ovog jezika i može se smatrati opštim programerskim znanjem. Kada su u pitanju bogate standardne biblioteke, među jezicima koji se široko koriste, jezik `C` je jedan od izuzetaka. Naime, ovaj jezik je u skoro svakom segmentu veoma sveden, pa tako i u svojoj standardnoj biblioteci. Ona, naravno,

postoji, ali sadrži samo mali broj implementiranih algoritama.
Korišćenje funkcija iz standardne biblioteke predstavlja dodatni i nešto drugačiji sloj u odnosu na znanje koje je predočeno u prethodnim delovima. U nastavku ćemo prikazati samo neke najznačajnije funkcije iz standardne biblioteke jezika C++, uz podsećanje da je dobro poznavati i veći njen deo. Veoma je poželjno i da programer za većinu funkcija razume kako one rade i da je u stanju da ih i sam implementira.

9.1 Korišćenje bibliotečke implementacije algoritama

Jedan od velikih izazova za početnike je pamćenje naziva i parametara velikog broja funkcija iz raspoloživih biblioteka. Savremeno programiranje podrazumeva korišćenje jezika sa bogatim bibliotekama, ali uz obavezni pristup dokumentaciji u kojoj su sve bibliotečke funkcije opisane i ilustrovane primerima upotrebe. Na primer, internet pretraga bilo koje bibliotečke funkcije jezika C++ će nas uputiti (između ostalog) na sajt `cppreference.com`, koji sadrži veoma detaljnu i dobro organizovanu dokumentaciju biblioteke jezika C++. Forumi namenjeni programerima, poput foruma `stackoverflow.com`, omogućavaju postavljanje pitanja na koje dobrovoljno odgovaraju iskusniji programeri. Na tim forumima uvek se može potražiti rešenje nekog problema, tj. opis korišćenja neke bibliotečke funkcije ili kombinacije funkcija (veoma je verovatno da je pitanje koje nas trenutno zanima neko već ranije postavio). Naravno, jako loše i veoma opasno je samo iskopirati ponuđeno rešenje, bez njegovog potpunog razumevanja. Pored dokumentacije i foruma, korišćenje bibliotečkih funkcija, a i programiranje uopšte, olakšavaju i savremena razvojna okruženja koja nude automatsko dopunjavanje naziva funkcija, integrisanu dokumentaciju, pa čak i integrisana rešenja zasnovana na veštačkoj inteligenciji koja pružaju predloge i opise koda. Sve su ovo važne olakšice, ali se dobar programer postaje jedino pažljivim proučavanjem biblioteke: najčešće korišćene funkcije (poput onih za sortiranje i pretragu) treba potpuno razumeti, a za ostale je dovoljno znati da postoje i umeti ih pronaći u dokumentaciji.

9.2 Bibliotečke funkcije za sekvencijalne kolekcije

9.2.1 Sortiranje

Sortiranje je jedan od fundamentalnih zadataka u računarstvu. Sortiranje podrazumeva uređivanje niza u odnosu na neku relaciju poretka (na primer, uređivanje niza brojeva po veličini — rastuće, neopadajuće, opadajuće ili nerastuće, uređivanje niza niski leksikografski ili po dužini, uređivanje niza struktura na osnovu vrednosti nekog polja i slično). Mnogi problemi nad nizovima mogu se jednostavnije i efikasnije rešiti u slučaju da je niz sortiran (na primer, sortirani nizovi se mogu efikasno pretraživati).
Većina programskih jezika, uključujući i programski jezik C++, u svojim bibliotekama ima funkcije za sortiranje nizova. U realnom programskom kodu uvek je preporuka vr-

šiti sortiranje korišćenjem tih funkcija, jer su one efikasno implementirane i detaljno testirane. S druge strane, izučavanje algoritama sortiranja može pomoći u savladavanju nekih važnih osnovnih algoritamskih tehnika i stoga je nezaobilazno u učenju programiranja, te su iz tog razloga i prikazani neki algoritmi sortiranja u poglavlju 6.6.1.
Kao i većina bibliotečkih funkcija za obradu sekvencijalnih kolekcija, funkcija sortiranja kao argumente prima dva iteratora ili dva pokazivača koja ograničavaju poluotvoreni segment niza koji se sortira. Ako su prosleđeni iteratori `from` i `to`, sortiraju se elementi iz intervala `[from, to)`. Ako želimo da sortiramo ceo vektor ili niz (što je najčešće slučaj), onda prosleđujemo iteratore `begin` i `end`, koji ukazuju na početak i na jednu poziciju iza kraja podataka. Na primer, sortiranje vektora možemo ostvariti na sledeći način.

```
vector<int> a = {3, 8, 1, 4, 9, 2, 6, 5, 7};
// sortiramo niz
sort(a.begin(), a.end());
// ispisujemo sortirani niz: 1, 2, 3, 4, 5, 6, 7, 8, 9
for (int x : a)
   cout << x << endl;
```

Umesto metoda `begin` i `end` možemo upotrebiti i funkcije, tj. koristiti poziv `sort(begin(a), end(a))`. Ovo je malo fleksibilnije, jer ispravno radi i za statički alocirane nizove.

```
int a[] = {3, 8, 1, 4, 9, 2, 6, 5, 7};
// sortiramo niz
sort(begin(a), end(a));
// ispisujemo sortirani niz: 1, 2, 3, 4, 5, 6, 7, 8, 9
for (int x : a)
   cout << x << endl;
```

I niske (tipa `string`) se smatraju sekvencijalnim kolekcijama (koje sadrže karaktere) i mogu se obrađivati na isti način kao i nizovi i vektori. Na primer, karakteri u niski se mogu sortirati na sledeći način (koristi se poredak određen ASCII kodovima karaktera).

```
string s = "Zdravo svima!";
sort(begin(s), end(s));
cout << s << endl; // ispisuje  !Zaadimorsvv
```

Kao što je rečeno, umesto dva iteratora, funkciji za sortiranje moguće je proslediti dva pokazivača. Na primer, niz `a` dužine `n` može se sortirati i pozivom `sort(a, a+n)`.

Može se sortirati i samo deo niza — tako što se prosleđuju iteratori koji ograničavaju željeni deo niza (a koji nisu `begin` i `end`).
Prilikom sortiranja se koristi podrazumevani poredak elemenata sekvencijalne kolekcije koja se sortira. Ako se sortiraju niske, to je leksikografski poredak (tj. leksikografsko proširenje poretka `<` nad pojedinačnim karakterima). U narednom primeru, nakon sortiranja dobija se niz `ananas`, `banana`, `jabuka`, `sljiva`, `visnja`:

```
vector<string> a = {"banana", "ananas", "jabuka", "visnja",
 ↪  "sljiva"};
sort(begin(a), end(a));
```

Redosled sortiranja može se promeniti zadavanjem funkcije poređenja (ona se navodi kao treći argument funkcije `sort`). U pitanju je funkcija koja prima dva elementa, poredi ih i vraća `true` ako prvi element treba da prethodi drugom u sortiranom nizu (a `false` inače). Na primer, naredni program sortira niske po dužini.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;

bool poredi_po_duzini(const string& a, const string& b) {
    return a.length() < b.length();
}

int main() {
   vector<string> a = {"audi", "folksvagen", "kia", "reno",
 ↪  "honda"};
   sort(begin(a), end(a), poredi_po_duzini);
   for (const string& s : a)
      cout << s << endl;
   return 0;
}
```

Prethodni program može dati sledeći izlaz (međusobni redosled niski iste dužine, u ovom primeru `audi` i `reno`, nije garantovan, o čemu će biti više reči u nastavku).

```
kia
audi
```

```
reno
honda
folksvagen
```

Umesto imenovane, moguće je upotrebiti i anonimnu funkciju poređenja.

```
sort(begin(a), end(a),
  [](const string& a, const string& b) {
    return a.length() < b.length();
  });
```

ili

```
auto poredi = [](const string& a, const string& b) {
                    return a.length() < b.length();
                };
sort(begin(a), end(a), poredi);
```

Prethodna funkcija poređenja ne nameće neki konkretan redosled niski iste dužine. Ako želimo garanciju da će niske iste dužine biti sortirane, na primer leksikografski, možemo funkciju poređenja definisati na sledeći način.

```
bool poredi(const string& a, const string& b) {
   if (a.length() == b.length())
      return a < b;
   return a.length() < b.length();
}
```

Česta potreba je da se niz sortira nerastuće i tada se može koristiti funkcijski objekat `greater<T>()` (tip `greater<T>` definisan je u zaglavlju `<functional>`), koji poredi elemente tipa `T` i vraća `true` ako je prvi element veći od drugog. Na primer,

```
vector<int> a = {3, 9, 1, 8, 4, 2, 6, 5, 7};
sort(begin(a), end(a), greater<int>());
// ispisuje 9 8 7 6 5 4 3 2 1
for (int x : a)
   cout << x << endl;
```

Prirodno je ponekad zahtevati da se elementi sortiraju po nekom kriterijumu, ali da se poredak elemenata koji su jednaki po tom kriterijumu ne promeni u odnosu na originalno stanje niza. Za sortiranje se kaže da je *stabilno* ako zadovoljava ovaj uslov. Funkcija `sort` ne vrši stabilno sortiranje, ali funkcija `stable_sort` vrši. Na primer, naredni program vrši sortiranje niski po dužini, pri čemu garantuje da se međusobni redosled niski iste dužine neće promeniti u odnosu na originalno stanje niza.

```
vector<string> niske =
  {"abc", "c", "ab", "ba", "cba", "a", "bca", "bc", "b"};
auto poredi_duzinu = [](const string& a, const string& b) {
   return a.length() < b.length();
};
stable_sort(begin(niske), end(niske), poredi_duzinu);
```

Rezultat ovog sortiranja će biti `c`, `a`, `b`, `ab`, `ba`, `bc`, `abc`, `cba`, `bca`.

9.2.2 *Linearna pretraga*

Pretraživanje sekvencijalne kolekcije podrazumeva proveru da li niz sadrži datu vrednost. Ako kolekcija nije sortirana, pretraživanje se može izvršiti funkcijom `find`. Ova funkcija vraća iterator koji ukazuje na prvo pojavljivanje tražene vrednosti, a ako kolekcija ne sadrži traženi element, onda funkcija vraća iterator iza poslednjeg elementa kolekcije (koji se dobija funkcijom `end`). Pozicija elementa u kolekciji se može odrediti izračunavanjem rastojanja između iteratora na početak kolekcije i iteratora koji ukazuje na pronađeni element.

```
vector<int> a = {3, 8, 4, 0, 1, 6, 2, 0, 5};
// trazimo vrednost 6 u nizu
auto it = find(begin(a), end(a), 6);
if (it != end(a))
   cout << "Element je pronadjen na poziciji: "
        << distance(begin(a), it) << endl;
else
   cout << "Element nije nadjen" << endl;
```

Poslednju poziciju traženog elementa možemo pronaći ako kolekciju pretražujemo unazad, korišćenjem iteratora `rbegin` i `rend` (koji kolekciju obilaze od poslednjeg ka prvom elementu):

```
auto it = find(rbegin(a), rend(a), 0);
if (it != rend(a))
   cout << "Poslednja nula je na poziciji: "
        << distance(it, rend(a)) - 1 << endl;
```

Ako element nije pronađen, vraća se `rend(a)`. Pošto se `rend(a)` nalazi neposredno ispred prvog elementa kolekcije, rastojanje od pronađenog elementa do `rend(a)` jednako je njegovoj poziciji uvećanoj za 1, pa se od tog rastojanja oduzima 1.
Broj pojavljivanja elemenata se može dobiti funkcijom `count`.

```
vector<int> a = {3, 8, 4, 0, 1, 6, 2, 0, 5};
int broj_nula = count(begin(a), end(a), 0);
cout << "Broj nula u nizu je: " << broj_nula << endl;
```

Umesto neke konkretne vrednosti možemo tražiti i prvi element koji zadovoljava neko zadato svojstvo. Funkcija `find_if`, pored iteratora koji ograničavaju elemente koji se pretražuju, prima i funkciju koja proverava da li trenutni element zadovoljava željeni uslov. Na primer, pozicija i vrednost prvog negativnog elementa u nizu se može naći na sledeći način.

```
bool negativan(int x) {
   return x < 0;
}

int main() {
    int a[] = {3, 8, 4, -1, -2, 7, -3, 5};
    auto it = find_if(begin(a), end(a), negativan);
    if (it != end(a))
        cout << distance(begin(a), it) << " " << *it << endl;
    else
        cout << "Nema negativnih elemenata" << endl;
}
```

Ako je funkcija koja proverava uslov jednostavna, nekad je pogodnije upotrebiti *anonimnu* funkciju:

```
...
auto it = find_if(begin(a), end(a), [] (int x) { return x <
 ↪  0; });
```

Funkcija `find_if_not` pronalazi prvi element koji ne zadovoljava dati uslov.
Funkcija `count_if` izračunava broj elemenata koji zadovoljavaju dati uslov, što je ilustrovano narednim kodom.

```
auto paran = [] (int x) { return x % 2 == 0; };
int broj_parnih = count_if(begin(a), end(a), paran);
cout << broj_parnih << endl;
```

Funkcija `all_of` proverava da li svi elementi zadovoljavaju dati uslov, funkcija `any_of` proverava da li postoji element koji zadovoljava dati uslov, dok funkcija `none_of` proverava da li nijedan element ne zadovoljava dati uslov.

```
auto negativan = [] (int x) { return x < 0; };
if (all_of(begin(a), end(a), negativan))
   cout << "Svi su negativni" << endl;
if (any_of(begin(a), end(a), negativan))
   cout << "Postoji negativan" << endl;
if (none_of(begin(a), end(a), negativan))
   cout << "Nijedan nije negativan" << endl;
```

9.2.3 *Binarna pretraga*

Ako je niz sortiran, onda je neuporedivo efikasnije nego linearnu upotrebiti binarnu pretragu, koja nam je na raspolaganju kroz funkciju `binary_search`. Ona ne vraća iterator na pronađeni element, već samo podatak tipa `bool` koji govori da li element postoji u nizu.

```
vector<int> a = {3, 8, 9, 12, 13, 18, 19, 27};
int x;
cin >> x;
if (binary_search(begin(a), end(a), x))
   cout << "Element " << x << " postoji u nizu" << endl;
else
   cout << "Element " << x << " ne postoji u nizu" << endl;
```

Funkcija `lower_bound` vrši binarnu pretragu sortiranog niza i vraća iterator koji ukazuje na prvu poziciju na kojoj se nalazi element koji je jednak ili veći od date vrednosti. Ako takav element ne postoji (tj. ako su svi elementi manji od date vrednosti), onda funkcija vraća iterator na poziciju iza kraja kolekcije, tj. iterator koji se dobija funkcijom `end`. Funkcija `upper_bound` vraća iterator koji ukazuje na prvu poziciju na kojoj se nalazi element koji je strogo veći od date vrednosti. Na primer,

```
vector<int> a = {1, 2, 2, 4, 4, 4, 7, 7, 9, 11, 11};
int x = 4;
auto l = lower_bound(begin(a), end(a), x);
auto d = upper_bound(begin(a), end(a), x);
cout << "Element " << x << " se javlja "
     << distance(l, d) << " puta" << endl;
```

U navedenom kodu, iterator `l` ukazuje na prvo pojavljivanje elementa 4 u nizu (jer se pronalazi prvi element koji je veći ili jednak 4), a iterator `d` na prvo pojavljivanje elementa 7 (jer se traži prvi element koji je strogo veći od 4). Razlika između njih je 3, što ukazuje na to da se element 4 javlja tri puta u nizu.

9.2.4 *Kopiranje, preslikavanje, filtriranje*

Za kopiranje nekog sadržaja iz jednog niza ili vektora u drugi (ili sa jednog mesta u istom vektoru na drugo mesto) može se koristiti funkcija `copy`. Na primer,

```
int a[] = {3, 8, 4, 2, 6, 9, 11, 17};
int b[8];
// kopiramo elemente niza a u niz b
copy(begin(a), end(a), begin(b));
```

Ili:

```
vector<int> a = {3, 8, 4, 2, 6, 9, 11, 17};
vector<int> b(8);
copy(begin(a), end(a), begin(b));
```

Naravno, kada se kopira celokupan sadržaj vektora, jednostavnije je upotrebiti dodelu nad vektorima (`b = a;`). Potreba za kopiranjem javlja se kada se kopira samo deo niza ili kada se rezultat formira od elemenata više nizova.
Funkcija `copy` podrazumeva da u nizu ili u vektoru u koji smeštamo rezultat ima dovoljno prostora. U suprotnom, dolazi do nedefinisanog ponašanja.
Moguće je da vektor u koji se kopira još uvek nema alociranu memoriju i da mu se prilikom kopiranja elementi dodaju na kraj (uz proširivanje niza kada je to potrebno). Tada se koristi poseban iterator `back_inserter`.

```
vector<int> a = {3, 8, 4, 2, 6, 9, 11, 17};
vector<int> b;
```

```
copy(begin(a), end(a), back_inserter(b));
```

Funkcija `copy_n` prima iterator koji ukazuje na početak dela niza koji se kopira i broj elemenata koji se kopiraju (umesto iteratora). Na primer, prvih `k` elemenata niza `a` se može iskopirati u niz `b` pozivom funkcije `copy_n(begin(a), k, begin(b))`.
Za filtriranje se može koristiti funkcija `copy_if`, koja kopira samo elemente koji zadovoljavaju dati uslov.

```
vector<int> a = {3, 8, 4, 2, 6, 9, 11, 17};
vector<int> parni;
copy_if(begin(a), end(a), back_inserter(parni),
           [] (int x) { return x % 2 == 0; });
```

Preslikavanje se vrši funkcijom `transform`. Na primer, naredni kôd uvećava svaki element niza `a` za 1 i smešta rezultat u novi niz `b`.

```
int a[] = {3, 8, 4, 2, 6, 7, 5, 9};
int b[8];
auto uvecaj_za_1 = [] (int x) { return x + 1; };
transform(begin(a), end(a), begin(b), uvecaj_za_1);
```

Naredni kôd pretvara sva mala slova iz niske u velika slova.

```
string s = "dobar dan";
transform(begin(s), end(s), begin(s), ::toupper);
```

Prva dva iteratora određuju deo niske koja se transformiše (u pitanju je cela niska), treći iterator određuje mesto na koje će se smeštati rezultat (ovde je u pitanju ponovo niska `s`), dok je četvrti parametar funkcija koja se primenjuje na svaki element (koristimo bibliotečku funkciju `toupper` iz zaglavlja `<cctype>`, nasleđenu iz jezika C, koja je opisana u poglavlju 9.3).
Funkcija `fill` popunjava dati raspon kolekcije datom vrednošću. Na primer, poziv `fill(begin(a), end(a), -1)` popunjava ceo niz vrednostima -1.
Funkcija `replace` menja jednu vrednost drugom. Na primer, poziv funkcije `replace(begin(a), end(a), -1, 0)` menja u nizu `a` sve vrednosti -1 vrednostima 0. Funkcija `replace_if` menja sve vrednosti koje zadovoljavaju dati uslov datom vrednošću. Na primer, naredni kôd menja sve negativne vrednosti u nizu vrednošću 0.

```
int a[] = {1, -2, 3, -4, 5, -6};
auto negativan = [] (int x) { return x < 0; };
replace_if(begin(a), end(a), negativan, 0);
```

9.2.5 *Brisanje elemenata*

Nekada je potrebno izbrisati iz kolekcije određenu vrednost ili vrednosti. Ova operacija je malo neobična, jer u kolekciji ne mogu da ostanu „rupe“ na mestima izbrisanih elemenata. Stoga su funkcije koje uklanjaju elemente definisane tako da pomeraju preostale elemente ka početku kolekcije. Funkcija `remove` briše sva pojavljivanja date vrednosti, a funkcija `remove_if` briše sve elemente koji zadovoljavaju dati uslov. Obe funkcije vraćaju iterator (ako su kao argumenti zadati iteratori, a pokazivač ako su kao argumenti zadati pokazivači na elemente kolekcije) na prvu poziciju iza novog sadržaja kolekcije (novi kraj kolekcije). Ako je u pitanju vektor, običaj je da se nakon pomeranja elemenata na početak metodom `erase` izvrši skraćivanje vektora (njoj se može proslediti iterator koji je povratna vrednost funkcije `remove` ili `remove_if`).

```
vector<int> a = {1, 0, 2, 0, 3, 0, 4};
a.erase(
   remove(begin(a), end(a), 0),
   end(a)
);
// sadrzaj vektora je {1, 2, 3, 4}

auto paran = [] (int x) { return x % 2 == 0; };
a.erase(
   remove_if(begin(a), end(a), paran),
   end(a)
);
// sadrzaj vektora je {1, 3}
```

Naglasimo da je brisanje neefikasna operacija i da je u slučaju čestog brisanja umesto nizova i vektora bolje koristiti neke druge kolekcije (liste, skupove).

9.3 *Rad sa karakterima*

U jeziku C++, funkcije za rad sa pojedinačnim karakterima su preuzete iz programskog jezika C i za njihovo korišćenje se uključuje zaglavlje `<cctype>`. Navedimo osnovne funkcije ovog zaglavlja.

```
int isalpha(int c); int isdigit(int c);
int isalnum(int c); int isspace(int c);
int isupper(int c); int islower(int c);
int toupper(int c); int tolower(int c);
```

Ove funkcije služe za ispitivanje i konvertovanje karaktera. Sve ove funkcije imaju argument tipa `int` i vraćaju vrednost tipa `int`. Vrednost koju vraćaju funkcije za ispitivanje (one čije ime počinje sa `is`) tumačimo kao istinitosnu vrednost – *tačno*, ako je različita od nule, i *netačno*, ako je jednaka nuli. Funkcije `toupper` i `tolower` vraćaju kôd (numeričku vrednost) konvertovanog karaktera.

- Funkcija `isalpha(c)` vraća vrednost različitu od nule ako je `c` slovo, nulu inače;
- Funkcija `isupper(c)` vraća vrednost različitu od nule ako je `c` veliko slovo, nulu inače;
- Funkcija `islower(c)` vraća vrednost različitu od nule ako je `c` malo slovo, nulu inače;
- Funkcija `isdigit(c)` vraća vrednost različitu od nule ako je `c` cifra, nulu inače;
- Funkcija `isalnum(c)` vraća vrednost različitu od nule ako je `c` slovo ili cifra, nulu inače;
- Funkcija `isspace(c)` vraća vrednost različitu od nule ako je `c` belina (razmak, tabulator, novi red, itd.), nulu inače;
- Funkcija `toupper(c)` vraća karakter `c` konvertovan u veliko slovo ili, ukoliko je to nemoguće — sâm karakter `c`;
- Funkcija `tolower(c)` vraća karakter `c` konvertovan u malo slovo ili, ukoliko je to nemoguće — sâm karakter `c`.

Važno je naglasiti da ove funkcije očekuju argument koji je ili vrednost tipa `unsigned char` ili vrednost `EOF`. Pošto tip `char` na mnogim sistemima ima i negativne vrednosti (na primer, bajtovi kojima se u kodiranju UTF-8 zapisuju slova poput `č`), poziv ovih funkcija za takav karakter dovodi do nedefinisanog ponašanja. Zato je bezbedno karakter pre poziva konvertovati u tip `unsigned char`, na primer, `isalpha((unsigned char)c)`.

9.4 Rad sa niskama

Niske su sekvencijalne kolekcije karaktera i sve funkcije koje rade sa sekvencijalnim kolekcijama rade i sa niskama. Uz to, klasa `string` ima nekoliko specifičnih, korisnih operatora i metoda.

- Niske se obično inicijalizuju konstantnim niskama (koje su navedene između dvostrukih navodnika) ili se učitavaju. Moguće je konstruisati nisku određene dužine

koja je popunjena datim karakterom. Na primer, `string(80, '-')` gradi nisku koja ima 80 crtica.

- Dve niske se mogu nadovezati operatorom `+`. Ovaj operator je moguće primeniti i na nisku tipa `string` i pojedinačni karakter i rezultat je nova niska.

```
string s = "zdravo";
string t = s + '!';
cout << t << endl; // ispisuje zdravo!
string r = '!' + s;
cout << r << endl; // ispisuje !zdravo
```

Međutim, često, umesto da gradimo novu nisku, želimo da postojećoj niski dodamo neki karakter ili nisku na kraj (što je efikasnije od izgradnje nove niske ili dodavanja teksta na početak niske). Za to koristimo operator `+=`.

```
string s = "zdravo";
s += '!';
cout << s << endl; // ispisuje: zdravo!
s += "svima";
cout << s << endl; // ispisuje: zdravo!svima
```

- Metoda `substr` izdvaja podnisku date niske. Prvi argument je obavezan i označava poziciju na kojoj počinje podniska. Ako se ne navede drugi argument, tada se podniska izdvaja do kraja zadate niske. Kao drugi argument, može se navesti i željeni broj karaktera podniske. Ako je niska prekratka, izdvajaju se karakteri do kraja niske.

```
string s = "zdravo svima!";
cout << s.substr(7) << endl;    // ispisuje: svima!
cout << s.substr(0, 6) << endl; // ispisuje: zdravo
```

- Metoda `find` pronalazi poziciju prvog pojavljivanja podniske unutar niske. Ako podniska ne postoji, vraća se specijalna vrednost `string::npos`. Ne treba mešati ovu metodu sa funkcijom `find`, koja radi za sve sekvencijalne kolekcije i koja se može upotrebiti za pronalaženje pojedinačnog karaktera.

```
string s = "zdravo_svima!";
cout << s.find("avo") << endl; // ispisuje 3
auto it = find(begin(s), end(s), 'a');
cout << distance(begin(s), it) << endl; // ispisuje 3
```

- Metoda `replace` menja postojeću nisku tako što deo te niske (zadat početnom pozicijom i brojem karaktera) zameni datom niskom. Nova niska se ne gradi, već se menja sama niska nad kojom je metoda pozvana. Na primer,

```
string s = "zdravo svima!";
s.replace(0, 6, "pozdrav");
cout << s << endl; // ispisuje: pozdrav svima!
```

- Niske se mogu porediti relacijskim operatorima <, <=, >, >= (koji odgovaraju leksikografskim proširenjima relacija <, <=, >, >= nad karakterima). Metoda `compare` takođe vrši leksikografsko poređenje i vraća pozitivnu vrednost ako je niska na kojoj je pozvana leksikografski veća od date niske, negativnu vrednost ako je manja, a nulu ako su niske jednake.

- Funkcije `stoi`, `stol`, `stoll`, `stof`, `stod` i `stold` služe za konverziju niske u odgovarajući brojevni tip (redom `int`, `long`, `long long`, `float`, `double` i `long double`). Na primer, izraz `stoi("123")` je tipa `int` i ima vrednost `123`.

- Funkcija `to_string` prevodi datu brojevnu vrednost u nisku (tipa `string`). Funkcija ispravno radi za različite brojevne tipove. Na primer, izraz `to_string(123)` je tipa `string` i ima vrednost `"123"`. Izraz `to_string(123.45)` vraća vrednost `"123.45"` (od standarda C++26; u ranijim verzijama vrednost je `"123.450000"`).

9.5 *Datoteke/tokovi*

U dosadašnjim programima sve podatke smo učitavali sa standardnog ulaza (korišćenjem `cin`), što obično podrazumeva učitavanje sa tastature, i ispisivali smo ih na standardni izlaz (korišćenjem `cout`), što obično podrazumeva ispis na ekran. Međutim, u mnogim programima potrebno je učitati podatke koji su trajno sačuvani ili rezultate rada programa sačuvati na trajni način. Podaci se na disku smeštaju u *datoteke*. Datoteke mogu sadržati tekst (zapisan pomoću niza čitljivih karaktera grupisanih u redove), ali i druge tipove podataka (na primer, slike, zvuk, video-zapise koji su zapisani na način

koji nije lako čitljiv čoveku). Ove prve zovemo *tekstualne* datoteke, a ove druge *binarne* datoteke. Jezik C++ podržava rad sa oba tipa datoteka.
Ulazno-izlazne operacije su generalno veoma spore i, ako se intenzivno koriste u nekom programu, mogu bitno uticati na njegovu efikasnost. Postoje načini da se rad ulazno-izlaznih operacija ubrza u nekoj meri (ali u ovoj knjizi se nećemo baviti njima).
U nastavku ćemo se fokusirati na rad sa tekstualnim datotekama. Jednostavnosti radi, podrazumevaćemo da je sav tekst zapisan isključivo pomoću karaktera ASCII tabele (malih i velikih slova engleske abecede, cifara i specijalnih znakova).
Učitavanje i ispis podataka u jeziku C++ ostvaruje se korišćenjem *tokova* (engl. stream). Ulazni tokovi služe za učitavanje podataka u program, a izlazni tokovi za ispis podataka iz programa. Ulazni tok `cin` predstavlja standardni ulaz programa, izlazni tok `cout` predstavlja standardni izlaz programa, a izlazni tok `cerr` predstavlja standardni izlaz za greške. Korišćenjem usluga operativnog sistema prilikom pokretanja programa moguće je izvršiti redirekciju ovih tokova. Na primer, ako se program pokrene na sledeći način

```
./program < ulaz.txt > izlaz.txt 2> greske.txt
```

tada će svako čitanje sa standardnog ulaza (pomoću `cin`) biti zapravo čitanje iz datoteke `ulaz.txt`, svaki ispis na standardni izlaz (pomoću `cout`) biće zapravo upis u datoteku `izlaz.txt`, dok će svaki ispis poruke o grešci (pomoću `cerr`) završiti u datoteci `greske.txt`.
Postoje načini da se koriste datoteke i bez redirekcije.

9.5.1 Ulazni tok za učitavanje iz tekstualne datoteke (`ifstream`)

Pored standardnog ulaznog toka, korisnik može kreirati i koristiti druge ulazne tokove da bi učitao podatke (tekst) iz datoteka. Tip podataka kojim se predstavlja ulazni tok vezan za datoteku je `ifstream` (za njegovo korišćenje je potrebno da budu uključena zaglavlja `<iostream>` i `<fstream>`). Kreiranje ulaznog toka i njegovo povezivanje sa datotekom se vrši pomoću konstruktora, čiji je parametar putanja do datoteke (putanja može biti bilo apsolutna, bilo relativna, u odnosu na radni direktorijum programa, tj. na direktorijum iz kog je program pokrenut, a ne nužno na direktorijum u kom je izvršivi program smešten). Na primer, za čitanje podataka iz datoteke `ulaz.txt` možemo kreirati sledeći tok:

```
ifstream ulaz("ulaz.txt");
```

Alternativno, možemo deklarisati tok, pa ga tek kasnije povezati sa datotekom korišćenjem metode `open`.

```
ifstream ulaz;
ulaz.open("ulaz.txt");
```

Ako navedena putanja ne postoji ili ako korisnik nema pravo čitanja podataka iz navedene datoteke, kreiranje toka neće uspeti. Kreiranje toka može biti neuspešno i iz drugih razloga, te je poželjno proveriti da li je kreiranje prošlo uspešno. Tu proveru je moguće uraditi u sledećem obliku:

```
if (!ulaz.is_open()) {
   cerr << "Greska pri otvaranju datoteke" << endl;
   return 1;
}
```

Nakon uspešnog otvaranja toka, podaci se iz njega mogu učitavati na isti način kao u slučaju toka `cin`. Na primer,

```
int x;
ulaz >> x; // ucitavamo jedan broj
string rec;
ulaz >> rec; // ucitavamo jednu rec (do prve beline)
string linija;
getline(ulaz, linija); // ucitavamo jednu liniju teksta
```

Na primer, naredni deo programa ispisuje celokupan sadržaj tekstualnog ulaznog toka `ulaz` na standardni izlaz.

```
string linija;
while (getline(ulaz, linija)) {
   cout << linija << endl;
}
```

Metodom `ignore()` preskače se čitanje jednog karaktera, dok se izrazom oblika `ulaz >> ws` ili `cin >> ws` preskaču sve vodeće beline (`std::ws` je takozvani *manipulator toka*; `ws` dolazi od *whitespace*). Ilustrujmo primenu ovakvog izraza. Neka je potrebno učitati sa ulaza jedan ceo broj, a zatim u narednom redu ime i prezime (koje može sadržati i razmake), a na ulazu je zadato:

```
123
Ana Lazic
```

Ukoliko se učitavanje vrši na sledeći način:

```
int x;
string ime;
cin >> x;
getline(cin, ime);
```

niska `ime` će biti prazna. Naime, operator `>>` čita broj `123`, ali ne i karakter `'\n'` iza njega, pa funkcija `getline` pročita samo ostatak prvog reda (koji je prazan). Karakter `'\n'` biće preskočen i niska `ime` će dobiti vrednost `Ana Lazic` ukoliko se koristi `cin >> x >> ws;` (ili samo `cin >> ws;` pre poziva funkcije `getline`). Naglasimo da operator `>>` sam preskače vodeće beline, pa manipulator `ws` ima smisla koristiti pre funkcija koje to ne rade, kao što su `getline` i `get`.
Datumi u formi `03/07/2024` mogu da se učitavaju na sledeći način (koji ignoriše karaktere `'/'`):

```
int dan, mesec, godina;
cin >> dan; cin.ignore(); cin >> mesec; cin.ignore(); cin >>
 ↪  godina;
```

Nakon svakog pokušaja čitanja iz ulaznog toka, moguće je proveriti da li je čitanje bilo uspešno proverom stanja ulaznog toka. Metodom `fail()` proverava se da li je došlo do greške prilikom čitanja, a metodom `eof()` da li je prilikom pokušaja čitanja dostignut kraj datoteke.
Nakon učitavanja svih potrebnih podataka, tok treba zatvoriti. To se može uraditi ili eksplicitno, metodom `close` (što je poželjno), ili će se tok automatski zatvoriti prilikom uklanjanja odgovarajuće promenljive (što se automatski dešava na kraju bloka u kom je promenljiva definisana).

9.5.2 *Izlazni tok za upis u tekstualnu datoteku (`ofstream`)*

Upis u datoteku se vrši na sličan način kao i učitavanje. Umesto ulaznog toka `ifstream`, koristi se izlazni tok `ofstream`, koji se sa datotekom u koju će se vršiti upis povezuje na isti način kao i `ifstream`.

```
ofstream izlaz("izlaz.txt");
```

ili

```
ofstream izlaz;
izlaz.open("izlaz.txt");
```

Upis podataka u datoteku se zatim može vršiti primenom operatora << (na isti način kao što smo ispisivali podatke na tok `cout`).
Na kraju korišćenja toka, poželjno je zatvoriti ga metodom `close()`.
Naredni program kopira sadržaj datoteke `ulaz.txt` u datoteku `izlaz.txt`, pretvarajući pritom sva mala slova u velika.

```
#include <fstream>
#include <iostream>
#include <string>
#include <algorithm> // zbog transform
#include <cctype>    // zbog toupper

using namespace std;

int main() {
    // otvaramo ulaz.txt za citanje
    ifstream ulaz("ulaz.txt");
    if (!ulaz.is_open()) {
        cerr << "Greska pri otvaranju datoteke ulaz.txt!" <<
 ↪  endl;
        return 1;
    }

    // otvaramo izlaz.txt za pisanje
    ofstream izlaz("izlaz.txt");
    if (!izlaz.is_open()) {
        cerr << "Greska pri otvaranju datoteke izlaz.txt!"
 ↪  << endl;
        return 1;
    }

    // citamo liniju po liniju iz ulazne datoteke
    string linija;
    while (getline(ulaz, linija)) {
        // pretvaramo mala u velika slova
```

```
        transform(linija.begin(), linija.end(),
↪   linija.begin(), ::toupper);

        // upisujemo transformisanu liniju na izlaz
        izlaz << linija << endl;
    }

    // zatvaramo datoteke
    ulaz.close();
    izlaz.close();

    cout << "Izlaz je sacuvan u datoteci izlaz.txt!" << endl;
    return 0;
}
```

9.5.3 *Baferovanje*

Da bi se smanjio broj pristupa disku ili SSD uređaju, tokovi koriste tehniku *baferovanog izlaza*. To znači da se podaci koje treba upisati u tok privremeno smeštaju u glavnu memoriju računara u tzv. bafer (engl. buffer), tj. prihvatnik, i odlaže se njihov stvarni upis u datoteku (sve dok ih, po nekom kriterijumu, nema dovoljno da bi se pristupilo disku ili SSD uređaju). Pražnjenje bafera može se zahtevati metodom `flush`. Često se (u tekstualnim datotekama) za prelazak u novi red koristi `endl` i tom prilikom se ova metoda poziva iza scene, tj. sigurni smo da će posle svakog poziva ispisa koji uključuje `endl` podaci biti predati operativnom sistemu (koji ih zatim upisuje na disk). Ovo je često poželjno ponašanje, ali nekada može usporiti ispis (naročito u slučaju ispisa velikih količina podataka) i tada je umesto `endl` poželjno koristiti samo oznaku prelaska u novi red `'\n'`, koja ne uzrokuje pražnjenje bafera.

Prečesto pražnjenje bafera je jedan od mogućih razloga velikog usporenja rada programa. Naročito su opasni programi koji naizmenično, u petlji, vrše čitanje podataka sa standardnog ulaza i ispis na standardni izlaz. Naime, prilikom svakog čitanja sa standardnog ulaza vrši se pražnjenje bafera standardnog izlaza. Motivacija za to je sledeća. Često se u programima pre čitanja podataka sa standardnog ulaza korisniku prikazuje poruka koje podatke treba da unese:

```
string ime;
cout << "Kako se zoves?";
cin >> ime;
```

Da bi se osiguralo da će korisnik videti poruku pre unosa podataka, sistem je izgrađen tako da se pre svakog učitavanja sa `cin` poziva metoda `flush` za tok `cout`.

9.5.4 Odnos sa C bibliotekom za ulaz/izlaz

Jezik C++ zadržava kompatibilnost sa programskim jezikom C i podržava korišćenje standardnog C zaglavlja `<cstdio>`, u kojem su deklarisane mnoge funkcije za rad sa datotekama (najpoznatije su `printf`, `scanf`, `fopen`, `fclose`, `fscanf`, `fprintf`, `getchar`, `fgetc` itd.). Iako mešanje biblioteka `<iostream>` i `<fstream>` sa jedne strane i `<cstdio>` sa druge strane nije preporučljivo, ono ne dovodi do greške. Međutim, to ima svoju cenu, jer je prilikom svakog čitanja ili upisa podataka potrebno izvršiti sinhronizaciju ovih biblioteka, što usporava program (ukoliko sistem nije specifično instruiran).

9.5.5 Tokovi za čitanje iz niske (`istringstream`) i upis u nisku (`ostringstream`)

`istringstream` i `ostringstream` su klase iz zaglavlja `<sstream>`, koje rade sa niskama, za razliku od `ifstream` i `ofstream`, koje rade sa datotekama. Tok tipa `istringstream` omogućava čitanje podataka iz niske kao da je u pitanju ulazni tok (na primer, kao `cin`).

```
istringstream ulaz("123 456");
int a, b;
ulaz >> a >> b; //  123 i 456 iz niske
```

Slično, `ostringstream` omogućava upisivanje podataka u nisku, kao da je u pitanju izlazni tok (na primer, kao `cout`). Nakon što se podaci upišu u tok, niska se može dobiti metodom `str`.

```
int x = 123;
ostringstream oss;
oss << "Zdravo, " << "svete! " << x << '\n';
string niska = oss.str(); // "Zdravo, svete! 123\n"
```

9.5.6 Argumenti komandne linije programa

Jedan način da se određeni podaci proslede programu je i da se navedu u komandnoj liniji prilikom njegovog pokretanja. Argumenti koji su tako navedeni prenose se programu kao argumenti funkcije `main`. Prvi argument (koji se obično naziva `argc`, od engleskog *argument count*) je broj argumenata komandne linije (uključujući i sam naziv programa) navedenih prilikom pokretanja programa. Drugi argument (koji se obično naziva `argv`, od engleskog *argument vector*) je niz niski karaktera koje sadrže argumente — svaka niska direktno odgovara jednom argumentu. Iz istorijskih razloga, tj. iz razloga

kompatibilnosti sa programskim jezikom C, te niske nisu predstavljene tipom `string`, već su predstavljene nizovima karaktera tako da se na kraju svakog tako zadatog niza karaktera nalazi specijalni karakter `'\0'`, što je uobičajeni način predstavljanja niski u programskom jeziku C. Nazivu programa odgovara niska `argv[0]`. Ako je `argc` tačno 1, to znači da ne postoje dodatni argumenti nakon imena programa. Dakle, `argv[0]` je ime programa, `argv[1]` do `argv[argc-1]` su tekstovi argumenata programa, a element `argv[argc]` sadrži nulti pokazivač (`nullptr`).
Identifikatori `argc` i `argv` su proizvoljni i funkcija `main` može biti deklarisana i na sledeći način:

```
int main (int br_argumenata, char* argumenti[]);
```

Naredni jednostavan program štampa broj argumenata i sadržaj vektora argumenata komandne linije:

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

int main(int argc, char* argv[]) {
   int i;
   cout << "argc = " << argc << endl;
   // nulti argument uvek je ime programa (na primer, a.out)
   for (i = 0; i < argc; i++)
       cout << "argv[" << i << "] = " << argv[i] << endl;
   return 0;
}
```

Ukoliko se program prevede sa `g++ -o echoargs echoargs.cpp` i pozove sa `./echoargs -U zdravo svima "dobar dan"`, ispisaće:

```
argc = 5
argv[0] = ./echoargs
argv[1] = -U
argv[2] = zdravo
argv[3] = svima
argv[4] = dobar dan
```

Primetimo da ispis ispravno radi, iako na tok `cout` nije prosleđena niska tipa `string`, već niska `argv[i]`, što je adresa početka niza karaktera na čijem se kraju nalazi specijalni karakter `'\0'` (to je podatak tipa `char*`).

Niske kojima su zadati argumenti komandne linije ne bi trebalo menjati tokom rada programa.
Čest vid upotrebe argumenata komandne linije je da se programu navedu različite opcije. Obično se opcije označavaju pojedinačnim karakterima i navode se iza karaktera - (ili iza karaktera -- ili, na Windows sistemima, iza karaktera /). Pri tome, često je predviđeno da se iza jednog simbola - može navesti više karaktera koji određuju opcije. Takav je, na primer, poziv:

```
./program -a -bcd 134 -ef zdravo
```

Naredni program ispisuje sve opcije pronađene u komandnoj liniji:

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

int main(int argc, char* argv[]) {
   // Za svaki argument komandne linije, pocevsi od argv[1]
   // (preskace se ime programa)
   for (int i = 1; i < argc; i++) {
      // Ukoliko i-ti argument pocinje crticom
      if (argv[i][0] == '-') {
          // ispisuju se sva njegova slova od pozicije 1
          int j;
          for (j = 1; argv[i][j] != '\0'; j++)
              cout << "Prisutna je opcija: " << argv[i][j]
 ↪  << endl;
      }
   }
   return 0;
}
```

Pre upotrebe, niske karaktera na čijem se kraju nalazi `'\0'` možemo konvertovati u tip `string`.

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

int main(int argc, char* argv[]) {
    // Za svaki argument komandne linije, pocevsi od argv[1]
    // (preskace se ime programa)
```

```
    for (int i = 1; i < argc; i++) {
        // pravimo podatak tipa string na osnovu i-tog
        ↪ argumenta
        string arg = argv[i];
        // proveravamo da li argument pocinje crticom
        if (!arg.empty() && arg[0] == '-') {
            // ispisuju se sva njegova slova od pozicije 1
            for (size_t j = 1; j < arg.length(); j++)
                cout << "Prisutna je opcija: " << arg[j] <<
↪   endl;
        }
    }
    return 0;
}
```

10. Principi pisanja programa i dokumenta

Programi napisani na višem programskom jeziku sredstvo su komunikacije između čoveka i računara, ali i između ljudi samih. Razumljivost, čitljivost programa, iako nebitna za računar, od ogromne je važnosti za kvalitet i upotrebljivost programa. Naime, u održavanje programa obično se uloži daleko više vremena i truda nego u njegovo pisanje, a održavanje sistema često ne rade oni programeri koji su program napisali. Pored toga, razumljivost programa omogućava lakšu analizu njegove ispravnosti i složenosti. Preporuke za pisanje često nisu kruta pravila, već predstavljaju samo smernice i ideje kojima se treba rukovoditi u pisanju programa, u aspektima formatiranja, nazubljivanja, imenovanja promenljivih i funkcija, itd.

U daljem tekstu će, kao jedan primer konvencija za pisanje programa, biti ukazivano na preporuke iz teksta *Linux Kernel Coding Style* Linusa Torvaldsa, autora operativnog sistema Linux, koji je napisan na jeziku C. Nekoliko saveta i preporuka u nastavku preuzeto je iz znamenite knjige *The Practice of Programming* autora Brajana Kernigena i Roba Pajka. Preporuke navedene u nastavku često se odnose na sve programske jezike, ali ponekad samo na jezike C/C++. I sve ove savete i preporuke treba razmatrati sa rezervom, jer postoje i mnoge druge grupe preporuka i konvencija.

10.1 Timski rad i konvencije

Za svaki obimniji projekat potrebno je usaglasiti konvencije za pisanje programa. Da bi ih se lakše pridržavalo, potrebno je detaljno motivisati i obrazložiti pravila. Ima različitih konvencija i one često izazivaju duge i zapaljive rasprave između programera. Mnogi će, međutim, reći da nije najvažnije koja konvencija se koristi, nego koliko strogo se nje pridržava. Strogo i konzistentno pridržavanje konvencije u okviru jednog projekta izuzetno je važno za njegovu uspešnost. Jedan isti programer treba da bude spreman da u različitim timovima i različitim projektima koristi različite konvencije.

Kako bi se olakšalo baratanje programom koji ima na stotine datoteka koje menja veliki broj programera, u timskom radu obično se koriste *sistemi za upravljanje verzijama*

(engl. version control system), kao što su git, SVN, CVS, Mercurial, Bazaar. I ovi sistemi nameću dodatna pravila i omogućavaju dodatne konvencije koje tim treba da poštuje (na primer, konvencija može da bude da u zajedničku verziju programa ne može da se stavi datoteka sa kojom se čitav program ne kompilira uspešno).

10.2 Vizuelni elementi programa

Prva ideja o programu formira se na osnovu njegovog izgleda – njegovih vizuelnih elemenata, kao što su broj linija u datoteci, broj karaktera u liniji, nazubljivanje, grupisanje linija i slično. Vizuelni elementi programa i njegovo formatiranje često su od ključne važnosti za njegovu čitljivost. Formatiranje, konkretno nazubljivanje, u nekim jezicima (na primer, Python) čak utiče na značenje programa.

Formatiranje i vizuelni elementi programa treba da olakšaju razumevanje koda koji se čita, ali i pronalaženje potrebnog dela koda ili datoteke sa nekim delom programa. Formatiranje i vizuelni elementi programa treba da olakšaju i proces pisanja programa. U tome, pomoć autoru programa mogu da pružaju alati u okviru kojih se piše program – specijalizovani editori teksta ili editori koji su deo integrisanih razvojnih okruženja (engl. IDE, Integrated Development Environment) koja povezuju editor, kompilator, debager i druge alatke potrebne u razvoju softvera. Neke od namenskih alatki koje olakšavaju pisanje programa su: „ulepšivači“ (engl. beautifier), poput programa `indent`, koji mogu da formatiraju već kreirane datoteke sa programskim kodom; programi za proveru pravopisa, koji mogu da otkriju jednostavne leksičke i sintaksičke greške u programu i da nude moguće ispravke; „linteri“, programi koji vrše statičku analizu programa i koji mogu da ukažu na određene stilske (ali i ozbiljnije) greške, itd.

10.2.1 Broj karaktera u redu

U modernim programskim jezicima dužina reda programa nije ograničena.[1] Ipak, predugi redovi mogu da stvaraju probleme. Na primer, predugi redovi mogu da zahtevaju horizontalno „skrolovanje“ kako bi se video njihov kraj, što može da drastično oteža čitanje i razumevanje programa. Takođe, ukoliko se program štampa, dugi redovi mogu da budu presečeni i da naruše formatiranje. Zbog ovih i ovakvih problema, preporučuje se pridržavanje nekog ograničenja – obično 80 karaktera u redu. Konkretna preporuka za 80 karaktera u redu je istorijska i potiče od ograničenja na bušenim karticama, starim ekranima i štampačima. Ipak, ona je i danas široko prihvaćena kao pogodna. Ukoliko red programa ima više od 80 karaktera, to najčešće ukazuje na to da kôd treba reorganizovati uvođenjem novih funkcija ili promenljivih. Broj 80 (ili bilo koji drugi) kao ograničenje za broj karaktera u redu ne treba shvatati kruto, već kao načelnu preporuku koja može biti narušena ako se tako postiže bolja čitljivost.

[1] Iako za većinu jezika standard ne propisuje maksimalnu dužinu reda, često je za konkretne kompilatore dužina reda programa ograničena nekim velikim brojem, daleko većim od uobičajenih dužina redova.

10.2.2 Broj naredbi u redu, zagrade i razmaci

Red programa može da bude prazan ili da sadrži jednu ili više naredbi. Prazni redovi mogu da izdvajaju blokove blisko povezanih naredbi (na primer, blok naredbi za koje se može navesti komentar o tome šta je njihova svrha). Ako se prazni redovi koriste neoprezno, mogu da naruše umesto da poprave čitljivost. Naime, ukoliko ima previše praznih linija, smanjen je deo koda koji se može videti i sagledavati istovremeno na ekranu. Po jednoj konvenciji, zagrade koje označavaju početak i kraj bloka navode se u zasebnim redovima (u istoj koloni), a po drugoj, otvorena zagrada se navodi u nastavku naredbe, a zatvorena u zasebnom redu ili u redu zajedno sa ključnom rečju `while` ili `else`. Torvalds preporučuje ovu drugu konvenciju, uz izuzetak da se otvorena vitičasta zagrada na početku definicije funkcije piše u zasebnom redu.
Naredni primer prikazuje deo koda napisan sa većim brojem praznih redova i prvom konvencijom za zagrade:

```
for (int i = 0; i < n-1; i++)
{

  int m = i;

  for (int j = i+1; j < n; j++)
  {

    if (a[j] < a[m])
       m = j;
  }

  swap(a[i], a[m]);
}
```

Isti deo koda može biti napisan sa manjim brojem praznih redova i drugom konvencijom za zagrade. Ovaj primer prikazuje kompaktnije zapisan kôd koji je verovatno čitljiviji većini iskusnih C/C++ programera:

```
for (int i = 0; i < n-1; i++) {
  int m = i;
  for (int j = i+1; j < n; j++) {
    if (a[j] < a[m])
       m = j;
  }
```

```
    swap(a[i], a[m]);
}
```

Jedan red može da sadrži i više od jedne naredbe. To je prihvatljivo samo (a tada može da bude i preporučljivo) ako se radi o jednostavnim i na neki način povezanim inicijalizacijama ili jednostavnim dodelama vrednosti članovima strukture, na primer:

```
...
int i = 10; double suma = 0;
tacka.x = 0; tacka.y = 0;
```

Ukoliko je u petlji ili u `if` bloku samo jedna naredba, onda nisu neophodne zagrade koje označavaju početak i kraj bloka i mnogi programeri ih ne pišu. Međutim, iako nisu neophodne, one mogu olakšati razumevanje koda u kojem postoji višestruka `if` naredba. Dodatno, ukoliko se u blok sa jednom naredbom i bez vitičastih zagrada u nekom trenutku doda druga naredba, lako može da se previdi da postaje neophodno navesti i zagrade.

Veličina blokova koda je takođe važna za preglednost, pa je jedna od preporuka da vertikalno rastojanje između otvorene vitičaste zagrade i zatvorene vitičaste zagrade koja joj odgovara ne bude veće od jednog ekrana.

Obično se preporučuje navođenje razmaka oko ključnih reči i oko binarnih operatora, izuzev `.` i `->`. Ne preporučuje se korišćenje razmaka kod poziva funkcija i unarnih operatora, izuzev (eventualno) kod operatora `sizeof` i operatora kastovanja. Ne preporučuje se navođenje nepotrebnih zagrada, posebno u okviru povratne vrednosti. Na primer:

```
if (uslov) {
    *a = -b + c + sizeof (int) + f(x);
    return -1;
}
```

10.2.3 *Nazubljivanje teksta programa*

Nazubljivanje teksta programa za većinu programskih jezika (uključujući jezike C/C++ i Javu) nebitno je kompilatoru, ali je skoro neophodno programeru. Nazubljivanje naglašava strukturu programa i olakšava njegovo razumevanje. Red programa može biti uvučen u odnosu na početnu kolonu za nekoliko blanko karaktera ili nekoliko tab karaktera. Tab karakter može da se u okviru editora interpretira na različite načine (tj. kao

različit broj belina), te je preporučljivo u programu sve tab karaktere zameniti razmacima (za šta u većini editora postoji mogućnost) i čuvati ga u tom obliku. Na taj način, svako će videti program (na ekranu ili odštampan) na isti način.

Ne postoji kruto pravilo za broj karaktera za jedan nivo uvlačenja. Neki programeri koriste 4, a neki 2 – sa motivacijom da u redovima od 80 karaktera može da stane i kôd sa dubokim nivoima. Torvalds, sa druge strane, preporučuje broj 8, jer omogućava bolju preglednost. Za delove programa koji imaju više od tri nivoa nazubljivanja, on kaže da su ionako sporni i zahtevaju prepravku.

10.3 Imenovanje promenljivih i funkcija

Imenovanje promenljivih i funkcija veoma je važno za razumljivost programa i sve je važnije što je program duži. Pravila imenovanja mogu da olakšaju i izbor novih imena tokom pisanja programa. Imena promenljivih i funkcija (pa i datoteka programa) treba da sugerišu njihovu ulogu i tako olakšaju razumevanje programa.

Globalne promenljive, strukture i funkcije treba da imaju opisna imena, potencijalno sačinjena od više reči. U *kamiljoj* notaciji (popularnoj među Java i C++ programerima), imena od više reči zapisuju se tako što svaka nova reč (sem eventualno prve) počinje velikim slovom, na primer, `brojKlijenata`. U notaciji sa podvlakama (popularnoj među C programerima), sve reči imena pišu se malim slovima, a reči su razdvojene podvlakama, na primer, `broj_klijenata`. Imena makroa i konstanti pišu se obično svim velikim slovima, a imena globalnih promenljivih počinju velikim slovom.

Lokalne promenljive, a posebno promenljive koje se koriste kao brojači u petljama, treba da imaju kratka i jednostavna, a često najbolje, jednoslovna imena – jer se razumljivost lakše postiže sažetošću. Imena za brojače u petljama su često `i`, `j`, `k`, za pokazivače `p` i `q`, a za niske `s` i `t`. Preporuka je i da se lokalne promenljive deklarišu što kasnije u okviru funkcije i u okviru bloka u kojem se koriste (a ne u okviru nekog šireg bloka).

Jedan, delimično šaljiv, savet za imenovanje (i globalnih i lokalnih) promenljivih kaže da broj karaktera u imenu promenljive treba da zavisi od broja linija njenog dosega i to tako da bude proporcionalan logaritmu broja linija njenog dosega.

Za promenljive i funkcije nije dobro koristiti generička imena kao što su `rezultat`, `izracunaj(...)`, `uradi(...)`, već sugestivnija, kao što su, na primer, `kamata`, `izracunaj_kamatu(...)`, `odstampaj_izvestaj_o_kamati(...)`.

Imena funkcija dobro je da budu bazirana na glagolima, na primer, bolje je `izracunaj_kamatu(...)` nego `kamata(...)` i `get_time(...)` nego `time(...)`. Za funkcije koje vraćaju istinitosnu vrednost, ime treba da sugeriše u kom slučaju se vraća vrednost *tačno*, na primer, bolje je ime `is_prime(...)` nego `check_prime(...)`.

Mnoge promenljive označavaju neki broj entiteta (na primer, broj klijenata, broj studenata, broj artikala) i za njih se može usvojiti konvencija po kojoj imena imaju isti prefiks ili sufiks (na primer, `br_studenata` ili `num_students`).

I programeri kojima to nije maternji jezik, iako to nije zahtev projekta, često imenuju promenljive i funkcije na osnovu reči engleskog jezika. To je posledica istorijskih razloga i dominacije engleskog jezika u programerskoj praksi, kao i samih ključnih reči skoro svih programskih jezika (koje su na engleskom). Prihvatljivo je (ako nije zahtev projekta drugačiji) imenovanje i na maternjem jeziku i na engleskom jeziku — jedino je neprihvatljivo mešanje ta dva. Imenovanje na bazi engleskog i komentari na engleskom mogu biti pogodni ukoliko postoji i najmanja mogućnost da se izvorni program koristi u drugim zemljama, ili od strane drugih timova, ili da se učini javno dostupnim i slično. Naime, u programiranju (kao i u mnogim drugim oblastima) engleski jezik je opšteprihvaćen u svim delovima sveta i tako se može osigurati da program lakše razumeju svi.
Neki programeri smatraju da se kvalitet imenovanja promenljivih i funkcija može „testirati“ na sledeći zanimljiv način: ako se kôd može pročitati preko telefona tako da ga sagovornik na drugoj strani razume, onda je imenovanje dobro.

10.4 Pisanje izraza

Za dobrog programera neophodno je da poznaje sva pravila programskog jezika jer će verovatno češće i više raditi na tuđem nego na svom kodu. S druge strane, programer u svojim programima ne mora i ne treba da koristi sva sredstva izražavanja tog programskog jezika, već može i treba da ih koristi samo delom, oprezno i uvek sa ciljem pisanja razumljivih programa. Ponekad programer ulaže veliku energiju u pisanje najkonciznijeg mogućeg koda, što može da bude protraćen trud, jer je obično važnije da kôd bude jasan, a ne kratak. Sve ovo odnosi se na mnoge aspekte pisanja programa, uključujući pisanje izraza.
Preporučuje se pisanje izraza u jednostavnom i intuitivno jasnom obliku. Na primer, umesto:

```
!(c < '0') && !(c > '9')
```

bolje je:

```
'0' <= c && c <= '9'
```

Zagrade, čak i kada nisu neophodne, nekome ipak mogu da olakšaju čitljivost. Prethodni primer može da se zapiše i na sledeći način:

```
('0' <= c) && (c <= '9')
```

Slično, naredbi

```
prestupna = g % 4 == 0 && g % 100 != 0 || g % 400 == 0;
```

ekvivalentna je naredba

```
prestupna = ((g % 4 == 0) && (g % 100 != 0)) || (g % 400 ==
 ↪  0);
```

koja se može smatrati znatno čitljivijom. Naravno, čitljivost je subjektivna, te su moguća i razna međurešenja. Na primer, moguće je podrazumevati da programer jasno razlikuje aritmetičke, relacijske i logičke operatore i da njihov prioritet razlikuje i bez navođenja zagrada, a da se zagrade koriste da bi se naglasila razlika u prioritetu operatora iste vrste (na primer, između logičkih operatora `&&` i `||`). Dodatno, ako ona označava godinu, bolje ime za promenljivu g je `godina`, pa se time dolazi do naredbe:

```
prestupna = (godina % 4 == 0 && godina % 100 != 0) ||
            (godina % 400 == 0);
```

Iako je opšta preporuka da se navodi razmak oko binarnih operatora, neke konvencije preporučuju izostavljanje tih razmaka u dužim izrazima, i to oko operatora višeg prioriteta, čime se zapisom sugeriše prioritet operatora, kao u sledećem primeru:

```
a*b + c*d
```

Suviše komplikovane izraze treba zameniti jednostavnijim i razumljivijim. Na primer, umesto

```
x *= (c +=  a < b ? f("a") : f("b"));
```

možemo koristiti daleko čitljiviju narednu varijantu:

```
if (a < b)
   c += f("a");
else
   c += f("b");
x *= c;
```

Kernigen i Pajk navode i primer u kojem je moguće i poželjno pojednostaviti komplikovana izračunavanja. Ako je potrebno izdvojiti tri bita najmanje težine iz broja `bitoff`, umesto izraza:

```
bitoff - ((bitoff >> 3) << 3)
```

bolje je koristiti (ekvivalentan) izraz:

```
bitoff & 0x7
```

Zbog komplikovanih, a u nekim situacijama i nedefinisanih, pravila poretka izračunavanja i dejstva propratnih efekata (kao, na primer, kod operatora inkrementiranja i dekrementiranja), dobro je pojednostaviti kôd kako bi njegovo izvršavanje bilo jednoznačno i jasno. Na primer, umesto:[2]

```
str[i++] = str[i++] = ' ';
```

bolje je:

```
str[i++] = ' ';
str[i++] = ' ';
```

Poučan je i sledeći čuveni primer: nakon dodele `a[a[1]]=2;`, element `a[a[1]]` nema nužno vrednost 2 (ako je na početku vrednost `a[1]` bila jednaka 1, a vrednost `a[2]` različita od 2). Navedeni primer pokazuje da treba biti veoma oprezan sa korišćenjem indeksa niza koji su i sami elementi niza ili neki komplikovani izrazi.

10.5 Korišćenje idioma

Idiomi su ustaljene jezičke konstrukcije koje predstavljaju celinu. Idiomi postoje u svim jezicima, pa i u programskim. Tipičan idiom u jeziku C je sledeći oblik `for`-petlje:

```
for (i = 0; i < n; i++)
  ...
```

Kernigen i Pajk zagovaraju korišćenje idioma gde god je to moguće. Na primer, umesto varijanti

[2] Od standarda C++17 ponašanje ovog konkretnog izraza je definisano (desna strana dodele izračunava se pre leve), ali je on i dalje teško razumljiv.

```
i = 0;
while (i <= n-1)
  a[i++] = 1.0;
```

```
for (i = 0; i < n; )
  a[i++] = 1.0;
```

```
for (i = n; --i >= 0; )
  a[i] = 1.0;
```

smatraju da je bolja varijanta:

```
for (i = 0; i < n; i++)
  a[i] = 1.0;
```

jer je najčešća i najprepoznatljivija. Štaviše, Kernigen i Pajk predlažu, pomalo ekstremno, da se bez dobrog razloga i ne koristi nijedna forma `for`-petlji osim navedene. Kao idiom za beskonačnu petlju navode:

```
for (;;)
  ...
```

Ova preporuka potiče iz jezika C, u kome tip `bool` i vrednost `true` dugo nisu postojali, pa se biralo između `for (;;)` i `while (1)`. U jeziku C++ je podjednako ustaljen oblik `while (true)`, koji se u ovoj knjizi koristi jer jasnije iskazuje nameru (videti poglavlje 6.2). Važno je da se u okviru jednog projekta dosledno koristi jedan oblik.
Glavni argument za korišćenje idioma je da se kôd brzo razume, a i da svaki drugi („neidiomski“) konstrukt privlači dodatnu pažnju, što je dobro, jer se bagovi češće kriju u njima.

10.6 Korišćenje konstanti

Konstantne vrednosti, veličine nizova, pozicije karaktera u niskama, faktori za konverzije i druge slične vrednosti koje se pojavljuju u programima često se zovu *magični brojevi* (jer obično nije jasno odakle dolaze i na osnovu čega su dobijeni). Kernigen i

Pajk kažu da je, osim 0 i 1, svaki broj u programu kandidat da se može smatrati magičnim, te da treba da ima ime koje mu je pridruženo. Na taj način, ukoliko je potrebno promeniti vrednost magične konstante (na primer, maksimalna dužina imena ulice) – to je dovoljno uraditi na jednom mestu u kodu. Na primer, u narednoj deklaraciji

```
char imeUlice[50];
```

pojavljuje se magična konstanta 50, te se u nastavku programa broj 50 verovatno pojavljuje u svakoj obradi imena ulica. Promena tog ograničenja zahtevala bi mnoge izmene koje ne bi mogle da se sprovedu automatski (jer se broj 50 možda pojavljuje i u nekom drugom kontekstu). Zato je bolja, na primer, varijanta kojom se magičnom broju pridružuje simboličko ime pretprocesorskom direktivom `#define`:

```
#define MAKS_IME_ULICE 50
char imeUlice[MAKS_IME_ULICE];
```

U tom slučaju, pretprocesor zamenjuje sva pojavljivanja tog imena konkretnom vrednošću pre procesa kompilacije, te kompilator (pa i debager) nema nikakvu informaciju o simboličkom imenu koje je pridruženo magičnoj konstanti niti o njenom tipu. Zbog toga se preporučuje da se magične konstante uvode kao konstantne promenljive, ako upotreba to dozvoljava:

```
const unsigned int MAKS_IME_ULICE = 50;
```

Naglasimo da se u jeziku C++ konstantna promenljiva celobrojnog tipa koja je inicijalizovana konstantnim izrazom i sama smatra konstantnim izrazom, pa se, na primer, može koristiti i za dimenzije nizova (u jeziku C to nije slučaj i tamo se za tu svrhu koriste simbolička imena uvedena direktivom `#define` ili nabrojive konstante).

U jednom stilu pisanja funkcija, rezultati funkcije se vraćaju kroz listu argumenata, a povratna vrednost ukazuje na to da li je funkcija uspešno obavila zadatak. Za povratne vrednosti onda postoje dve česte konvencije: jedna je da se vraća istinitosna vrednost *tačno* (`true`, ako se koristi tip `bool`, ili `1`, ako je povratni tip ceo broj), ako je funkcija uspešno obavila zadatak, a *netačno* (`false` ili `0`) inače. Druga konvencija je da se vraća nešto detaljnija informacija, te da se vraća `0` ako je izvršavanje funkcije proteklo bez problema, a nekakav celobrojni kôd greške inače. Kodovi greške nikako ne treba da budu magične konstante, već mogu biti predstavljeni simboličkim imenima ili, još bolje, enumerisanim konstantama.

U većim programima, konstante od značaja za čitav program (ili veliki njegov deo) obično se čuvaju u zasebnoj datoteci zaglavlja (koju koriste sve druge datoteke kojima su ove konstante potrebne).

Konstante se u programima mogu koristiti i za kodove karaktera. To je loše ne samo zbog narušene čitljivosti, već i zbog narušene prenosivosti – naime, nije na svim računarima podrazumevana ASCII karakterska tabela. Dakle, umesto, na primer:

```
if (65 <= c && c <= 90)
  ...
```

bolje je pisati

```
if ('A' <= c && c <= 'Z')
  ...
```

a još bolje koristiti funkcije iz standardne biblioteke, kad god je to moguće:

```
if (isupper(c))
  ...
```

Slično, zarad bolje čitljivosti treba pisati `nullptr` (za nultu vrednost pokazivača; u jeziku C se za to koristi `NULL`) i `'\0'` (za završnu nulu u niskama) umesto konstante `0`.
U programima ne treba koristiti kao konstante ni veličine tipova – zbog čitljivosti, a i zbog toga što se mogu razlikovati na različitim računarima. Zato, na primer, za dužinu tipa `int` nikada ne treba pisati 2 ili 4, već `sizeof(int)`. Za promenljive i elemente niza, bolje je pisati `sizeof(a)` i `sizeof(b[0])` umesto `sizeof(int)` (ako su promenljiva `a` i niz `b` tipa `int`), zbog mogućnosti da se promenljivoj ili nizu u nekoj verziji programa promeni tip.

10.7 Pisanje komentara

Čak i ako se autor pridržavao mnogih preporuka za pisanje jasnog i kvalitetnog koda, ukoliko kôd nije dobro komentarisan, njegovo razumevanje može i samom autoru predstavljati teškoću već nekoliko nedelja nakon pisanja. Komentari treba da olakšaju razumevanje koda i predstavljaju njegov svojevrsni dodatak.
Postoje alati koji olakšavaju kreiranje dokumentacije na osnovu komentara u samom kodu i delom je generišu automatski (na primer, Doxygen).

- **Komentari ne treba da objašnjavaju ono što je očigledno.** Komentari ne treba da govore *kako* kôd radi, već *šta* radi (i zašto). Na primer, naredna dva komentara su potpuno suvišna:

```
k += 1.0;  /* k se uvecava za 1.0 */
```

```
return OK; /* vrati OK */
```

U prvom slučaju, komentar ima smisla ako objašnjava zašto se nešto radi, na primer:

```
k += 1.0;  /* u ovom slucaju, kao bonus, kamatna stopa
              uvecava se za 1.0 */
```

- **Komentari treba da budu koncizni.** Da ne bi oduzimali previše vremena čitaocu, komentari treba da budu što je moguće kraći i jasniji, da ne ponavljaju informacije koje su već navedene drugde u komentarima ili su očigledne iz koda. Previše komentara ili predugi komentari predstavljaju opasnost za čitljivost programa.

- **Komentari treba da budu usklađeni sa kodom.** Ako se promeni kôd programa, a ne i prateći komentari, to može da uzrokuje mnoge probleme i nepotrebne izmene u programu u budućnosti. Ukoliko se neki deo programa promeni, uvek je potrebno proveriti da li je novo ponašanje u skladu sa komentarima (za taj, ali i za druge delove programa). Usklađenost koda i komentara je lakše postići ako komentari ne govore ono što je očigledno iz koda.

- **Komentarima treba objasniti ulogu datoteka i globalnih objekata.** Komentarima treba, na jednom mestu, tamo gde su definisani, objasniti ulogu datoteka, globalnih objekata kao što su funkcije, globalne promenljive i strukture. Funkcije treba komentarisati pre same definicije, a Torvalds čak savetuje da se izbegavaju komentari unutar tela funkcije. Čitava funkcija može da zaslužuje komentar (pre prvog reda), ali ako pojedini njeni delovi zahtevaju komentarisanje, onda je moguće da funkciju treba reorganizovati i/ili podeliti na nekoliko funkcija. Ni ovo pravilo nije kruto, i u specifičnim situacijama prihvatljivo je komentarisanje delikatnih delova funkcije („posebno pametnih ili ružnih“).

- **Loš kôd ne treba komentarisati, već ga popraviti.** Često kvalitetno komentarisanje *kako* i *zašto* neki loš kôd radi zahteva više truda nego pisanje tog dela koda iznova tako da je očigledno kako i zašto on radi. Ipak, ukoliko je kôd veoma komplikovan i teško je ili nemoguće ga pojednostaviti, potrebno je u komentaru objasniti zašto je komplikovan, kako radi i zašto je izabrano takvo rešenje.

- **Komentari treba da budu laki za održavanje.** Treba izbegavati stil pisanja komentara u kojem i mala izmena komentara zahteva dodatni posao u formatiranju. Na primer, promena narednog opisa funkcije zahteva izmene u tri reda komentara:

```
/*************************************
* Funkcija area racuna povrsinu trougla *
*************************************/
```

- **Komentari mogu da uključuju standardne fraze.** S vremenom se nametnulo nekoliko oznaka („markera“) na bazi fraza koje se često pojavljuju u okviru komentara. Njih je lako pronaći u kodu, a mnoga razvojna okruženja prepoznaju ih i prikazuju u istaknutoj boji kako bi privukli pažnju programera (kao svojevrsna lista stvari koje treba obaviti). Najčešći markeri su:
  - `TODO` marker: označava zadatke koje tek treba obaviti, koji kôd treba napisati.
  - `FIXME` marker: označava deo koda koji radi, ali ga treba popraviti, u smislu opštijeg rešenja, lakšeg održavanja ili bolje efikasnosti.
  - `BUG`: označava deo koda koji je gotov i očekuje se da radi, ali je pronađen bag.

Uz navedene markere i prateći tekst, često se navodi i ime onoga ko je uneo komentar, kao i datum unošenja komentara.

10.8 Modularnost

Veliki program je teško ili nemoguće razmatrati ako nije podeljen na celine. Prednosti podele programa na celine (kraći kôd, lakše održavanje, čitljivost, šira upotrebljivost i lakša podela posla u timu), kao i osnovne smernice za pisanje funkcija, opisane su u poglavlju 7.1. Podela programa na celine vrši se još od prvih faza pisanja programa i predstavlja jedan od najvažnijih aspekata dizajna programa.

10.8.1 Modularnost i podela na funkcije

Za mnoge programske jezike osnovni vid postizanja modularnosti je podela koda na funkcije, a kod objektno-orijentisanih jezika važna je i organizacija klasa u programu. Duge funkcije poželjno je podeliti na manje, na primer, na one koje obrađuju specijalne slučajeve.

10.8.2 Modularnost i podela na datoteke

Veliki programi sastoje se od velikog broja datoteka koje bi trebalo da budu organizovane na razuman način u direktorijume. Jednu datoteku treba da čine definicije funkcija koje su međusobno povezane i predstavljaju nekakvu celinu.
Datoteke zaglavlja obično imaju sledeću strukturu:

- definicije tipova;
- definicije konstanti;
- deklaracije globalnih promenljivih;
- deklaracije funkcija.

a izvorne datoteke sledeću strukturu:

- uključivanje sistemskih datoteka zaglavlja;
- uključivanje lokalnih datoteka zaglavlja;
- definicije tipova;
- definicije konstanti;
- deklaracije/definicije globalnih promenljivih;
- definicije funkcija.

Program treba deliti na datoteke imajući u vidu delom suprotstavljene zahteve. Jedna datoteka ne treba da bude duža od nekoliko stotina linija (na primer, dve ili tri stotine). Ukoliko logička struktura programa nameće dužu datoteku, onda vredi preispitati postojeću organizaciju podataka i funkcija. S druge strane, datoteke ne treba da budu prekratke i treba da predstavljaju zaokružene celine. Preterana usitnjenost (u preveliki broj datoteka) može da oteža upravljanje programom i njegovu razumljivost.
Integrisana razvojna okruženja i program `make` (poglavlje 7.12.2) značajno olakšavaju rad sa programima koji su sačinjeni od više datoteka.

10.9 Upravljanje izuzecima i greškama

Svaka od funkcija koje čine program ima neki specifičan zadatak. Generalno se može očekivati da će taj zadatak biti uspešno obavljen, ali postoje mnogi scenariji gde to i nije tako. Na primer,

- ako se tokom izvršavanja funkcije dogodi celobrojno deljenje nulom – obično će doći do greške i prekida izvršavanja programa;
- ako je prekoračena predviđena veličina programskog steka – doći će do greške i prekida izvršavanja programa;
- ako se pristupa oslobođenoj memoriji na hipu – može doći do greške i prekida izvršavanja programa;

- ako se upisuje sadržaj u neki niz nakon njegove granice – može doći do greške i prekida izvršavanja programa.

U nekim situacijama, još neugodnije, program se ne prekida, nego nastavlja sa radom dajući pogrešne rezultate (to su najčešće mesta gde standard jezika ostavlja nedefinisano ponašanje).

Neke od ovih grešaka moguće je i potrebno predupredi. U tu svrhu funkcije, umesto da vraćaju samo rezultat svog rada, mogu da vraćaju (kroz povratnu vrednost, listu argumenata ili na neki drugi način) i nekakvu informaciju o tome da li je zadatak obavljen uspešno (često tu informaciju zovemo „status“). Određivanje, prenos i korišćenje takvih informacija zovemo *upravljanje greškama* (engl. *error handling*).

Generalno, program može da obrađuje i situacije koje logički ne bi smele da se dogode. Time se delovi programa štite od neispravnih ulaza i omogućava nastavak njegovog izvršavanja i u neočekivanim okolnostima. Ovaj pristup programiranju i upravljanju greškama naziva se *odbrambeno programiranje*.
Ipak, nije neophodno, pa ni preporučeno, da se pokušava da se sve moguće greške preduprede jer to vodi komplikovanom kodu teškom za razumevanje i održavanje. Naime, za neke funkcije će se *pretpostavljati* da su neki preduslovi tačni i da je funkcija pozvana na predviđeni način (na primer, u funkcijama koje vrše binarnu pretragu ne proverava se da su elementi niza zaista sortirani – dužnost onoga ko poziva ovu funkciju je da obezbedi da taj preduslov bude ispunjen). Svaki program ima svoju *specifikaciju*, kojom se, između ostalog, definiše dopušten skup ulaznih podataka. Zadatak programera je da obezbedi da program ispravno radi u slučaju kada ulazni podaci zadovoljavaju tu specifikaciju. U slučaju kada ti podaci ne zadovoljavaju specifikaciju, ponašanje programa je nedefinisano jer se ne očekuje da će program biti korišćen sa tim neispravnim ulazima (ispravnost ulaznih podataka je obaveza onoga ko poziva program). Isto važi i za svaku pojedinačnu funkciju. Na primer, funkcija koja vrši binarnu pretragu niza, u slučaju kada niz nije sortiran, može da vrati bilo koju vrednost. Obično se programi koji se pišu tako da ih programer koristi samostalno ili programi koji obrađuju neke podatke koji su automatski generisani i koji su sigurno ispravni, mogu pisati tako da nije potrebno proveravati ispravnost tih ulaznih podataka. Drugim rečima, u mnogim programima prihvatljivo je specifikacijom suziti prostor dopuštenih ulaza i time ga pojednostaviti. S druge strane, programi koji se pišu za širi krug korisnika i programi koji treba da budu robusni i dugotrajni imaju slabije pretpostavke o ispravnosti ulaza, i dužnost programera je da obezbedi proveru ispravnosti ulaza i prijavljivanje odgovarajućih grešaka kada ulaz nije ispravan. U svakom slučaju, programer mora da ima jasno u vidu specifikaciju problema koji rešava i da svoje programe i funkcije tome prilagodi.

U nekim situacijama, preduslovi programa se ne proveravaju na klasičan način, ali se naglašava da su oni podrazumevani naredbom `assert(preduslov);`. U režimu debagovanja, ukoliko `preduslov` nije ispunjen kada se dođe do ove naredbe, program

će prekinuti rad.[3] To može da pomogne u otklanjanju greške, jer kada je program u realnoj upotrebi (takozvana produkciona verzija, engl. release version), situacija u kojoj `preduslov` nije ispunjen apsolutno ne sme da se dogodi (na primer, u softveru koji upravlja avionom ne sme da se dogodi da je trenutna brzina aviona negativan broj), i to mora da obezbedi i garantuje dizajn programa. U ovom kontekstu, naredba `assert(preduslov);` ima i drugu svrhu: da eksplicitno daje informaciju o podrazumevanom uslovu. Slično kao što možemo zahtevati (i obezbediti dizajnom programa) da se neka funkcija može pozvati samo pod nekim uslovima, tako se može zahtevati (i obezbediti dizajnom programa) da se neki delovi koda jedne funkcije izvršavaju samo pod nekim uslovima, koji sprečavaju neke greške. U takvim situacijama nije potrebno (pa ni poželjno) proveravati da li dolazi do greške koja bi trebalo da je onemogućena dizajnom. Na primer, ako se funkcija binarne pretrage u programu poziva nakon poziva funkcije za sortiranje, niz će sigurno biti sortiran i ne bi imalo nikakvog smisla da program vrši eksplicitnu proveru da li je niz zaista sortiran.

Za razliku od jezika C, jezik C++ ima mehanizam *izuzetaka* (engl. exceptions). Programer izdaje posebnu naredbu (`throw`) koja se aktivira u slučaju greške, kojom se prekida kôd koji se trenutno izvršava i tok programa preusmerava se na poseban deo koda koji se bavi obradom grešaka (`catch`). Time se postiže da su normalan tok programa i obrada grešaka fizički razdvojeni u samom kodu, što pojednostavljuje programiranje i čini programe čitljivijim i lakšim za održavanje. Naredni kôd ilustruje mehanizam izuzetaka. U funkciji `deljenje`, izuzetkom ili greškom smatra se situacija kada je delilac (u celobrojnom deljenju) jednak nuli i tada se (u fazi izvršavanja) generiše greška sa objašnjenjem „Deljenje nulom!“. Svaka funkcija koja koristi funkciju `deljenje` (i posredstvom drugih funkcija) može da predvidi mogućnost grešaka i da ih *uhvati* – konstrukcijom `try { ... } catch (tip_izuzetka& e) { ... }`.

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

void deljenje(int a, int b) {
  if (b == 0) {
    throw runtime_error("Deljenje nulom!");
  }
  cout << a / b << endl;
}

int main() {
```

[3] `assert` je makro definisan u zaglavlju `<cassert>`. Provere koje on vrši isključuju se ako je definisan makro `NDEBUG`, što se obično radi prilikom prevođenja produkcione verzije programa.

```
    try {
        deljenje(10, 0); // Ovo deljenje ce izbaciti izuzetak
    } catch (const runtime_error& e) {
        cout << "Greska: " << e.what() << endl;
    }
    try {
        deljenje(10, 2); // Ovo deljenje nece izbaciti izuzetak
    } catch (const runtime_error& e) {
        cout << "Greska: " << e.what() << endl; // hvatanje i
    ↪  upravljanje greskom
    }
    return 0;
}
```

U navedenom programu, poziv `deljenje(10, 0)` dovodi do izuzetka, a poziv `deljenje(10, 2)` ne, pa će biti dobijen naredni izlaz.

```
Greska: Deljenje nulom!
5
```

11. Razvoj programa

11.1 Razvojno okruženje

Programski kôd može se pisati u bilo kojem editoru teksta, čak i u onim najjednostavnijim. Kada dođe do prevođenja, potpuno je nebitno u kakvom editoru je programski kôd unet. S druge strane, međutim, neke funkcionalnosti editora mogu programeru olakšati unos programa i čitav proces programiranja. Primer takve funkcionalnosti je naglašavanje sintakse (engl. syntax highlighting), kojim se različitim bojama označavaju različite jezičke klase u programu. I neki sasvim jednostavni editori podržavaju ovu funkcionalnost, dok najmoćniji editori mogu da pružaju i mnogo više. Postoje i alati koji, pored moćnog editora, objedinjuju i mnoge dodatne alatke koje čine proces razvoja softvera efikasnijim. Te alate zovemo *integrisana razvojna okruženja* (engl. integrated development environment, IDE). Razvojna okruženja su glavni alat za većinu programera. Osnovna svojstva svakog razvojnog okruženja su: integrisani editor teksta, podrška za olakšano kreiranje izvršivih programa i integrisani debager. Većina razvojnih okruženja ima dodatne, uobičajene funkcionalnosti integrisanih razvojnih okruženja.
U nastavku su nabrojana neka uobičajena svojstva razvojnih okruženja:

- udoban grafički korisnički interfejs: umesto kucanja instrukcija kojima se pokreću kompilator i druge akcije, programer koristi lakšu komunikaciju zasnovanu na prozorima, ikonicama i menijima i na upotrebi miša.
- moćan tekstualni editor: u razvojnom okruženju, editor teksta je alatka u kojoj programer unosi i modifikuje tekst programa i koja može da ima mnoštvo dodatnih svojstava:
  - automatsko isticanje teksta: editor može poznavati sintaksička pravila za mnoge programske jezike, te može automatski modifikovati izgled (ne i sadržaj) teksta tako što će neke reči naglasiti – prikazati u specifičnoj boji, podebljanim ili kurzivnim fontom. Ovakvo isticanje teksta programski kôd čini znatno čitljivijim i u njemu se lakše otkrivaju sintaksičke greške.

  - automatsko formatiranje koda: editor je u stanju da modifikuje kôd programa i održava ga tako da je formatiran u skladu sa nekim konkretnim pravilima i preporukama (na primer, o nazubljivanju, o poziciji vitičastih zagrada, o razmacima oko operatora, itd.).
  - inteligentno upotpunjavanje koda: na osnovu znanja o konkretnom programskom jeziku i konkretnom programu koji se razvija, editor može biti u stanju da predloži dopunu za jezičku konstrukciju čiji je unos započet. Na primer, ako programer počne da unosi reč `double`, posle nekoliko slova biće mu predložen nastavak koji upotpunjuje ovu reč. Dodatno, ukoliko programer unese ime neke promenljive `x` tipa, na primer, `vector`, čim otkuca `x.`, biće mu ponuđen spisak metoda koje se mogu primeniti na objekte tog tipa.

- podrška za refaktorisanje koda: refaktorisanje koda je proces unapređivanja njegovog kvaliteta (u smislu modularnosti, konciznosti, čitljivosti, lakoće održavanja), bez menjanja njegove funkcionalnosti. Razvojna okruženja mogu automatski refaktorisati kôd u nekoj meri, u skladu sa nekim poznatim shemama. Na primer, kôd koji se ponavlja na više mesta može automatski biti izdvojen u novu funkciju ili, suprotno, kôd neke funkcije može biti inlajnovan na mesto njenog pozivanja.

- integrisan kompilator: razvojno okruženje može da uključuje kompilatore za pojedine jezike ili da koristi kompilatore koji su raspoloživi na računaru koji se koristi. Kompilatoru se na udoban način, kroz grafički korisnički interfejs, mogu zadavati opcije (koje se inače zadaju kroz komandnu liniju).

- podrška za razvoj programa koji se sastoje od više (potencijalno mnogo) datoteka: razvojno okruženje omogućava kreiranje *projekata* koji se sastoje iz više izvornih, programskih datoteka ili i drugih vrsta datoteka i omogućava jednostavno dodavanje ili brisanje delova projekta. Okruženje prati stanje datoteka koje čine projekat i, na primer, kada se vrši kompilacija, kompiliraju se samo one jedinice koje su se promenile od prethodne kompilacije.

- integrisan debager: debager je alat koji olakšava detektovanje, lociranje i ispravljanje grešaka (bagova, engl. bug) u drugom programu. On omogućava programeru da kontrolisano izvršava program, tj. da ide korak po korak kroz izvršavanje programa, zaustavi se na označenim mestima (engl. breakpoints), prati vrednosti promenljivih, stanje programskog steka, sadržaj memorije i druge elemente programa.

- podrška za automatsko testiranje: razvojna okruženja omogućavaju automatsko izvršavanje skupina testova kako bi se pojačalo uverenje o ispravnosti koda pre integrisanja u neku širu celinu. Programer može da zada skup testova, način pozivanja programa i slično i, nakon izvršavanja testova, dobija pregled rezultata

testova, na primer, konzolnih izlaza za svaki test. Obično su omogućene raznovrsne obrade takvih rezultata, kako bi se lako otkrili testovi koji nisu uspešno prošli.

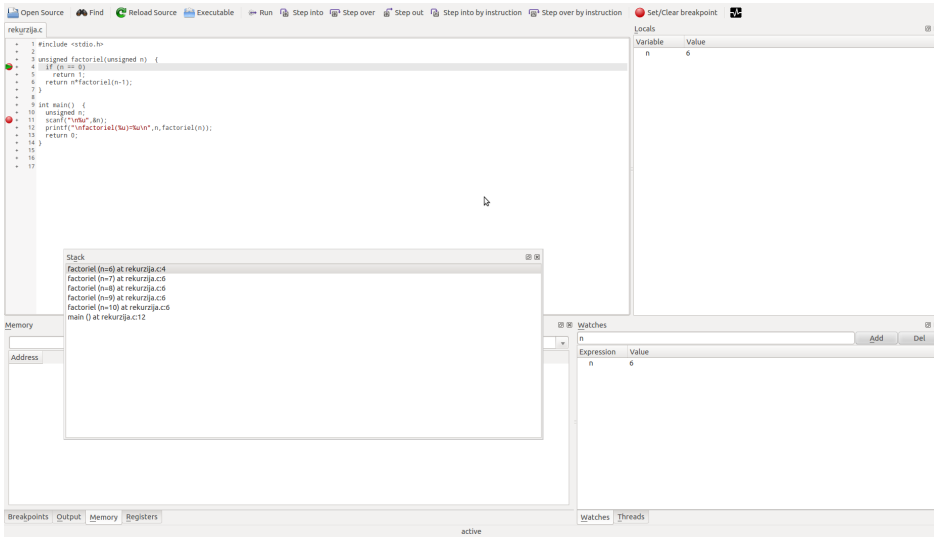
- integrisan profajler: profajliranje je dinamička analiza programa, tj. analiza programa tokom njegovog izvršavanja, kojom se procenjuje vreme izvršavanja programa i njegovih delova, broj pozivanja nekih funkcija, upotreba memorije, itd. Ovakve analize omogućavaju fokusiranje programera na kritične delove koda i popravljanje efikasnosti programa. Razvojna okruženja obično uključuju neki profajler koji se sam sastoji od niza pojedinačnih alatki.

- veza sa sistemom za upravljanje verzijama: *sistemi za upravljanje verzijama* (sistemi za kontrolu verzija, engl. version control systems) olakšavaju timski rad na projektima koji se sastoje od mnoštva datoteka. Članovi tima redovno preuzimaju tekuću zvaničnu verziju projekta, vrše izmene lokalno na svom računaru i, kada su izmene gotove, šalju ih u zajedničku verziju kako bi mogli da ih preuzmu i svi drugi članovi tima.

Postoji mnoštvo raspoloživih razvojnih okruženja za sve računarske platforme, uključujući mnoštvo besplatnih.
Razvojna okruženja mogu biti lokalna ili dostupna putem interneta. U prvom slučaju, okruženje se instalira na lokalni računar, zajedno sa pratećim alatkama, i onda se može koristiti i bez veze sa internetom. Okruženja dostupna putem interneta mogu da ne zahtevaju nikakve promene na lokalnoj mašini, te na njoj ne zauzimaju prostor niti zahtevaju trud za postavljanje sistema. Dodatna pogodnost ovakvih sistema je da mogu da se koriste na različitim platformama.
Razvojna okruženja su osnovni i ključni alat u radu skoro svakog programera. Međutim, početnicima u programiranju savetuje se da najpre ovladaju procesom pisanja programa u svedenom okruženju, tj. da koriste jednostavan editor i kompilaciju iz komandne linije, kako bi razumeli komponente i faze u tom poslu.

11.2 Pregled procesa debagovanja

Pojednostavljeno rečeno, testiranje je proces proveravanja ispravnosti programa, sistematičan pokušaj da se u programu (za koji se pretpostavlja da je ispravan) pronađe greška. S druge strane, debagovanje se primenjuje kada se zna da program ima grešku. Debager je alat za praćenje izvršavanja programa radi otkrivanja konkretne greške (baga, engl. bug). To je program napravljen da olakša detektovanje, lociranje i ispravljanje grešaka u drugom programu. On omogućava programeru da ide korak po korak kroz izvršavanje programa, prati vrednosti promenljivih, stanje programskog steka, sadržaj memorije i druge elemente programa.



Slika 11.1: Ilustracija rada debagera `kdbg`

Slika 11.1 ilustruje rad debagera `kdbg`, koji nudi grafički korisnički interfejs za rad sa debagerom `gdb`, koji se često koristi za razvoj programa u GNU/Linux okruženju. Uvidom u prikazane podatke, programer može da uoči traženu grešku u programu. Da bi se program debagovao, potrebno je da bude preveden za *debag* režim izvršavanja. Za to se u kompilatoru `g++` koristi opcija `-g`. Ako je izvršivi program `mojprogram` dobijen na taj način, može se debagovati navođenjem naredbe:

```
kdbg mojprogram
```

Debageri su danas uglavnom tesno integrisani sa okruženjima za razvoj programa (na primer, okruženje Visual Studio, ali i editor Visual Studio Code pružaju veoma udobnu podršku za debagovanje C++ programa).

11.3 Testiranje

Testiranje je najznačajnija vrsta dinamičkog ispitivanja ispravnosti programa (ispitivanja tokom njegovog rada). Testiranje može da obezbedi visok stepen pouzdanosti programa. Neka tvrđenja o programu je moguće testirati, dok neka nije. Na primer, tvrđenje „program ima prosečno vreme izvršavanja 0,5 sekundi“ je (u principu) proverivo testovima, pa čak i tvrđenje „prosečno vreme između dva pada programa je najmanje 8 sati sa verovatnoćom 95%“. Međutim, tvrđenje „prosečno vreme izvršavanja programa je dobro“ suviše je neodređeno da bi moglo da bude testirano. Primetimo da je, na

primer, tvrđenje „prosečno vreme između dva pada programa je najmanje 8 godina sa verovatnoćom 95%“ u principu proverivo testovima, ali nije praktično izvodivo.
U idealnom slučaju, treba sprovesti iscrpno testiranje rada programa za sve moguće ulazne vrednosti i proveriti da li izlazne vrednosti zadovoljavaju specifikaciju. Međutim, ovakav iscrpan pristup testiranju skoro nikada nije praktično primenljiv. Na primer, iscrpno testiranje korektnosti programa koji sabira dva 32-bitna broja zahtevalo bi ukupno $2^{32} \cdot 2^{32} = 2^{64}$ različitih testova. Pod pretpostavkom da svaki test traje jednu nanosekundu, iscrpno testiranje bi zahtevalo približno $1{,}8 \cdot 10^{10}$ sekundi, što je oko 585 godina. Dakle, testiranjem nije praktično moguće dokazati ispravnost netrivijalnih programa. S druge strane, testiranjem je moguće dokazati da program nije ispravan, tj. pronaći greške u programima.
S obzirom na to da iscrpno testiranje nije praktično primenljivo, obično se koristi tehnika testiranja tipičnih ulaza programa, kao i specijalnih, karakterističnih ulaznih vrednosti za koje postoji veća verovatnoća da dovedu do neke greške. U slučaju pomenutog programa za sabiranje, tipični slučaj bi se odnosio na testiranje korektnosti sabiranja nekoliko slučajno odabranih parova brojeva, dok bi za specijalne slučajeve mogli biti proglašeni slučajevi kada je neki od sabiraka 0, 1, -1, najmanji negativan broj, najveći pozitivan broj i slično.
Postoje različite metode testiranja. Više reči o njima biće u sledećem tomu ove knjige.

12. Projektni zadaci

12.1 Prelazi između slika

Obrada slika široko se koristi, kako od strane korisnika za lične potrebe, tako i za svrhe objavljivanja na internetu, u štampanim izdanjima itd. Postoje mnogi programi (na primer, Photoshop i GIMP) koji nude mnoštvo mogućih obrada slika, uključujući promenu dimenzije, zatamnjivanje, izoštravanje i slično. Standardna biblioteka jezika C++ ne sadrži funkcije za rad sa slikama, te je za takve svrhe potrebno koristiti neku dodatnu biblioteku. Jedna takva je popularna biblioteka `OpenCV`.

`Mat` je klasa definisana u okviru biblioteke `OpenCV` i ona sadrži informacije o formatu slike, o njenim dimenzijama, kao i matricu piksela koji čine sliku. Nećemo ulaziti u detalje opisa ove klase, već ćemo ukratko opisati samo one funkcije i metode koje su nam potrebne. Funkcija `imread` učitava sliku iz datoteke zadatog imena u objekat tipa `Mat`. Slika se podrazumevano učitava u boji i interno se čuva u vidu tri *kanala* – po jedan za plavu, zelenu i crvenu boju (biblioteka `OpenCV` koristi redosled BGR). Učitavanje u boji može se zahtevati i eksplicitno, parametrom `IMREAD_COLOR`, kao što će biti urađeno u narednom projektu. Metodom `empty` proveravamo da li je slika uspešno učitana i, ako nije, program završava rad. Slika se prikazuje primenom funkcije `imshow` (iz biblioteke `OpenCV`): njeni parametri su ime prozora u kom se slika prikazuje i sama slika u obliku `Mat` objekta.

Više informacija o ovoj biblioteci, uključujući uputstva za njeno instaliranje, može se naći na adresi `https://docs.opencv.org`. Pod sistemom Linux, biblioteka se može instalirati na sledeći način:

```
sudo apt update
sudo apt install libopencv-dev
```

Naredni program naizmenično prikazuje tri zadate slike, sa prelaznim stanjima. Program pretpostavlja da su sve tri slike istih dimenzija (širine i visine): pikseli se čitaju sa istih pozicija u obe slike između kojih se pravi prelaz, pa bi za slike različitih dimenzija došlo do pristupa van granica slike, tj. do nedefinisanog ponašanja.

```cpp
#include "opencv2/opencv.hpp"
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>

int main() {
  // spisak slika koje se prikazuju
  std::vector<std::string> slike
    {"beograd1.jpg", "beograd2.jpg", "beograd3.jpg"};

  // trenutna slika
  int brojSlike = 0;

  // stara i nova slika izmedju kojih se pravi prelaz
  cv::Mat stara, nova;

  // ucitavamo prvu sliku
  stara = cv::imread(slike[brojSlike]);
  if (stara.empty()) {
    std::cerr << "Greska pri ucitavanju: "
              << slike[brojSlike] << std::endl;
    return 1;
  }

  // ucitavamo drugu sliku
  brojSlike = (brojSlike + 1) % slike.size();
  nova = cv::imread(slike[brojSlike]);
  if (nova.empty()) {
    std::cerr << "Greska pri ucitavanju: "
              << slike[brojSlike] << std::endl;
    return 1;
  }

  // udeo druge slike u prelazu (od 0 do 1)
  double udeo = 0.0;

  while (true) {
    // matrica koja sadrzi jedan po jedan frejm
    cv::Mat frejm(stara.rows, stara.cols, CV_8UC3);
```

```
// frejm gradimo kombinovanjem dve slike
for (int vrsta = 0; vrsta < frejm.rows; vrsta++) {
  for (int kolona = 0; kolona < frejm.cols; kolona++) {
    for (int boja = 0; boja < 3; boja++) {
      cv::Vec3b s = stara.at<cv::Vec3b>(vrsta, kolona);
      cv::Vec3b n = nova.at<cv::Vec3b>(vrsta, kolona);
      double p1 = s[boja] / 255.0;
      double p2 = n[boja] / 255.0;
      double p = p1*(1 - udeo) + p2*udeo;
      frejm.at<cv::Vec3b>(vrsta, kolona)[boja] = 255 * p;
    }
  }
}

// prikazujemo trenutnu sliku
cv::imshow("Prelaz", frejm);

// uvecavamo udeo druge slike u prelazu
udeo += 0.01;

// ako je udeo > 1, u potpunosti je prikazana druga slika
if (udeo > 1) {
  // nov prelaz krece od tekuce slike
  stara = nova;

  // ucitavamo novu sliku
  brojSlike = (brojSlike + 1) % slike.size();
  nova = cv::imread(slike[brojSlike]);
  if (nova.empty()) {
    std::cerr << "Greska pri ucitavanju: "
              << slike[brojSlike] << std::endl;
    return 1;
  }

  // krecemo prelaz iz pocetka
  udeo = 0.0;
}

// prekidamo program kada se pritisne ESC
```

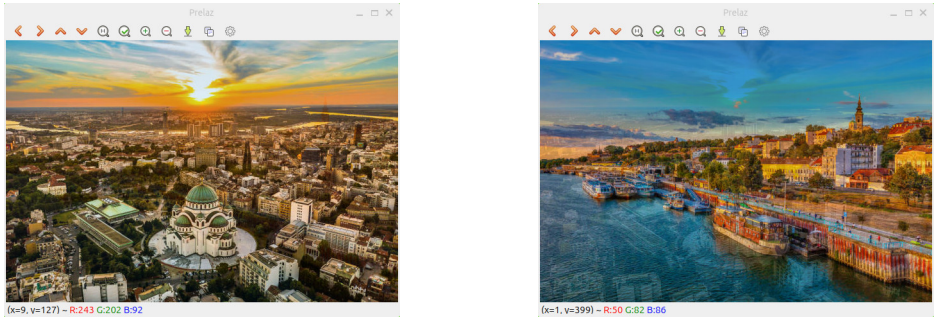
```
    int taster = cv::waitKey(15);
    if (taster == 27)
      break;

  }
  return 0;
}
```

Program se može kompilirati na sledeći način:

```
g++ prelaz.cpp -o prelaz `pkg-config --cflags --libs opencv4`
```

a onda pozvati na sledeći način: `./prelaz`. U tekućem direktorijumu treba da postoje slike čija imena su navedena u okviru funkcije `main` i one moraju biti istih dimenzija.



Slika 12.1: Ilustracija dva stanja tokom rada programa

12.2 Transformacija slika

Projekat koji sledi takođe koristi biblioteku `OpenCV`. U okviru ovog projekta, napravićemo jednostavnu aplikaciju koja omogućava nekoliko obrada slika. Iako i sama biblioteka `OpenCV` pruža podršku za takve obrade, mi ćemo je koristiti samo za učitavanje, prikazivanje i snimanje slika, a same obrade implementiraćemo samostalno.
Naš program počinje pretprocesorskim direktivama – za uključivanje uobičajenog zaglavlja `iostream` (i zaglavlja `string` i `algorithm`), ali i dva potrebna zaglavlja iz biblioteke `OpenCV`, iza kojih slede deklaracije, a onda i definicije funkcija za raznovrsne obrade slika:

```
#include <iostream>
#include <string>
#include <algorithm>
#include "opencv2/imgcodecs/imgcodecs.hpp"
#include <opencv2/highgui.hpp>

using namespace cv;
using namespace std;

void transponuj(Mat& slika);
void rotiraj(Mat& slika, bool uSmeruKazaljke);
void preslikaj(Mat& slika, bool horizontalno);
void pojacajKomponentu(Mat& slika, int boja);
void pikselizuj(Mat& slika, int velicina);
void zamuti(Mat& slika);

void transponuj(Mat& slika) {
  // stara slika ima dimenzije (slika.rows,slika.cols)
  // a nova (slika.cols,slika.rows)
  Mat novaSlika(slika.cols, slika.rows, CV_8UC3);
  for(int y = 0; y < slika.rows; y++) {
    for(int x = 0; x < slika.cols; x++) {
      novaSlika.at<Vec3b>(x, y) = slika.at<Vec3b>(y, x);
    }
  }
  slika = novaSlika;
}

void rotiraj(Mat& slika, bool uSmeruKazaljke) {
  // stara slika ima dimenzije (slika.rows, slika.cols),
```

```
  // a nova (slika.cols, slika.rows)
  Mat novaSlika(slika.cols, slika.rows, CV_8UC3);
  for(int y = 0; y < slika.rows; y++) {
    for(int x = 0; x < slika.cols; x++) {
      // piksel u vrsti y i koloni x stare slike
      Vec3b piksel = slika.at<Vec3b>(y, x);
      if (uSmeruKazaljke)
        novaSlika.at<Vec3b>(x, slika.rows-1-y) = piksel;
      else
        novaSlika.at<Vec3b>(slika.cols-1-x, y) = piksel;
    }
  }
  slika = novaSlika;
}

void preslikaj(Mat& slika, bool horizontalno) {
  // i stara i nova slika imaju dimenzije
  // (slika.rows, slika.cols)
  Mat novaSlika(slika.rows, slika.cols, CV_8UC3);
  for(int y = 0; y < slika.rows; y++) {
    for(int x = 0; x < slika.cols; x++) {
      novaSlika.at<Vec3b>(y, x) =
        horizontalno ?
          slika.at<Vec3b>(slika.rows-1-y, x)
          : slika.at<Vec3b>(y, slika.cols-1-x);
    }
  }
  slika = novaSlika;
}

void pojacajKomponentu(Mat& slika, int boja) {
  // i stara i nova slika imaju dimenzije
  // (slika.rows, slika.cols)
  Mat novaSlika(slika.rows, slika.cols, CV_8UC3);
  for(int y = 0; y < slika.rows; y++) {
    for(int x = 0; x < slika.cols; x++) {
      Vec3b piksel = slika.at<Vec3b>(y, x);
      piksel[boja] = (piksel[boja] + 255)/2;
      novaSlika.at<Vec3b>(y, x) = piksel;
    }
```

```
  }
  slika = novaSlika;
}


void pikselizuj(Mat& slika, int velicina) {
  // rezultujuca slika
  Mat novaSlika(slika.rows, slika.cols, CV_8UC3);

  for (int y = 0; y * velicina < slika.rows; y++)
    for (int x = 0; x * velicina < slika.cols; x++) {
      // blok stare slike koji odgovara jednom "pikselu"
      // nove slike cine vrste [y0, y1) i kolone [x0, x1);
      // uz ivicu slike blok moze biti manji
      int y0 = y * velicina;
      int y1 = min(y0 + velicina, slika.rows);
      int x0 = x * velicina;
      int x1 = min(x0 + velicina, slika.cols);
      // izracunavamo aritmeticku sredinu piksela bloka,
      // pa su nam potrebni zbirovi (posebno za plavu,
      // zelenu i crvenu komponentu) i broj piksela (n)
      int zbirB = 0, zbirG = 0, zbirR = 0, n = 0;
      for (int i = y0; i < y1; i++)
        for (int j = x0; j < x1; j++) {
          Vec3b piksel = slika.at<Vec3b>(i, j);
          zbirB += piksel[0];
          zbirG += piksel[1];
          zbirR += piksel[2];
          n++;
        }
      // ceo prostor novog "piksela" popunjavamo bojom koja
      // odgovara proseku piksela stare slike
      for (int i = y0; i < y1; i++)
        for (int j = x0; j < x1; j++)
          novaSlika.at<Vec3b>(i, j) =
            Vec3b(zbirB / n, zbirG / n, zbirR / n);
    }
  slika = novaSlika;
}
```

```
void zamuti(Mat& slika) {
  // nova slika je u pocetku kopija stare, pa pikseli na
   ↪  ivici
  // (koje ne zamucujemo) zadrzavaju svoje vrednosti
  Mat novaSlika = slika.clone();
  for(int y = 1; y < slika.rows-1; y++) {
    for(int x = 1; x < slika.cols-1; x++) {
      for(int c = 0; c < 3; c++) {
        // tezinsko usrednjavanje: sabiramo 9 piksela iz
        // okoline 3x3, a centralni piksel jos jednom (on
        // ima dvostruku tezinu), pa je zbir svih tezina 10
        // i zbir delimo sa 10
        unsigned boja = slika.at<Vec3b>(y, x)[c];
        for(int i = -1; i < 2; i++) {
          for(int j = -1; j < 2; j++) {
            boja += slika.at<Vec3b>(y+i, x+j)[c];
          }
        }
        novaSlika.at<Vec3b>(y, x)[c] = boja/10;
      }
    }
  }
  slika = novaSlika;
}


int main()
{
  string imeSlike;
  cout << "Unesite ime slike: ";
  cin >> imeSlike;
  // string imeSlike = samples::findFile("beograd2.jpg");

  Mat slika = imread(imeSlike, IMREAD_COLOR);
  if(slika.empty()) {
    cout << "Ne moze se ucitati slika: " << imeSlike << endl;
    return -1;
  }
  imshow(imeSlike, slika);
  int k;
```

```
// cekamo na pritisak tastera,
// sve dok nije pritisnut taster 'esc'
while ((k = waitKey(0)) != 27) {
  switch(k) {
    case 't': // transponovanje
      transponuj(slika);
      break;
    case 'r': // rotiranje u smeru kazaljke na satu
      rotiraj(slika, true);
      break;
    case 'c': // rotiranje suprotno smeru kazaljke na satu
      rotiraj(slika, false);
      break;
    case 'h': // horizontalno preslikavanje
      preslikaj(slika, true);
      break;
    case 'v': // vertikalno preslikavanje
      preslikaj(slika, false);
      break;
    case 'p': // pikselizacija
      pikselizuj(slika,8);
      break;
    case 'b': // zamucivanje
      zamuti(slika);
      break;
    case '0':
    case '1':
    case '2': // pojacaj boju (B='0',G='1',R='2')
      pojacajKomponentu(slika, k - '0');
      break;
    case 's': // snimanje
      imwrite("transformisana_" + imeSlike, slika);
      break;
    default:
      break;
  }
  imshow(imeSlike, slika);
}

return 0;
```

```
}
```

Glavninu funkcije `main` čini jednostavna petlja koja se izvršava sve dok korisnik ne pritisne taster `Esc`. Nekoliko drugih slova pokreće specifične obrade koje su podržane zasebnim funkcijama. Taster koji je pritisnut čita se (bez čekanja na pritisak na `enter`) primenom `OpenCV` funkcije `waitKey(0)` (parametar `0` govori da se na pritisak tastera čeka bez vremenskog ograničenja). Nakon izabrane obrade, iznova se prikazuje slika, sada modifikovana.
Između navedenog početka programa i funkcije `main` nalaze se deklaracije i definicije funkcija koje vrše obrade slike. Razmotrimo detaljnije funkciju `transponuj(Mat& slika)`. U okviru funkcije najpre se kreira nova slika, širine kao visina zadate slike (parametar `slika`), a visine kao njena širina. Argument `CV_8UC3` govori da će se za svaki piksel koristiti tri podatka tipa `unsigned char`, po jedan za svaki od kanala B, G, R (tim redosledom). Pojedinačnom pikselu slike `slika` koji se nalazi u vrsti `y` i koloni `x` može se pristupiti na sledeći način: `slika.at<Vec3b>(y, x)` (biblioteka `OpenCV` uvek navodi prvo vrstu, pa kolonu; `Vec3b` predstavlja `OpenCV` tip vektora koji ima tri elementa od po jedan bajt). Naredbom `novaSlika.at<Vec3b>(x, y) = slika.at<Vec3b>(y, x);` piksel zadate slike iz vrste `y` i kolone `x` kopira se u piksel nove slike u vrsti `x` i koloni `y`, čime se dobija transponovana matrica, pa time i transponovana slika.
Funkcija `zamuti` zamućuje sliku tako što svaki piksel (osim piksela na ivici slike) zamenjuje težinskom sredinom piksela iz njegove okoline veličine 3×3. Centralni piksel pri tome ima dvostruku težinu, a svi ostali jednostruku, pa je zbir težina 10 i zbir se deli sa 10.
Program se može kompilirati na sledeći način:

```
g++ transformacija_slika.cpp -o ts \
    `pkg-config --cflags --libs opencv4`
```

a onda pozvati na sledeći način: `./ts`. Program će tražiti ime slike koju treba transformisati. Ako je to slika `StudentskiTrg.jpg` i ako je primenjeno transponovanje, od polazne slike (levo) biće dobijena nova slika (desno).

Slika 12.2: Polazna slika (levo) i slika dobijena transponovanjem (desno)

12.3 ASCII kamera

Program koji sledi takođe koristi biblioteku `OpenCV`, čita podatke sa kamere (ako postoji) i dobijenu sliku prikazuje koristeći samo ASCII karaktere.

```
#include "opencv2/opencv.hpp"
#include <string>
#include <iostream>

int main() {
  // slike cemo citati sa video-kamere
  cv::VideoCapture kamera(0);

  // provera da li je otvaranje kamere uspelo
  if (!kamera.isOpened()) {
    std::cerr << "Greska: kamera nije otvorena.\n";
    return -1;
  }
```

```
// matrica koja sadrzi jedan po jedan frejm
cv::Mat frejm;
while (true) {
  // ucitavamo frejm sa kamere
  kamera >> frejm;

  // provera da li je ucitavanje uspelo
  if (frejm.empty()) {
    std::cerr << "Greska: ucitan je prazan frejm.\n";
    break;
  }

  // bela boja
  cv::Scalar bela(255, 255, 255);
  // crna boja
  cv::Scalar crna(0, 0, 0);

  // smanjujemo rezoluciju frejma na 20% originalne
  ↪  velicine
  cv::resize(frejm, frejm, cv::Size(), 0.2, 0.2);
  // i prebacujemo frejm u crno-belo
  cv::cvtColor(frejm, frejm, cv::COLOR_BGR2GRAY);

  // karakteri razlicite osvetljenosti
  std::string karakteri = "  _.,`;:!abcW#@@@";

  // frejm sa tekstom koji cemo prikazivati
  cv::Mat tekstFrejm(800, 800, CV_8UC3, crna);

  // font koji se koristi
  int font = cv::FONT_HERSHEY_SIMPLEX;
  // velicina fonta
  double velicinaFonta = 0.25;
  // debljina linija prilikom iscrtavanja slova
  int debljina = 1;

  // merimo velicinu slova A
  // (koristimo je kao velicinu polja za svaki karakter)
  int osnovnaLinija = 0;
  cv::Size velicinaSlova = cv::getTextSize("A", font,
```

```
        velicinaFonta, debljina, &osnovnaLinija);

    // analiziramo svaki piksel smanjene crno-bele slike
    for (int vrsta = 0; vrsta < frejm.rows; vrsta++) {
      for (int kolona = 0; kolona < frejm.cols; kolona++) {
        // osvetljenost piksela
        int osvetljenost = frejm.at<uchar>(vrsta, kolona);
        // pozicija karaktera u nizu (linearna transformacija
        // [0, 255] na [0, karakteri.length() - 1])
        int k = (osvetljenost / 255.0) *
                (karakteri.length() - 1);
        // konvertujemo karakter u string
        std::string tekst(1, karakteri[k]);
        // pozicija na kojoj cemo prikazati karakter
        // (koordinate donjeg levog ugla)
        cv::Point pozicija(
            (frejm.cols - kolona) * velicinaSlova.width,
            (vrsta + 1) * velicinaSlova.height);
        // ispisujemo karakter
        cv::putText(tekstFrejm, tekst, pozicija, font,
            velicinaFonta, bela, debljina);
      }
    }

    // prikazujemo ASCII sliku
    cv::imshow("Kamera", tekstFrejm);

    // prekidamo program kada se pritisne ESC
    int taster = cv::waitKey(30);
    if (taster == 27)
      break;
    else if (taster == 's') {
      // ispisujemo ASCII sliku i na standardni izlaz
      for (int vrsta = 0; vrsta < frejm.rows; vrsta++) {
        for (int kolona = 0; kolona < frejm.cols; kolona++) {
          // osvetljenost piksela
          int osvetljenost = frejm.at<uchar>(vrsta, kolona);
          // pozicija karaktera u nizu
          int k = (osvetljenost / 255.0) *
                  (karakteri.length() - 1);
```

```
            // ispisujemo karakter
            std::cout << karakteri[k];
          }
          std::cout << std::endl;
        }
      }
    }

    // oslobadjamo kameru
    kamera.release();
    // zatvaramo sve prozore
    cv::destroyAllWindows();

    return 0;
}
```

12.4 Mastermajnd

U ovom projektu program pogađa broj koji je zamislio igrač, u skladu sa pravilima igre mastermajnd.

Igrač zamišlja četvorocifreni broj čije su sve cifre različite (dozvoljena je i vodeća nula, na primer, `0918`). Računar zatim ispisuje jedan po jedan pokušaj, a igrač svaki pokušaj ocenjuje unošenjem dva broja: broja cifara koje su pogođene i nalaze se na pravom mestu i ukupnog broja pogođenih cifara (bez obzira na to da li su na pravom mestu ili ne). Na primer, ako je zamišljen broj `1234`, a računar pokuša sa `1325`, igrač unosi `1 3` (cifra 1 je na pravom mestu, a pogođene su cifre 1, 2 i 3). Igra se završava kada igrač unese ocenu `4 4`.

Računar u svakom koraku traži narednu varijaciju (bez ponavljanja cifara) koja je u skladu sa ocenama svih prethodnih pokušaja, tj. koja bi, da je ona zamišljeni broj, dobila upravo te ocene.

```
#include <iostream>
#include <vector>
#include <thread>
#include <cstdlib>
#include <ctime>

using namespace std;
```

```
typedef vector<unsigned> Varijacija;

bool postoje_iste_cifre(const Varijacija& varijacija)
{
  for (int i = 0; i < varijacija.size(); i++)
    for (int j = i + 1; j < varijacija.size(); j++)
      if (varijacija[i] == varijacija[j])
        return true;
  return false;
}

void sledeca_varijacija(Varijacija& varijacija, int n)
{
  int k = varijacija.size();
  int i;
  for (i = k-1; i >= 0 && varijacija[i] == n-1; i--)
    varijacija[i] = 0;
  if (i >= 0)
    varijacija[i]++;
}

void sledeca_varijacija_bez_ponavljanja(Varijacija&
 ↪  varijacija, int n)
{
  do {
    sledeca_varijacija(varijacija, n);
  } while (postoje_iste_cifre(varijacija));
}

void oceni_pokusaj(const Varijacija& resenje, const
 ↪  Varijacija& pokusaj,
                   int& pogodjenih_na_mestu, int&
                    ↪  pogodjenih_ukupno) {
  // assert(resenje.size() == pokusaj.size());
  int k = resenje.size();

  // brojimo koliko ima pogodjenih na pravom mestu
  pogodjenih_na_mestu = 0;
  for (int i = 0; i < k; i++)
    if (resenje[i] == pokusaj[i])
```

```
      pogodjenih_na_mestu++;

  // brojimo koliko ima pogodjenih ukupno
  pogodjenih_ukupno = 0;
  for (int i = 0; i < k; i++)
    for (int j = 0; j < k; j++)
      if (resenje[i] == pokusaj[j])
        pogodjenih_ukupno++;
}

// raniji pokusaj pogotka koji je korisnik ocenio
struct Pokusaj {
  Varijacija varijacija;
  int pogodjenih_na_mestu;
  int pogodjenih_ukupno;

  Pokusaj(const Varijacija& v, int na_mestu, int ukupno) {
    varijacija = v;
    pogodjenih_na_mestu = na_mestu;
    pogodjenih_ukupno = ukupno;
  }
};

// provera da li je trenutna varijacija u skladu sa svim
 ↪  prethodnim pokusajima
// tj. da li je moguce da je korisnik dao ocene koje je dao
// ako bi ona bila tacno resenje
bool zadovoljava_sve_prethodne_pokusaje(
        const Varijacija& varijacija,
        const vector<Pokusaj>& prethodni_pokusaji) {
  for (const Pokusaj& pokusaj : prethodni_pokusaji) {
    int pogodjenih_na_mestu, pogodjenih_ukupno;
    oceni_pokusaj(varijacija, pokusaj.varijacija,
                  pogodjenih_na_mestu, pogodjenih_ukupno);
    if (pogodjenih_ukupno != pokusaj.pogodjenih_ukupno ||
        pogodjenih_na_mestu != pokusaj.pogodjenih_na_mestu)
      return false;
  }
  return true;
}
```

```
void ispisi_varijaciju(const Varijacija& varijacija) {
  this_thread::sleep_for(1ms);
  for (int i = 0; i <= varijacija.size(); i++)
    cout << "\b";
  for (int x : varijacija)
    cout << x;
  cout << " " << flush;
}

int main()
{
  srand(time(nullptr));

  cout << "Zamislite cetvorocifreni broj cije su cifre
 ↪  razlicite "
          "(dozvoljena je i vodeca nula), a ja cu pokusati da
           ↪  ga pogodim."
       << endl;
  cout << "Ocenite svaki pokusaj koji cu napraviti (prvo
 ↪  unesite broj "
          "pogodaka na mestu, pa ukupan broj pogodaka)" <<
           ↪  endl;

  // racunar nasumicno odredjuje pocetnu varijaciju
  Varijacija varijacija = {0, 1, 2, 3};
  int broj_varijacija = rand() % 1000;
  for (int i = 0; i < broj_varijacija; i++) {
    sledeca_varijacija_bez_ponavljanja(varijacija, 10);
    ispisi_varijaciju(varijacija);
  }

  // svi pokusaji koje je racunar do sada napravio (i ocene
   ↪  koje je dobio)
  vector<Pokusaj> prethodni_pokusaji;

  // racunar pokusava sve dok ne pogodi
  while(true) {
    // korisnik ocenjuje trenutnu varijaciju
    int pogodjenih_na_mestu, pogodjenih_ukupno;
```

```
    cin >> pogodjenih_na_mestu >> pogodjenih_ukupno;

    // da li je racunar pogodio resenje
    if (pogodjenih_na_mestu == varijacija.size() &&
        pogodjenih_ukupno == varijacija.size()) {
      cout << "Pogodak!" << endl;
      break;
    }

    // nije jos - pamtimo prethodni pokusaj racunara
    // i ocene korisnika za taj pokusaj
    prethodni_pokusaji.push_back(Pokusaj(varijacija,
↪   pogodjenih_na_mestu,
                                         pogodjenih_ukupno));

    // racunar trazi sledecu varijaciju koja je u skladu
    // sa ocenama svih prethodnih pokusaja
    do {
      sledeca_varijacija_bez_ponavljanja(varijacija, 10);
      ispisi_varijaciju(varijacija);
    } while (!zadovoljava_sve_prethodne_pokusaje(
                varijacija, prethodni_pokusaji));
  }

  return 0;
}
```

Primetimo da program pretpostavlja da igrač ispravno ocenjuje pokušaje. Ako ocene nisu međusobno saglasne (tj. ako ne postoji broj koji bi dobio sve zadate ocene), program nijednu varijaciju neće prihvatiti i beskonačno će ih nabrajati.

13. Dodatak

13.1 Statistike

Bitne statistike serija elemenata su minimum, maksimum, zbir, prosek (aritmetička sredina), proizvod i slično. Za serije elemenata koje su smeštene u neke sekvencijalne kolekcije, ove statistike se mogu izračunavati bibliotečkim funkcijama.
Iterator koji ukazuje na minimalni element u seriji dobija se funkcijom `min_element`, a iterator koji ukazuje na maksimalni element dobija se funkcijom `max_element`.

```
vector<int> a = {3, 8, 4, 1, 9, 6, 2, 7, 5};
cout << "Najmanji element: "
     << *min_element(begin(a), end(a)) << endl;
cout << "Najveci element: "
     << *max_element(begin(a), end(a)) << endl;
```

Prilikom određivanja minimalnog i maksimalnog elementa koristi se podrazumevani poredak elemenata datog tipa (elementi se porede relacijom kojoj odgovara operator <). Na primer, ako se obrađuje sekvencijalna kolekcija (npr. vektor) elemenata tipa `string`, pronalazi se element koji je prvi (za `min_element`) ili poslednji (za `max_element`) u leksikografskom poretku (koji je proširenje relacije < nad karakterima). Ako želimo da koristimo neki drugi poredak, možemo kao treći argument navesti funkciju poređenja. Na primer, naredni kôd pronalazi najkraću nisku (i ispisuje `nar`).

```
vector<string> voce = {"jabuka", "pomorandza", "nar",
 ↪  "kajsija"};
auto poredi_duzinu = [](const string& a, const string& b) {
   return a.length() < b.length();
};
```

```
cout << *min_element(begin(voce), end(voce), poredi_duzinu);
```

Ne postoji funkcija sum koja izračunava zbir elemenata niza. Za izračunavanje zbira može se koristiti funkcija accumulate ili funkcija reduce (obe su deklarisane u zaglavlju <numeric>).

```
int a[] = {8, 3, 4, 5, 2, 6};
int zbir = accumulate(begin(a), end(a), 0);
cout << "Zbir elemenata niza je: " << zbir << endl;
```

ili

```
int a[] = {8, 3, 4, 5, 2, 6};
int zbir = reduce(begin(a), end(a));
cout << "Zbir elemenata niza je: " << zbir << endl;
```

Primetimo da funkcija accumulate ima i treći parametar, koji određuje inicijalnu vrednost zbira, ali i tip rezultata. Funkcija accumulate se uvek izračunava sleva nadesno, dok se reduce može izvršavati u proizvoljnom redosledu, što omogućava i paralelizaciju (na primer, mogućnost da svako od 4 jezgra procesora izračunava zbir jedne četvrtine elemenata niza).
Izračunavanje proseka se svodi na izračunavanje zbira i zatim deljenje brojem elemenata niza (primetimo da smo zbir računali kao podatak tipa double, što je određeno pre svega inicijalnom vrednošću 0.0, a zatim i tipom promenljive u kojoj pamtimo zbir).

```
vector<int> a = {8, 3, 4, 5, 2, 6};
double zbir = accumulate(begin(a), end(a), 0.0);
double prosek = zbir / a.size();
```

Proizvod elemenata serije se takođe može izračunati funkcijom accumulate.

```
double a[] = {8, 3, 4, 5, 2, 6};
double proizvod = accumulate(begin(a), end(a), 1.0,
 ↪  multiplies<double>());
```

Inicijalna vrednost proizvoda je 1.0 (ovde se radi o podacima tipa double). Četvrti parametar je funkcija koja se primenjuje na objedinjavanje tekućeg rezultata i tekućeg elementa kolekcije. Kada se taj argument ne navede, podrazumeva se funkcija

plus, koja sabira tekući zbir i tekući element kolekcije. Za množenje je upotrebljena funkcija multiplies, koja množi te dve vrednosti (ona je deklarisana u zaglavlju <functional>). Naravno, tu funkciju je moguće i definisati samostalno (na primer, kao anonimnu funkciju), što ima smisla kada ne postoji odgovarajuća bibliotečka funkcija.

```
double a[] = {8, 3, 4, 5, 2, 6};
double proizvod = accumulate(begin(a), end(a), 1.0,
                      [](double x, double y) { return x * y;
                       ↪  });
```

13.2 Menjanje redosleda elemenata niza

Često je potrebno promeniti redosled elemenata neke kolekcije.
Obrtanje redosleda elemenata niza se vrši bibliotečkom funkcijom reverse.

```
string s = "Ana voli milovana";
reverse(begin(s), end(s));
cout << s << endl; // ispisuje: anavolim ilov anA
```

Još jedna često korišćena transformacija kolekcije je rotacija, koja se može ostvariti funkcijom rotate. Prvi i treći parametar su iteratori koji ograničavaju deo kolekcije koji se rotira, a drugi parametar je iterator koji ukazuje na element koji će postati početni nakon rotacije.

```
string s = "zdravo svima";
rotate(begin(s), next(begin(s), 3), end(s));
cout << s << endl; // ispisuje: avo svimazdr
```

Funkcija shuffle nasumično permutuje elemente kolekcije. Kao treći argument joj se prosleđuje generator pseudoslučajnih brojeva, za čije je korišćenje potrebno uključiti zaglavlje <random>.

```
vector<int> a = {1, 2, 3, 4, 5};
random_device rd;          // izvor slucajnosti (seme)
mt19937 generator(rd());   // generator pseudoslucajnih
 ↪  brojeva
```

```
shuffle(begin(a), end(a), generator);
```

Nakon izvršavanja prethodnog koda, niz a će sadržati elemente od 1 do 5, ali će njihov redosled biti izmenjen (na primer, niz može da sadrži redom elemente 5, 2, 1, 4, 3). Starije verzije jezika sadržale su i funkciju `random_shuffle`, koja je uklonjena iz standarda C++17.

Funkcija `next_permutation` pronalazi narednu permutaciju u leksikografskom redosledu. Funkcija vraća `true` ako postoji naredna permutacija. Naredni kôd ispisuje sve permutacije elemenata od 1 do 5.

```
vector<int> a = {1, 2, 3, 4, 5};
do {
   // ispisujemo elemente niza
   for (int x : a)
      cout << x;
   cout << endl;
} while(next_permutation(begin(a), end(a)));
```