Deo I

C++ – Pregled

Postoje dva glavna aspekta programa koji pišemo:

1. Kolekcija *algoritama* (to jest, programiranih instrukcija za rešavanje određenog zadatka);
2. Kolekcija *podataka* na osnovu kojih se izvršavaju algoritmi da bi se dobila sva pojedinačna rešenja.

U kratkoj istoriji računarstva ova dva glavna aspekta programa, algoritmi i podaci, nisu se menjala. Razvijao se samo njihov odnos. Ovaj odnos se naziva *paradigma programiranja* (engl. *programming paradigm).*

U proceduralnoj paradigmi programiranja skup algoritama direktno oblikuje problem. Sistem uzimanja i vraćanja materijala koji se mogu pozajmljivati u javnoj biblioteci, kao što su knjige, video kasete i slično predstavlja niz procedura, među kojima se kao glavne ističu uzimanje i vraćanje materijala iz biblioteke. Podaci su pohranjeni zasebno i pristupa im se ili sa globalne lokacije ili tako što prolaze kroz procedure. Tri vodeća proceduralna jezika su FORTRAN, C i Pascal. Jezik C++ takođe podržava proceduralno programiranje. Pojedinačne procedure, kao što su `check_in()`, `check_out()`, `overdue()`, `fine()` i tako dalje nazivaju se funkcije. Deo III, Programiranje zasnovano na procedurama, stavlja akcenat na podršku koju jezik C++ pruža za proceduralnu paradigmu programiranja, a posebno ističe funkcije, šablon funkcije i generičke algoritme.

70-tih godina, pažnja pri projektovanju programa pomerena je sa proceduralne paradigme na *apstraktne tipove podataka (engl. abstract data types – ADT* (sada se to uglavnom naziva programiranje *zasnovano na objektima (*engl. *object-based*)). U ovoj paradigmi, problem je oblikovan pomoću skupa apstraktnih podataka. U jeziku C++ te apstraktne podatke nazivamo *klasama* (engl. *classes*). Sistem vraćanja u biblioteci, na primer, prema ovoj paradigmi predstavljen je kao interakcija između objekata primeraka klasa kao što su knjiga, član, datum isticanja (aspekt vremena) i neizbežna vrednost (aspekt novca) koji predstavljaju apstrakcije biblioteke. Algoritmi koji se odnose na svaku klasu nazivaju se *javni interfejs* klase (engl. *public interface*). Podaci se privatno pohranjuju u svakom objektu; pristup podacima je skriven od osnovnog

programa. Apstraktne tipove podataka podržavaju tri programska jezika CLU, Ada i Modula-2. Deo IV, „Programiranje zasnovano na objektima" govori o podršci koju C++ pruža kod paradigme programiranja sa apstraktnim tipovima podataka i ilustruje tu podršku.

Objektno orijentisano programiranje proširuje apstraktne tipove podataka mehanizmima *nasleđivanja* (engl. *inheritance*) (ponovno korišćenje postojeće implementacije) i *dinamičkog povezivanja* (engl. *dynamic binding*) (ponovno korišćenje postojećeg javnog interfejsa). Sada postoje specijalni tip/podtip odnosi između prvobitno nezavisnih tipova. Knjiga, video-kaseta, snimak i dečija igračka su svaki posebno *vrsta* materijala u biblioteci, iako svaki ima svoja pravila uzimanja i vraćanja. Deljeni javni interfejs i privatni podaci smešteni su u apstraktnu klasu bibliotečki materijal (engl. *LibraryMaterial*). Svaka posebna klasa bibliotečkih materijala *nasleđuje* deljeno ponašanje od apstraktne klase bibliotečki materijal i treba da obezbedi samo algoritme i podatke koji podržavaju njeno ponašanje. Tri vodeća jezika koja podržavaju objektno orijentisanu paradigmu su Simula, Smalltalk i Java. U delu V, „Objektno orijentisano programiranje", posebno se govori o podršci koju obezbeđuje jezik C++ za objektno orijentisanu paradigmu programiranja.

C++ je jezik više paradigmi. Iako ga u osnovi smatramo za objektno orijentisani jezik, on pruža podršku za proceduralno programiranje i programiranje zasnovano na objektima. Prednost je u tome da možemo da obezbedimo rešenje koje najbolje odgovara problemu – u praksi nijedna paradigma ne predstavlja najbolje rešenje za sve probleme. Nedostatak je što usled toga ovaj jezik postaje veći i složeniji.

U delu I predstavljamo kratak pregled jezika C++. Prvi razlog za to je naša želja da obezbedimo početni uvod u svojstva jezika kako bismo mogli da se na aspekte jezika slobodnije pozivamo pre nego što ih potpuno obradimo. Na primer, klasama se ne bavimo detaljno sve do poglavlja 13, ali da smo do tada čekali da pomenemo klase, završili bismo predstavljajući mnogo više nereprezentativnih i uglavnom nevažnih primera programa.

Drugi razlog zbog koga smo pružili širok početni pregled jezika je estetski. Osim ako ne slušate prelepe i složene Betovenove sonate ili ako niste stimulisani regeom Skota Joplina, jednostavno ćete postati prilično nestrpljivi i biće vam dosadno uz očigledno nevažne detalje o povisilicama, snizilicama, oktavama i akordima; ali dok ne savladate ove detalje, komponovanje muzike vam je uglavnom nedostupno. Slično tome stoje stvari i kod programiranja. Prolaženje kroz lavirint prioriteta operatora ili pravila koja vladaju kod standardnih aritmetičkih konverzija je neophodno ali i obavezno dosadna osnova za ovladavanje jezikom C++.

Poglavlje 1 pruža kratak uvod u osnovne elemente jezika: ugrađene tipove podataka, promenljive, izraze, naredbe i funkcije. Ono se bavi najmanjim pravim programom C++, govori o procesu prevođenja programa, kratko prikazuje pretprocesor i prvi put prikazuje podršku za ulaz i izlaz. Ono predstavlja nekoliko jednostavnih ali celovitih programa C++ koji ohrabruju čitaoca da ih kompajlira i izvrši.

U poglavlju 2 predstavljamo proceduralni program, program zasnovan na objektima, a zatim objektno orijentisani program i implementacije niza – to jest, numerisanu kolekciju elemenata istog tipa. Zatim poredimo apstraktni niz sa klasom vector iz Standardne biblioteke jezika C++, a zatim dajemo prvi prikaz generičkih algoritama iz Standardne biblioteke. Usput vas motivišemo i bacamo prvi pogled na podršku jezika C++ za obradu podataka, šablone i prostore imena. Na taj način se uvodi ceo jezik iako su u tekstu mnogi detalji ostavljeni za kasnije.

Neki čitaoci će smatrati da su delovi poglavlja 2 teško čitljivi. Materijal je predstavljen bez potpunih objašnjenja koja se obično očekuju za priručnik (objašnjenje je dato u poglavljima koja slede). Ako ste razočarani ili nestrpljivi zbog nivoa detalja, preporučujemo da brzo pređete ili preskočite taj deo, a da im se kasnije vratite kada vam materijal bude nešto poznatiji. U poglavlju 3 započinjemo tradicionalniji način pisanja, tako da čitaocu koji se nije snašao u poglavlju 2 preporučujemo da krene od njega.

1

Početak

U ovom poglavlju predstavljeni su osnovni elementi jezika: ugrađeni tipovi podataka, definicije imenovanih objekata, izrazi, naredbe, definicije i korišćenje ime, novanih funkcija. Opisan je najmanji pravi program u jeziku C++, ukratko se govori o procesu kompajliranja programa, daje se prikaz pretprocesora kao i prvi opis podrške za ulaz i izlaz. U ovom poglavlju izložili smo i nekoliko jednostavnih ali celovitih programa u jeziku C++.

1.1 Rešavanje problema

Programi se često pišu kao odgovor na neki problem ili zadatak koji treba rešiti. Hajde da pogledamo primer. Knjižara upisuje u dosije naslov i izdavača svake knjige koju proda. Informacije se unose onim redom kojim se knjige prodaju. Svake dve nedelje vlasnik ručno izračunava broj primeraka svakog prodatog naslova i broj prodatih primeraka knjiga svakog izdavača. Spisak je uređen po abecednom poretku prema izdavaču i koristi se radi ponovnog naručivanja. Zamoljeni smo da obezbedimo program koji će obaviti ovaj posao.

Jedan od metoda rešavanja velikog problema je da ga podelimo na nekoliko manjih. Idealan slučaj bi bio da je manje probleme lakše rešiti i da oni objedinjeni rešavaju veliki problem. Ako su manji problemi koji su nastali deljenjem suviše veliki za rešavanje, u sledećem koraku ćemo te probleme razdeliti na još manje, nastavljajući taj proces dok, nadajmo se, ne budemo imali rešenje za svaki poddeo problema. Ova strategija poznata je pod različitim nazivima: *podeli pa vladaj* i *napredovanje u koracima.* Naš problem u knjižari lepo smo podelili na četiri potproblema ili zadatka:

1. Pročitati podatke o prodaji.
2. Izračunati prodaju prema naslovu i izdavaču.
3. Urediti naslove prema izdavaču.
4. Ispisati rezultate.

Stavke 1, 2 i 4 predstavljaju probleme koje umemo da rešimo; njih ne moramo dalje da delimo. Međutim, još uvek ne znamo šta bismo mogli da uradimo sa stavkom 3. Zato ćemo ponovo primeniti naš metod na ovu stavku:

3a. Urediti prodate knjige prema izdavaču.

3b. Za svakog izdavača urediti prodate knjige prema naslovima.

3c. Uporediti susedne naslove u okviru svake grupe izdavača. Za svaki par koji se slaže, povećati broj pojavljivanja prvog i obrisati drugi.

Stavke 3a, 3b, i 3c predstavljaju probleme koje sada umemo da rešimo. Pošto možemo da rešimo sve potprobleme koje smo odredili, mi smo u stvari rešili prvobitni, veći problem. I ne samo to, vidimo da privobitni redosled zadataka nije bio tačan. Redosled potrebnih postupaka je sledeći:

1. Pročitati podatke o prodaji.
2. Urediti podatke o prodaji – najpre prema izdavaču, a zatim prema naslovima istog izdavača.
3. Sažeti duplikate naslova.
4. Ispisati rezultate u novoj datoteci.

Redosled postupaka koji se dobija kao rezultat naziva se *algoritam.* Sledeći korak je kompajliranje našeg algoritma na određeni programski jezik – u ovom slučaju, C++.

1.2 Program u jeziku C++

U jeziku C++ svaki postupak se naziva *izraz* (engl. *expression*). Izraz koji se završava tačkom i zarezom naziva se *naredba* (engl. *statement*). Naredba je najmanja nezavisna celina u programu C++. U prirodnom jeziku, analognu konstrukciju predstavlja rečenica. Na primer, ovo su naredbe u jeziku C++:

```
int book_count = 0;
book_count = books_on_shelf + books+on_order;
count << "the value of book_count: " << book_count;
```

Prva naredba je naredba *deklaracije*. `book_count` se različito naziva – *identifikator* (engl. *identifier*), *simbolička promenljiva* (engl. *symbolic variable*) (ili skraćeno *promenljiva*) ili *objekat.* On definiše memorijsku lokaciju računara kojoj odgovara ime `book_count` i sadrži celobrojnu vrednost. 0 predstavlja *literalnu konstantu* (engl. *literal constant*). `book_count` je *inicijalizovan (engl. initialized)* na početnu vrednost nula.

Druga naredba je naredba *dodele* (engl. *assignment*). Ona smešta na memorijskoj lokaciji računara, koja odgovara imenu `book_count`, rezultat sabiranja `books_on_shelf` i `books_on_order`. Pretpostavka je da su i ovo celobrojne promenljive koje su definisane i kojima su dodeljene vrednosti u prethodnim delovima programa.

Treća naredba je *izlazna* naredba. `cout` je izlazna destinacija koja se odnosi na korisnikov terminal. `<<` je operator izlaza. Naredba se piše u `cout` – to jest, na

korisnikovom terminalu – najpre *literalna niska* (engl. *string literal*) zapisana pod znacima navoda, a zatim vrednost koja je pohranjena na memorijskoj lokaciji računara `book_cont`. Izlazni izveštaj ove naredbe je

```
the value of book_count: 11273
```

pod pretpostavkom da `book_count` u tom trenutku ima vrednost 11273.

Naredbe su logički grupisane u imenovane celine koje se nazivaju *funkcije.* Na primer, sve naredbe koje su neophodne da se datoteka prodaje pročita organizovane su u funkciju koja se zove `readIn()`. Slično tome, organizujemo funkcije `sort()`, `compact()` i `print()`.

U jeziku C++, svaki program mora da sadrži funkciju pod nazivom `main()`, koju obezbeđuje programer, da bi program mogao da se izvrši. Evo kako bi funkcija `main()` mogla da se definiše za prethodni algoritam:

```
int main()
{
  readIn();
  sort();
  compact();
  print();
  return 0;
}
```

C++ program počinje sa izvršavanjem od prve naredbe funkcije `main()`. U ovom slučaju, program počinje tako što izvršava funkciju `readIn()`. Izvršavanje programa nastavlja se tako što sledi izvršavanje naredbi unutar funkcije `main()`. Normalno izvršavanje programa završava se dolaskom do poslednje naredbe funkcije `main()`.

Funkcija se sastoji od četiri dela: tipa rezultata, imena funkcije, liste parametara i tela funkcije. Prva tri dela se zajedno nazivaju *prototip funkcije* (engl. *function prototype*).

Lista parametara zapisana u zagradi sadrži nula ili više parametara koji su razdvojeni zarezima. Telo funkcije nalazi se između dve vitičaste zagrade. Ono se sastoji od niza naredbi programa.

U ovom primeru, telo funkcije `main()` *poziva* (engl. *invokes*) funkcije `readIn()`, `sort()`, `compact()` i `print()`. Kada se one izvrše, izvršava se naredba

```
return 0;
```

Pomoću prethodno definisane naredbe `return` jezika C++, završava se izvršavanje funkcije. Kada se navede vrednost, kao što je 0, ta vrednost postaje *rezultat* (engl. *returned value*) funkcije. U ovom slučaju, rezultat 0 ukazuje da je funkcija `main()` uspešno izvršena. (U Standardnom jeziku C++, podrazumeva se da funkcija `main()` vraća vrednost 0 ako nije eksplicitno upotrebljena povratna naredba.)

A sada da pogledamo kako da program spremite za izvršavanje. Prvo moramo da navedemo definicije za `readIn()`, `sort()`, `compact()` i `print()`. Za potrebe našeg izlaganja dovoljno dobar će biti sledeći jednostavan primer:

```
void readIn(){cout << "readIn()\n";}
void sort()  {cout << "sort()\n";  }
```

```
void compact()  { cout << "compact()\n"; }
void print()    { cout << "print()\n";   }
```

`void` se koristi da odredi funkciju koja nema rezultat. Kako što je definisano, svaka funkcija će jednostavno prijaviti svoje prisustvo na korisnikovom terminalu kada je pozove funkcija `main()`. Kasnije možemo da zamenimo jednostavne primere pravim implementacijama funkcija.

Ovaj postupni metod u projektovanju programa obezbeđuje korisnu mogućnost kontrolisanja grešaka u programu koje se ne mogu izbeći. Program koji bi odmah radio suviše je složen i može da vas zbuni.

Ime izvorne datoteke programa sastoji se uglavnom iz dva dela: imena datoteke – na primer, `bookstore` – i sufiksa datoteke. Sufiks datoteke, prema konvenciji, služi za određivanje sadržaja datoteke. Datoteka

```
bookstore.h
```

prema konvenciji se tumači kao *zaglavlje (engl. header)* u jeziku C i C++. (Međutim, standardna zaglavlja u jeziku C++ nemaju sufiks – ona predstavljaju poslovični izuzetak ovog pravila.)

Datoteka

```
bookstore.c
```

se prema konvenciji tumači kao datoteka sa tekstom programa na jeziku C, dok se u operativnom sistemu UNIX datoteka

```
bookstore.C
```

tumači kao datoteka sa tekstom programa na jeziku C++. Sufiks za C++ programske datoteke razlikuje se u pojedinim implementacijama jezika C++, posebno od kada u DOS-u malo i veliko C ne može da se razlikuje. Druge konvencije sufiksa za razlikovanje datoteka sa tekstom programa na jeziku C++ su

```
bookstore.cxx
bookstore.cpp
```

Slično tome, sufiks za zaglavlja se takođe razlikuje u različitim implementacijama jezika C++ (ovo je jedan od razloga zbog kojih standardna zaglavlja na jeziku C++ nemaju sufiks datoteke). Pročitajte u uputstvu za korišćenje kompajlera koji je odgovarajući sufiks na platformi koju koristite.

Pomoću nekog editora za tekst, upišite sledeći program u izvornu datoteku jezika C++.

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

void read()     { cout << "read()\n";    }
void sort()     { cout << "sort()\n";    }
void compact()  { cout << "compact()\n"; }
void write()    { cout << "write()\n";   }
```

```
int main() {
  readIn();
  sort();
  compact();
  write();

  return 0;
}
```

`iostream` je standardno zaglavlje biblioteke u/i tokova (obratite pažnju da nema sufiks). Ono sadrži informacije o imenu `cout` koje su neophodne za ovaj program. `#include` je *direktiva pretprocesora* (engl. *prepocessor directive*). Ona utiče da se sadržaj zaglavlja `iostream` učita u našu datoteku sa tekstom programa. (Odeljak 1.3 govori o direktivama pretprocesora.)

Imena koja su definisana u Standardnoj biblioteci jezika C++, kao što je ime `cout`, ne mogu se koristiti u programu osim ako ne pratite sledeću pretprocesorsku direktivu

```
#include <iostream>
```

naredbom

```
using namespace std;
```

Ova naredba se zove *direktiva za upotrebu* (engl. *using directive*). Imena u standardnoj biblioteci jezika C++ su deklarisana u prostoru imena pod nazivom prostor imena `std` i ne vide se u datotekama sa tekstom programa, osim ako eksplicitno ne odredimo da budu vidljiva. Direktiva za upotrebu govori kompajleru da koristi imena iz biblioteke deklarisana u prostoru imena `std`. (Više ćemo govoriti o prostorima imena i korišćenju direktiva u odeljcima 2.7 i 8.5.)[1]

Kada se program zapiše u datoteci, recimo, `prog1.C`, sledeći korak je kompajliranje. To ćemo uraditi na sledeći način pod UNIX-ovim operativnim sistemom ($ predstavlja odzivni znak sistema):

```
$ CC prog1.C
```

Ime komande koje se koristi za pozivanje kompajlera na jezik C++ je različito u određenim implementacijama. (U Windowsu se komanda obično poziva kada se mišem odabere stavka iz menija). CC je ime komande za kompajler na jezik C++ koji koristimo kod UNIX-ovih radnih stanica. Proverite uputstvo ili pitajte administratora sistema koje je ime komande za C++ na vašem sistemu.

Deo posla kompajlera je da analizira tačnost teksta programa. Kompajler ne može da otkrije da li je značenje programa ispravno, ali može da otkrije greške u *formi* (engl. *form*) programa. Dve uobičajene vrste grešaka u programu su:

1. Sintaksna greška. Programer je napravio „gramatičku" grešku u jeziku C++. Na primer:

[1] U vreme kada je napisana ova knjiga, nisu sve implementacije jezika C++ podržavale prostore imena. Ako vaša implementacija ne podržava prostore imena, korišćenje direktive se mora izostaviti. Pošto su mnogi primeri u ovoj knjizi prevedeni sa implementacijama koje ne podržavaju prostore imena, u većini primera koda nismo koristili direktive.

```
int main ( { // greška: nedostaje ')'
  readIn():  // greška: neodgovarajući znak ':'
  sort();
  compact();
  print();

  return 0   // greška: nedostaje ';'
}
```

2. Greške u tipu. Svaki podatak u jeziku C++ ima odgovarajući tip. Vrednost 10, na primer, je ceo broj. Reč „zdravo" između znakova navoda predstavlja nisku. Ako je funkciji koja zahteva celobrojni argument data niska, kompajler signalizira grešku u tipu.

Poruka o grešci sadrži broj reda i kratak opis onoga što kompajler smatra da je pogrešno urađeno. Preporučujemo vam da ispravljate greške onim redom kojim su prijavljene. Često jedna greška može imati kaskadni efekat i prouzrokovati da kompajler prijavi više grešaka nego što ih zaista ima. Kada se greška ispravi, program treba ponovo kompajlirati. Ovaj ciklus se često naziva *pisanje-kompajliranje-otklanjanje grešaka* (engl. *edit-compile-debug*).

Drugi deo posla kompajlera je da kompajlira formalno ispravan tekst programa. Ovo kompajliranje koje se naziva *generisanje koda* (engl. *code generation*) obično generiše objektni ili asemblerski tekst instrukcija koje razume računar na kome se izvršava program.

Rezultat uspešnog kompajliranja je izvršna datoteka. Kada se pokrene, naš program generiše sledeći izlazni izveštaj:

```
readIn()
sort()
compact()
print()
```

Jezik C++ definiše ugrađeni skup osnovnih tipova podataka: celobrojne tipove i tipove brojeva u pokretnom zarezu, znakovni tip i Bulov tip koji sadrže vrednosti tačno ili netačno. Svakom tipu odgovara *ključna reč* jezika. Svaki objekat u programu ima određeni tip. Na primer, naredbe:

```
int      age = 10;
double   price = 19.99;
char     delimiter = ' ';
bool     found = false;
```

definišu četiri objekta – `age, price, delimiter` i `found` odnosno – celobrojni, broj u pokretnom zarezu sa dvostrukom tačnošću, znakovni i Bulov tip. Svaki objekat ima navedenu početnu vrednost: ceo broj 10, broj u pokretnom zarezu 19.99, prazninu i Bulovu vrednost `false (netačno).`

Konverzija (engl. *conversions*) tipova između ugrađenih tipova odvija se implicitno. Na primer, kada dodelimo objektu `age`, koji je tipa `int` navedenu konstantnu vrednost tipa `double`, kao u primeru:

```
age = 33.333;
```

vrednost koja je u stvari dodeljena objektu `age` je zaokružena celobrojna vrednost 33. (O ovim *standardnim konverzijama* kao i o konverziji tipova uopšte, govori se detaljnije u odeljku 4.14.)

Prošireni skup osnovnih tipova podataka obezbeđen je u standardnoj biblioteci jezika C++, uključujući, između ostalog, nisku, kompleksni broj, vektor i listu. Na primer:

```
// zaglavlje neophodno za upotrebu objekta tipa string
#include <string>
string current_chapter = "Početak";

// zaglavlje neophodno za upotrebu objekta tipa vector
#include <vector>
vector<string> chapter_titles( 20 );
```

`current_chapter` je objekat tipa string inicijalizovan navedenom niskom „Početak". `chapter_titles` je vektor od 20 elemenata tipa string. Neobična sintaksa

```
vector<string>
```

upućuje kompajler da napravi vektorski tip koji može da sadrži elemente niske. Da bismo definisali objekat tipa vector koji može da sadrži 20 celobrojnih elemenata, treba da napišemo

```
vector<int> ivec( 20 );
```

(U nastavku teksta ćemo o vektorima govoriti opširnije.)

Ni jezik ni Standardna biblioteka ne mogu u praksi da obezbede sve tipove podataka koji su neophodni u programskom okruženju. Moderni jezici često obezbeđuju mogućnost definisanja tipova koje omogućava da uvedemo nove tipove u jezik koji se mogu koristiti manje-više isto kao i ugrađeni tipovi. Odgovarajuće sredstvo u jeziku C++ je mehanizam klase. String, complex, vector i list iz Standardne biblioteke su sve klase koje su programirane u jeziku C++. Tako je, u stvari, takođe i sa bibliotekom iostream.

Mehanizam klasa je možda najvažnija komponenta jezika C++. U poglavlju 2 dajemo detaljan pregled čitavog mehanizma klasa.

1.2.1 Kontrola toka programa

Podrazumeva se da se naredbe izvršavaju redom od prve do poslednje. Na primer, u prethodnom programu, koji je dole ponovljen, funkcija `read()` se uvek izvršava prva, zatim je prate `sort(), compact()`, a potom `write()`.

```
int main()
{
  read();
  sort();
  compact();
  write();

  return 0;
}
```

Međutim, ako je prodaja bilo posebno slaba, kao što je nula ili jedan primerak, teško da ima svrhe da se uređuje ili sažima, iako ipak moramo da napišemo taj jedan podatak ili da ukažemo da nije bilo prodaje. To možemo da uradimo pomoću uslovne naredbe *if* (ovo pretpostavlja da smo napisali da funkcija `read()` vraća broj pročitanih vrednosti):

```
// read() vraća broj pročitanih vrednosti
// tip rezultata je int
int read() { ... }
// ...
int main()
{
  int count = read();
  // ako je pročitano više od jedne stavke
  // onda uređujemo i sažimamo
  if ( count > 1) {
       sort();
       compact();
}
  if( count == 0 )
       count << "ovog meseca nema prodaje\n";
  else write();

  return 0;
}
```

Prva naredba obezbeđuje uslovno izvršavanje koje se zasniva na istinitosnoj vrednosti izraza u zagradi. U ovom revidiranom programu, funkcije `sort()` i `compact()` se pozivaju samo ako `count` ima veću vrednost od 1. U drugoj if naredbi, izvršavanje se grana. Ako je uslov tačan – u ovom slučaju, ako je `count` jednako 0 – jednostavno ćemo napisati da nije bilo prodaje; ili, kad god `count` nije jednako 0, pozivamo funkciju `write()`. O naredbi if detaljnije se govori u odeljku 5.3.

Druga uobičajena nesekvencijalna forma izvršavanja naredbe od prve do poslednje je iterativna, ili naredba *ponavljanja* (engl. *loop*). Njome se ponavlja jedna ili više naredbi dok neki uslov ostaje tačan. Na primer:

```
int main()
{
  int iterations = 0;
  bool continue_loop = true;
```

```
  while ( continue_loop ! = false )
  {
     iterations++;

    count  << "naredba ponavljanja while je izvršena "
      << ponavljanje << " puta\n";
    if ( iterations == 5 )
      continue_loop = false;
  }
return 0;
}
```

U ovom, na neki način neprirodnom primeru, naredba ponavljanja *while* izvršava se pet puta, sve dok `iterations` ne bude jednako 5, a `continue_loop` ne dobije vrednost `false`. Naredba

```
iterations++;
```

povećava `iterations` za 1. Realniji primer naredbe ponavljanja while videćete u odeljku 1.5, a detaljniji prikaz naredbi ponavljanja u poglavlju 5.

1.3 Direktive pretprocesora

Datoteke zaglavlja postale su deo programa pomoću *pretprocesorske direktive za uključivanje* (engl. *preprocessor include directive*). Pretprocesorske direktive su označene znakom (#) koji se nalazi u prvoj koloni reda programa. Program koji upravlja direktivama naziva se *pretprocesor* (on se obično nalazi u sastavu samog kompajlera).

Direktiva `#include` čita sadržaj imenovane datoteke. Ima jednu od ove dve forme:

```
#include <some_file.h>
#include "my_file.h"
```

Ako se ime datoteke nalazi u uglastim zagradama (<,>) smatra se da je datoteka standardno ili projektno zaglavlje. Ona se traži u prethodno definisanom skupu lokacija koji se može modifikovati podešavanjem odgovarajuće promenljive okruženja ili pomoću opcija komandne linije. (Metodi za pretraživanje se veoma razlikuju na različitim platformama, pa vam preporučujemo da pitate kolege ili pročitate uputstvo za korišćenje kompajlera ako želite više informacija.) Ako se ime datoteke nalazi pod znacima navoda, podrazumeva se da datoteka predstavlja zaglavlje koje definiše korisnik. Pretraživanje počinje u direktorijumu u kome je smeštena datoteka u koju se uključuje.

Datoteka za uključivanje može da sadrži direktivu `#include`. Zbog ugneždenih uključenih datoteka, zaglavlje može ponekad da bude uključeno više puta u jednu izvornu datoteku. Uslovne direktive sprečavaju da se više puta obradi isto zaglavlje. Na primer:

```
#ifndef BOOKSTORE_H
#define BOOKSTORE_H
     /* Sadržaj Bookstore.h stoji ovde */
#endif
```

Uslovna direktiva `#ifndef` proverava da li je `BOOKSTORE_H` možda prethodno definisana. `BOOKSTORE_H` je pretprocesorska konstanta. (Prema konvenciji pretprocesorske konstante se pišu velikim slovom.) Ako `BOOKSTORE_H` nije prethodno definisana, uslovna direktiva ima vrednost tačno i uključuju se i obrađuju i svi redovi između nje i direktive `#endif`. Nasuprot tome, ako uslovna direktiva ima vrednost netačno, ignorišu se svi redovi između nje i direktive `#endif`.

Da bi zaglavlje datoteke sigurno bilo obrađeno samo jednom, stavićemo direktivu

```
#define BOOKSTORE_H
```

posle direktive `#ifndef`. Na ovaj način konstanta `BOOKSTORE_H` je definisana kada je prvi put obrađen sadržaj zaglavlja i na taj način je sprečeno da se direktiva `#ifndef` vrednuje kao tačna u daljem procesiranju datoteke sa tekstom programa.

Ova strategija dobro funkcioniše ukoliko se u dva zaglavlja za proveru uključivanja ne upotrebljavaju pretprocesorske konstante sa istim imenom.

Direktiva `#ifdef` najčešće se koristi da uslovno uključi programski kôd zavisno od toga da li je definisana pretprocesorska konstanta. Na primer:

```
int main()
{
#ifdef DEBUG
  cout << "Počinje izvršavanje funkcije main()\n";
#endif
  string word;
  vector< string > text;
  while ( cin >> word )
  {
#ifdef DEBUG
   cout << "pročitana reč: " << word << "\n";
#endif
   text.push_back( word );
  }
  // ...
}
```

U ovom primeru ako `DEBUG` nije definisana, programski kôd koji se u stvari kompajlira je

```
int main()
{
  string word;
  vector< string > text;
  while  ( cin >> word )
  {
         text.push_back( word );
  }
  // ...
}
```

Inače, ako je `DEBUG` definisana programski kôd je prosleđen kompajleru na sledeći način:

```
int main()
{
 cout << "Počinje izvršavanje funkcije main()\n";
 string word;
 vector< string > text;
 while ( cin >> word )
 {
       cout << "pročitana reč: " << word << "\n";
       text.push_back( word );
 }
 // ...
}
```

Možemo da definišemo pretprocesorsku konstantu u komandnoj liniji kada kompajliramo program pomoću opcije `-D` iza koje stoji ime pretprocesorske konstante:[2]

```
$ CC  -DDEBUG main.C
```

ili u programu pomoću direktive `#define`.

Pretprocesorsko ime `__cplusplus` (dva znaka podvlačenja) je automatski definisano kada se kompajlira C++ tako da možemo uslovno da uključimo kôd zavisno od toga da li kompajliramo na jezik C++ ili C. Na primer:

```
#ifdef __cplusplus
  // u redu: kompajliramo na jezik C++
  // objasnićemo extern "C" u Poglavlju 7!
  extern "C"
#endif
   int min( int, int );
```

[2] Ovo je tačno za UNIX-ove sisteme. Windows programeri treba da provere uputstvo za korišćenje kompajlera.

Ime __STDC__ je definisano kada se kompajlira Standardni jezik C. Naravno, imena __cplusplus i __STDC__ nikada nisu definisana u isto vreme.

Još dva korisna prethodno definisana imena su __LINE__ i __FILE__. __LINE__ sadrži redni broj reda datoteke koja se trenutno kompajlira. __FILE__ sadrži ime datoteke koja se trenutno kompajlira. Ova imena se mogu koristi na sledeći način:

```
if ( element_count == 0 )
     cerr  << "Greška: " << __FILE__
       << " : red" << __LINE__
       << "element_count nesme biti 0.\n";
```

Dva dodatna prethodno definisana imena sadrže redom vreme (__TIME__) i datum (__DATE__) kompajliranja datoteke koja se trenutno kompajlira. Format vremena je hh:mm:ss tako da će, na primer, datoteka koja je kompajlirana ujutru u 8 sati i 17 minuta biti predstavljena kao 08:17:05. Ako je datoteka kompajlirana u četvrtak 31. oktobra 1996, datum će biti predstavljen kao

```
Oct 31 1996
```

Vrednosti imena __LINE__ i __FILE__ se ažuriraju kako se menjaju, ovim redosledom redova i datoteka koje se obrađuju. Međutim, ostala četiri prethodno definisana imena ostaju nepromenjena za vreme kompajliranja. Ove vrednosti se ne mogu modifikovati.

assert() je uglavnom koristan pretprocesorski makro koji postoji u standardnoj biblioteci jezika C. Često ga koristimo u tekstu da bismo istakli neophodan preduslov za tačno izvršavanje programa. Na primer, ako treba da učitamo tekstualnu datoteku i sortiramo reči, neophodan preduslov je obezbeđeno ime datoteke jer samo tako možemo da otvorimo datoteku. Da bimo koristili makro assert() moramo da uključimo odgovarajuće zaglavlje.

```
#include <assert.h>
```

Evo jednostavnog primera za njegovo korišćenje:

```
assert( filename != 0 );
```

Makro assert() proverava ispravnost uslova da filename nije jednako 0. Ovo predstavlja navedenu tvrdnju o neophodnom preduslovu za tačno izvršavanje programskog koda koji sledi posle tvrdnje. Ako se uslov oceni kao pogrešan – to jest, filename je jednako 0 – tvrdnja je nezadovoljena: dijagnostička poruka je odštampana i izvršavanje program se prekida.

assert.h je ime u jeziku C za zaglavlje biblioteke jezika C. Program jezika C++ može da referiše zaglavlje biblioteke jezika C pomoću njegovog imena u jeziku C ili C++. Ime u jeziku C++ za ovo zaglavlje je cassert. C++-ovo ime zaglavlja biblioteke jezika C je uvek C ime koje ima kao prefiks slovo c i u kome je izostavljen sufiks datoteke .h (kao što je prethodno objašnjeno, zaglavlje standardno u jeziku C++ nije određeno sufiksom datoteke zato što se sufiks za zaglavlja razlikuje u implementacijama jezika C++).

Pretprocesorska direktiva `#include` za zaglavlje jezika C nema isti efekat zavisno od toga da li se koristi ime iz jezika C ili C++. Sledeća pretprocesorska direktiva `#include`

```
#include <cassert>
```

prouzrokuje da sadržaj zaglavlja `cassert` bude učitan u tekst programa. Ali pošto su sva imena iz biblioteke jezika C++ deklarisana u prostoru imena `std`, ime `assert()` nije vidljivo u tekstu programa osim ako eksplicitno ne odredimo da bude vidljivo pomoću sledeće direktive za upotrebu:

```
using namespace std;
```

Pomoću pretprocesorske direktive `#include` koja koristi ime zaglavlja jezika C,

```
#include <assert.h>
```

ime `assert()` može da se koristi u tekstu programa bez potrebe korišćenja direktive za upotrebu.[3] (Prostore imena koriste proizvođači biblioteka za kontrolu zatrpavanja globalnog prostora imena koje pravi njihova biblioteka u prostoru imena korisničkog programa. O ovome se detaljnije govori u odeljku 8.5.)

1.4 Nekoliko reči o komentarima

Komentari služe kao pomoć ljudima koji čitaju naše programe; oni predstavljaju inženjersku etičnost. Oni mogu da predstavljaju skicu algoritma funkcije, opišu namenu promenljive ili razjasne nejasan segment koda. Komentari ne povećavaju veličinu izvršnog programa. Kompajler izuzima komentare iz programa pre generisanja koda.

Postoje dve oznake za komentare u jeziku C++. Par oznaka za komentare (/*,*/) je ista oznaka kao u jeziku C. Početak komentara je označen sa /*. Kompajler će tretirati sve što se nađe između oznake /* i odgovarajuće oznake */ kao deo komentara. Par oznaka komentara može da stoji tamo gde su dozvoljeni tabulator, razmaknica ili novi red i može da obuhvata više redova programa. Na primer:

```
/*
* Ovo je prvi prikaz definicije klase u jeziku C++.
* Klase se upotrebljavaju u objektno zasnovanom i
* objektno orijentisanom programiranju. Implementacija
* klase Screen predstavljena je u poglavlju 13.
*/
```

[3] U vreme kada je pisana ova knjiga nisu sve implementacije jezika C++ podržavale C++ imena za biblioteke zaglavlja iz jezika C. Pošto su mnogi primeri u ovoj knjizi prevedeni sa implementacijama koje ne podržavaju imena zaglavlja iz jezika C++, primeri se odnose na biblioteke zaglavlja iz jezika C koje ponekad imaju imena iz jezika C, a ponekad imena iz jezika C++.

```
class Screen {
  /* Ovo je telo klase */
public:
  void home();  /* pomeranje kursora do (0,0) */
  void refresh();/* crtanje objekta            */
private:
  /* Klase podržavaju "skrivanje informacija".     */
  /* Skrivanje informacija umanjuje upliv programa */
  /* u internu reprezentaciju klase                */
  /* (njenih podataka). To se izvodi               */
  /* upotrebom oznake "private: "                  */
  int height, width;
};
```

Suviše komentara izmešanih sa programskim kodom može kôd da učini nečitkim. Okruženo komentarima, na primer, deklarisanje `width` i `height` je skoro sakriveno. Uopšteno govoreći, bolje je da blok komenatara smestite iznad teksta koji on objašnjava. Kao i kod druge softverske dokumentacije komentari se, kako se razvija softver, moraju ažurirati da bi bili svrsishodni. Suviše često se dešava da se tokom vremena udalji značenje komentara od koda koji oni objašnjavaju.

Parovi oznaka komentara ne mogu se ugnezditi – to jest, jedan par komentara ne može da se pojavi u okviru drugog para. Pokušajte da kompajlirate sledeći program na svom sistemu. On potpuno zbunjuje većinu kompajlera.

```
#include <iostream>

/*
* parovi oznaka za komentare /* */ se ne ugnežđuju.
* "se ne ugnežđuju" se smatra tekstom programa,
* kao i ovi redovi koji slede.
*/

int main() {
    cout << "hello, world\n";
}
```

Jedan način da se ispravi problem ugneždenih parova komentara je da stavite razmak između zvezdice i kose crte:

```
/* */
```

Zvezdica i kosa crta se tretiraju kao oznaka komentara samo ako nisu razdvojene razmakom.

Druga oznaka komentara, dve kose crte (`//`), služi da započne komentar dužine jednog reda. Sve što je u redu programa desno od oznake tretira se kao komentar, a kompajler ga ignoriše. Na primer, evo naše klase Screen uz upotrebu obe oznake za komentare:

```
/*
* Ovo je prvi prikaz definicije klase u jeziku C++.
* Klase se upotrebljavaju u objektno zasnovanom i
* objektno orijentisanom programiranju. Implementacija
* klase Screen predstavljena je u poglavlju 13.
*/
class Screen {
  // Ovo je telo klase
public:
  void home();  // pomeranje kursora do (0,0)
  void refresh();// crtanje objekta
private:
  /* Klase podržavaju "skrivanje informacija".     */
  /* Skrivanje informacija umanjuje upliv programa */
  /* u internu reprezentaciju klase                */
  /* (njenih podataka). To se izvodi               */
  /* upotrebom oznake "private: "                  */

  // privatni podaci se nalaze ovde ...
};
```

Programi obično sadrže mešavinu obe forme komentara. Objašnjenja dužine nekoliko redova obično su izdvojena između parova oznaka komentara. Komentari dužine jednog ili pola reda često su označeni sa dve kose crte.

1.5 Prvi pogled na ulaz i izlaz

U jeziku C++ ulaz i izlaz omogućava biblioteka iostream, objektno orijentisana hijerarhija klasa implementirana u jeziku C++ koja predstavlja deo standardne biblioteke.

Ulaz sa terminala, koji se naziva *standardni ulazni tok,* povezan je sa prethodno definisanim iostream objektom `cin` (izgovara se „si-in"). Izlaz usmeren ka terminalu, koji se naziva *standardni izlazni tok,* povezan je sa prethodno definisanim iostream objektom `cout` (izgovara se „si-aut"). Treći prethodno definisani iostream objekat `cerr` (izgovara se „si-er") naziva se *standardni tok za greške* (engl. *standard error*) koji je takođe povezan sa terminalom. Objekat `cerr` obično se koristi da za korisnike programa generiše poruke o upozorenju i grešci.

Program koji želi da iskoristi biblioteku iostream mora da uključi pridruženo sistemsko zaglavlje:

```
#include <iostream>
```

Operator prosleđivanja (<<) koristi se da prosledi vrednost na standardni izlazni tok ili standardni tok za greške. Na primer:

```
int v1, v2;
// ...
cout << "Zbir od v1 + v2 = ";
cout << v1 + v2;
cout << '\n';
```

Sekvenca od dva znaka `\n` predstavlja znak za novi red. Kada je ispisan, znak za novi red završava red i prouzrokuje da naredni izlaz bude upućen na novi red. Umesto eksplicitno napisanog znaka za novi red, možemo primeniti prethodno definisani *manipulator* u/i tokova `endl`.

Manipulator obavlja operaciju na u/i toku umesto da jednostavno obezbedi podatke. `endl`, na primer, umeće znak za novi red u izlazni tok, a zatim realizuje bafer toka. Umesto da napišemo

```
cout << '\n';
```

napisaćemo

```
cout << endl;
```

(O prethodno definisanim manipulatorima u/i tokova govori se u poglavlju 20.)

Uzastopno pojavljivanje operatora prosleđivanja može biti nadovezano. Na primer:

```
cout << "Zbir od v1 + v2 = " << v1 + v2 << endl;
```

Svaki uzastopni operator se primenjuje u povratku na `cout`. Radi čitljivosti, nadovezana izlazna naredba može obuhvatiti nekoliko redova. Sledeća tri reda čine jednu izlaznu naredbu:

```
cout << "Zbir od "
     << v1 << " + "
     << v2 << " = " << v1 + v2 << endl;
```

Slično ovome, operator izdvajanja (>>) koristi se da pročita vrednost iz standardnog ulaza. Na primer:

```
string file_name;
// . . .
cout << "Molimo vas da upišete datoteku koja treba da se otvori: ";
cin >> file_name;
```

Uzastopno pojavljivanje operatora izdvajanja se takođe može nadovezati. Na primer[4]:

```
string ifile, ofile;
// ...
cout << "Molimo vas da upišete imena datoteka ulaznog i
             ➥ izlaznog toka: ";
cin  >> ifile >> ofile;
```

Kako možemo da pročitamo nepoznati broj ulaznih vrednosti? To smo već uradili na kraju odeljka 1.2. Deo koda

---

[4] Znak za nastavljanje koda (➥) ukazuje na mesto na kome bi linija koda trebalo da se prelomi da bi se uklopila u format strane, mada tu nije stvarno prelomljena. Ako ukucavate kôd iz knjige, ne prelamajte liniju na tom mestu – jednostavno nastavite dalje.

```
string word;
while ( cin >> word )
      // ...
```

će pročitati po jednu nisku iz standardnog ulaznog toka sa svakom iteracijom naredbe ponavljanja while, sve dok se ne pročitaju sve niske. Uslov

```
( cin >> word)
```

ima netačnu vrednost kada se dođe do kraja datoteke (kako se ovo događa objašnjeno je u poglavlju 20). Evo jednostavnog programa koji koristi navedeni deo koda:

```
#include <iostream>
#include <string>

int main()
{
  string word;

  while ( cin >> word )
       cout << "pročitana reč je: " << word << '\n';

  cout << "u redu: nema više reči koje treba pročitati: zdravo!\n";
  return 0;
}
```

Sledi prvih pet reči novele Džejmsa Džojsa *Fineganovo bdenije:*

```
riverrun, past Eve and Adam's
```

Kada upišete ove reči preko tastature, program formira sledeći izlazni izveštaj:

```
pročitana reč je: riverrun,
pročitana reč je: past
pročitana reč je: Eve
pročitana reč je: and
pročitana reč je: Adam's
pročitana reč je: u redu: nema više reči koje treba pročitati: zdravo!
```

(U poglavlju 6 ćemo videti kako možemo da uklonimo pravopisne znake iz raznih ulaznih niski.)

1.5.1 Čitanje i pisanje datoteke

Iostream biblioteka takođe podržava rad sa ulaznim i izlaznim datotekama. Svi operatori koji se mogu primeniti na standardni ulazni i izlazni tok mogu se takođe primeniti na datoteke koje su otvorene za čitanje ili pisanje (ili jedno i drugo). Da bismo otvorili datoteku za čitanje ili pisanje, pored zaglavlja `iostream` moramo da uključimo zaglavlje

```
#include <fstream>
```

Da bismo otvorili datoteku za pisanje, deklarišemo objekat tipa `ofstream`:

```
ofstream outfile( "name-of-file" );
```

Da bismo proverili da li je datoteka uspešno otvorena, možemo da napišemo

```
// ima vrednost false ako ne uspe otvaranje
if ( ! outfile )
    cerr << "Izvinite! Nismo mogli da otvorimo datoteku!\n";
```

Slično ovome, da bismo otvorili datoteku za čitanje, deklarisaćemo objekat tipa `ifstream`:

```
ifstream infile( "name of file" );
if ( ! infile )
    cerr << "Izvinite! Nismo mogli da otvorimo datoteku!\n"'
```

Ovo je mali program koji čita tekstualnu ulaznu datoteku, pod nazivom `in_file` i ispisuje svaku reč u izlaznoj datoteci, pod nazivom `out_file`, razdvajajući reči razmakom u izlaznoj datoteci.

```
#include <iostream>
#include <fstream>
#include <string>
int main()
{
  ofstream outfile( "out_file" );
  ifstream infile( "in_file" );
  if ( ! infile ) {
       cerr << "greška: ulazna datoteka ne može da se otvori!\n"';
      return -1;
  }
  if ( ! outfile ) {
       cerr << "greška: izlazna datoteka ne može da se otvori!\n";
      return -2;
  }
  string word;
  while  ( infile >> word )
            outfile << word << ' ';
  return 0;
}
```

U poglavlju 20 detaljno ćemo razmotriti biblioteku iostream, uključujući ulazne i izlazne datoteke. Sada kada imamo predstavu šta pruža jezik, pogledaćemo uvođenje novih tipova u jezik koristeći se mehanizmima klasa i šablona.