1

Osnovi programiranja na jeziku C++

U ovom poglavlju razvijamo mali program radi vežbanja temeljnih komponenti jezika C++. To su sledeće komponente:

1. Mali skup tipova podataka: Bulov, znakovni, celobrojni i brojevi u pokretnom zarezu.
2. Skup aritmetičkih, relacionih i logičkih operatora pomoću kojih se manipuliše ovim tipovima. Tu spadaju uobičajeni krivci, kao što su: sabiranje, jednakost, manje od i dodeljivanje, ali takođe i oni koji nisu baš klasični, kao što su inkrementalni, uslovni i operatori složenog dodeljivanja.
3. Skup naredbi uslovnog grananja i ponavljanja, kao što su naredba `if` i petlja `while`, kojima se menja kontrolni tok programa.
4. Mali broj složenih tipova, kao što su pokazivači i nizovi. Oni nam omogućavaju indirektno referenciranje nekog postojećeg objekta i definisanje kolekcije elemenata jednog tipa.
5. Standardna biblioteka uobičajenih apstrakcija u programiranju, kao što su string i vector.

1.1 Kako pisati C++ program

Od nas se traži da napišemo jednostavan program za pisanje poruke na korisnikovom terminalu kojom se od njega traži da upiše svoje ime. Čitamo upisano ime, pohranjujemo ga za kasniju upotrebu i na kraju pozdravljamo korisnika po imenu.

U redu. Dakle, odakle se počinje? Počinje se sa istog mesta od koga počinje svaki C++ program – od funkcije `main()`. Funkcija `main()` je korisnički implementirana funkcija koja ima sledeći opšti oblik:

```
int main()
{
   // ovde dolazi programski kôd
}
```

`int` je ključna reč jezika C++. *Ključne reči* su unapred definisana imena koja u jeziku imaju posebno značenje. `int` predstavlja ugrađeni celobrojni tip podataka. (U sledećem odeljku ću mnogo više reći o tipovima podataka.)

Funkcija je nezavisna sekvenca koda koji izvodi neko izračunavanje. Sastoji se od četiri dela: tipa rezultata, imena funkcije, liste parametara i tela funkcije. Pogledajmo načas svaki od ovih delova.

Tip rezultata funkcije obično predstavlja rezultat dobijen izračunavanjem. Tip rezultata funkcije `main()` je ceo broj. Vrednost koju kao rezultat daje funkcija `main()` pokazuje da li je program uspešan. Uobičajeno je da funkcija `main()` kao rezultat daje 0, što ukazuje na uspešnost. Vrednost različita od nule pokazuje da nešto nije u redu.

Ime funkcije bira programer i idealno je da ono pokazuje šta funkcija radi. Na primer, imena funkcija `main()` i `sort()` su prilično dobra imena. Loša imena su `f()` i `g()`. Zašto? Zato što ne informišu o tome šta funkcija radi.

`main` nije ključna reč jezika. Međutim, sistem za prevođenje koji izvršava C++ program očekuje da funkcija `main()` bude definisana. Ako se ona izostavi, program neće raditi.

Lista parametara funkcije se stavlja u zagrade i dolazi iza imena funkcije. Ako u listi parametara nema ničega, kao u `main()`, to pokazuje da funkcija ne prihvata parametre.

Lista parametara je obično lista tipova, odvojenih zarezom, koje korisnik predaje funkciji kada se ona izvršava. (Obično se kaže da je korisnik *pozvao* ili *pokrenuo* funkciju.) Na primer, ako pišemo funkciju `min()` koja u rezultatu treba da vrati manju od dve vrednosti, njena lista parametara treba da identifikuje tipove dve vrednosti koje želimo da uporedimo. Funkcija `min()` za poređenje dve celobrojne vrednosti mogla bi da se definiše ovako:

```
int min( int val1, int val2 )
{
   // ovde dolazi kôd programa...
}
```

Telo funkcije se stavlja u vitičaste zagrade (`{}`). Ono sadrži kod koji obezbeđuje izračunavanje funkcije. Dvostruka kosa crta (`//`) predstavlja komentar, programerovu napomenu o nekom aspektu koda. Komentar je namenjen čitaocu programa i tokom kompajliranja se ne uzima u obzir. Sve što sledi iza dve kose crte, do kraja reda, smatra se komentarom.

Naš prvi zadatak jeste da napišemo poruku na korisnikovom terminalu. Ulaz i izlaz nisu unapred definisani delovi jezika C++. Oni su podržani *objektno orijentisanom hijerarhijom klasa*, implementiranom na jeziku C++ kao deo njegove standardne biblioteke.

Klasa je korisnički definisan tip podataka. Mehanizam klasa je metod dodavanja tipova podataka koje naš program prepoznaje. Objektno orijentisana hijerarhija klasa definiše familiju srodnih tipova klasa, kao što su ulaz sa terminala ili iz

datoteke, izlaz na terminal ili u datoteku itd. (O klasama i objektno orijentisanom programiranju će biti još govora u ovoj knjizi.)

U jeziku C++ je unapred definisan mali skup osnovnih tipova podataka: Bulovi (logički), znakovni, celobrojni i tip brojeva u pokretnom zarezu. Iako predstavljaju osnovu svakog programiranja, oni nisu u središtu pažnje programa. Na primer, kamera mora da ima položaj u prostoru, koji se obično predstavlja pomoću tri broja sa pokretnim zarezom. Kamera takođe mora da ima ugao iz koga posmatra, koji se takođe definiše pomoću tri broja sa pokretnim zarezom. Obično postoji i proporcija kojom se iskazuje odnos širine i visine vidnog polja kamere. To se predstavlja jednim brojem sa pokretnim zarezom.

Na najprimitivnijem nivou, kamera se predstavlja pomoću sedam brojeva sa pokretnim zarezom, od kojih šest predstavljaju dve uređene n-torke x, y i z koordinata. Programiranje na ovako niskom nivou zahteva od programera da pažnju prenosi sa apstrahovanja kretanja kamere na odgovarajuću manipulaciju sa sedam vrednosti sa pokretnim zarezom koje u programu predstavljaju kameru.

Mehanizam klasa omogućava da dodajemo slojeve apstrahovanju tipova u programu. Na primer, možemo da definišemo klasu Point3d koja predstavlja položaj i orijentaciju u prostoru. Isto tako, možemo da definišemo klasu Camera koja sadrži dva objekta klase Point3d i jednu vrednost sa pokretnim zarezom. Tako i dalje predstavljamo kameru sa sedam vrednosti sa pokretnim zarezom. Razlika je u tome što sada tokom programiranja manipulišemo direktno klasom Camera umesto pomoću sedam vrednosti sa pokretnim zarezom.

Definicija klase se obično deli u dva dela, od kojih svaki predstavlja drugu datoteku: *datoteku zaglavlja* koja sadrži deklaraciju operacija koje klasa podržava, i *datoteku teksta programa* koja sadrži implementaciju tih operacija.

Da bismo koristili klasu, njenu datoteku zaglavlja stavljamo u program. Zahvaljujući datoteci zaglavlja, program poznaje klasu. Standardna biblioteka ulaza/izlaza jezika C++ zove se *iostream* biblioteka (ili biblioteka ulazno/izlaznih tokova). Ona se sastoji od kolekcije srodnih klasa koje podržavaju ulaz i izlaz na korisnikov terminal i u datoteku. Da bi se koristila biblioteka klasa iostream, moramo da uključimo njenu datoteku zaglavlja:

```
#include <iostream>
```

Da bismo pisali na korisnikovom terminalu, koristimo preddefinisani klasni objekat `cout`. Podatke koje želimo da `cout` napiše, usmeravamo korišćenjem operatora izlaza, ovako:

```
cout <<  "Molim upišite svoje ime: ";
```

Ovo predstavlja *naredbu* jezika C++, najmanju nezavisnu jedinicu C++ programa. Ona je isto što i rečenica u prirodnom jeziku. Naredba se završava tačkom i zarezom. Naša naredba izlaza ispisuje literalnu nisku (označenu dvostrukim znakovima navoda) na korisnikovom terminalu. Znakovi navoda identifikuju string; oni se ne prikazuju na terminalu. Korisnik vidi:

```
Molim upišite svoje ime:
```

Naš sledeći zadatak jeste da pročitamo korisnikov unos. Da bismo mogli da pročitamo ime koje korisnik upiše, moramo da definišemo jedan objekat u kome će se čuvati ta informacija. Objekat definišemo navođenjem tipa podataka objekta i dajući mu ime. Već smo videli jedan tip podatka: `int`. Međutim, on teško da je koristan za čuvanje nečijeg imena. Podesniji tip u ovom slučaju jeste klasa string iz standardne biblioteke:

```
string user_name;
```

Ovim smo definisali `user_name` kao objekat klase string. Može da deluje čudno, ali se definicija zove *naredba deklaracije*. Međutim, ova naredba neće biti prihvaćena ako prvo ne upoznamo program sa klasom string. To se radi uključivanjem datoteke zaglavlja sa klasom string:

```
#include <string>
```

Da bismo pročitali unos sa korisnikovog terminala, koristimo preddefinisani klasni objekat `cin`. Koristimo operator ulaza (>>) da bismo uputili `cin` da pročita podatke sa korisnikovog terminala u objekat odgovarajućeg tipa:

```
cin >> user_name;
```

Izlaz i ulaz na korisnikovom terminalu sada izgledaju ovako (korisnikov unos je ispisan polucrnim slovima):

```
Molim upišite svoje ime: ana
```

Preostalo nam je još samo da pozdravimo korisnika po imenu. Želimo da izlaz izgleda ovako:

```
Zdravo, ana ... i doviđenja!
```

Znam, i nije baš neki pozdrav, ali ovo je tek prvo poglavlje. Postaćemo inventivniji do kraja knjige.

Da bismo generisali pozdrav, prvi korak je da nastavimo izlaz u novom redu. To radimo ispisivanjem znakovnog literala za novi red u `cout`:

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

Znakovni literal se obeležava parom jednostrukih navodnika. Postoje dva osnovna oblika literala: znakovi koji se štampaju kao što su slova abecede (`'a'`, `'A'` itd), brojevi i znakovi interpunkcije (`';'`, `'-'` itd) i znakovi koji se ne štampaju, kao što je znak za novi red (`'\n'`) ili znak tabulatora (`'\t'`). Kako ne postoji slovni pandan znakovima koji se ne štampaju, oni koji se najčešće koriste, kao dva koja smo naveli, predstavljaju se posebnim sekvencama od dva znaka.

Pošto smo prešli u novi red, hoćemo da generišemo svoje `Zdravo`:

```
cout << "Zdravo, ";
```

Sada treba da postignemo da nam izlaz bude korisnikovo ime. Ono se nalazi u našem objektu tipa string `user_name`. Kako ćemo to uraditi? Isto kao sa ostalim tipovima:

```
cout << user_name;
```

Na kraju, pozdrav završavamo opraštajući se (napominjem da jedna literalna niska može da se sastoji od znakova koji se štampaju i od znakova koji se ne štampaju):

```
cout << " ... i doviđenja!\n";
```

Po pravilu, svi ugrađeni tipovi se ispisuju na isti način – odnosno, postavljanjem vrednosti sa desne strane operatora izlaza. Na primer,

```
cout << "3 + 4 = ";
cout << 3 + 4;
cout << '\n';
```

generiše sledeći izlaz:

```
3 + 4 = 7
```

Kada definišemo nove tipove klasa koje ćemo koristiti u našim aplikacijama, istovremeno obezbeđujemo i primerak operatora izlaza za svaku klasu. (U poglavlju 4 videćete kako se to radi.) To korisnicima naših klasa omogućuje da ispisuju individualne objekte klasa na tačno isti način kao ugrađene tipove.

Umesto da pišemo uzastopne naredbe izlaza u posebnim redovima, možemo da ih nanižemo u jednu složenu naredbu izlaza:

```
cout  << '\n'
      << "Zdravo,"
      << user_name
      << "... i doviđenja!\n";
```

Na kraju, možemo eksplicitno da završimo `main()` korišćenjem naredbe return:

```
return 0;
```

`return` je ključna reč. Izraz koji sledi za ključnom reči `return`, u ovom slučaju 0, predstavlja vrednost rezultata funkcije. Setite se, ako funkcija `main()` u rezultatu vrati 0, to znači da se program uspešno izvršio.[1]

Kada sastavimo sve delove, dobijamo naš prvi C++ program:

```
#include <iostream>
#include <string>
using namespace std; // ovo još nisam objasnio...
int main()
{
   string user_name
   cout  << "Molim upišite svoje ime: ";
   cin   >> user_name;
   cout  << '\n'
         << "Zdravo, "
```

[1] Ako na kraj funkcije `main()` ne stavimo eksplicitnu naredbu `return`, automatski se umeće naredba `return 0;`. U primerima programa u ovoj knjizi ne stavljam eksplicitne naredbe `return`.

```
                << user_name
                << "... i doviđenja!\n";
    return 0;
}
```

Kada se kompajlira i izvrši, ovaj kod daje sledeći izlaz (moj ulaz je istaknut polucrnim slovima):

```
Molim upišite svoje ime: ana
Zdravo, ana ... i doviđenja!
```

Tu je i jedna naredba koju nisam objasnio:

```
using namespace std;
```

Da vidimo da li mogu da je objasnim, a da vas ne preplašim. (U ovom trenutku preporučujem da duboko udahnete!) Reči `using` i `namespace` su ključne reči jezika C++. `std` je ime prostora imena standardne biblioteke. Sve što se nalazi u standardnoj biblioteci (kao klasa string i objekti – u/i tokovi – klase iostream `cout` i `cin`) enkapsulirano je unutar prostora imena `std`. Naravno, sledeće pitanje je šta je prostor imena (engl. *namespace*)?

Prostor imena je metod pakovanja imena biblioteke tako da mogu da se uvedu u okruženje korisnikovog programa, a da ne dođe do sukoba imena. (Do *sukoba imena* dolazi kada u aplikaciji postoje dva primerka sa istim imenom, između kojih program ne može da napravi razliku. Kada do toga dođe, program ne može da radi dok se ne razreši sukob imena.) Prostor imena je način da se vidljivost imena stavi u neke okvire.

Da bismo u programu koristili klasu string i objekte `cin` i `cout` hijerarhije klasa iostream, ne samo da moramo da uključimo string i iostream datoteke zaglavlja, već moramo i da učinimo vidljivim imena u okviru prostora imena `std`. *Direktiva using*

```
using namespace std;
```

je najjednostavniji način da se imena iz nekog prostora imena učine vidljivim. (Ako želite da saznate više o prostorima imena, pročitajte odeljak 8.5 iz [LIPPMAN98] ili odeljak 8.2 iz [STROUSTRUP97].)

Vežba 1.1

Uđite u program `main()` koji smo ranije pokazali. Ili ga napišite, ili ga učitajte; pročitajte predgovor da biste saznali kako da dođete do izvornog programa i rešenja za vežbe. Prevedite i izvršite program na svom sistemu.

Vežba 1.2

Iskomentarišite datoteku zaglavlja string:

```
// #include <string>
```

Sada prekompajlirajte program. Šta se dešava? Sada vratite zaglavlje string i iskomentarišite

```
//using namespace std;
```

Šta se dešava?

Vežba 1.3

Promenite ime funkcije `main()` u `my_main()` i ponovo kompajlirajte program. Šta se dešava?

Vežba 1.4

Pokušajte da proširite program: (1) Tražite od korisnika da upiše ime i prezime; i (2) izmenite izlaz tako da se ispišu ime i prezime.

1.2 Definisanje i inicijalizovanje podataka

Pošto smo privukli pažnju korisnika, izazovimo je na kviz. Prikazaćemo dva broja koja predstavljaju neki brojni niz i tražiti od korisnika da izračuna koja je sledeća vrednost u nizu. Na primer,

```
Vrednosti 2, 3 čine dva uzastopna elementa numeričkog niza.
Koji je sledeći broj?
```

Ove vrednosti su treći i četvrti elemenat Fibonačijevog niza: 1, 1, 2, 3, 5, 8, 13 itd. Prva dva elementa Fibonačijevog niza imaju vrednost 1. Vrednost svakog sledećeg elementa je zbir vrednosti prethodnih elemenata. (U poglavlju 2 pišemo funkciju koja izračunava elemente.)

Ako korisnik upiše 5, čestitamo joj i pitamo je da li bi želela da pokuša sa još jednim numeričkim nizom. Svaki drugi odgovor je netačan, i u tom slučaju pitamo korisnika da li bi želela da pogađa ponovo.

Da bismo program učinili zanimljivijim, pratimo rezultat iskazan brojem tačnih odgovora podeljenim sa brojem pokušaja.

Našem programu je potrebno najmanje pet objekata: objekat klase string da bi čuvao ime korisnika; tri cela broja – jedan za čuvanje pogađanja, drugi za čuvanje broja pogađanja i treći za čuvanje broja tačnih odgovora; objekat u pokretnom zarezu za čuvanje rezultata koji korisnik postigne.

Da bismo definisali objekat podatka, moramo da mu damo ime i da mu obezbedimo tip. Ime može biti svaka kombinacija slova, cifara i znaka podvlake. Razlikuju se mala i velika slova, te prema tome `user_name`, `User_name`, `uSeR_nAmE` i `user_Name` su imena četiri različita objekta.

Ime ne može da počne brojem. Na primer, `1_name` nije dozvoljeno, a ime `name_1` je dozvoljeno. Isto tako, ime ne sme da bude jednako nekoj ključnoj reči jezika.

Na primer, `delete` je ključna reč, pa ne možemo da je koristimo za označavanje entiteta u programu. Što je istovremeno i objašnjenje zašto se radnja uklanjanja znaka iz klase string zove `erase()`, a ne `delete()`.)

Svaki objekat mora da ima određen tip podatka. Ime objekta nam dozvoljava da se direktno pozivamo na njega. Od tipa podatka zavisi skup vrednosti koje objekat može da sadrži i količina memorije koja mora da mu se dodeli za čuvanje tih vrednosti. U prethodnom odeljku smo videli definiciju objekta `user_name`. U novom programu ćemo je ponovo upotrebiti:

```
#include <string>
string user_name;
```

Klasa je tip podataka koje definiše programer. C++ ima i skup ugrađenih tipova podataka: Bulove (logičke), celobrojne, brojeve u pokretnom zarezu i znakove. Svakom odgovara određena ključna reč koja omogućava da se odredi tip podatka. Na primer, da bi se sačuvala vrednost koju unese korisnik, definišemo objekat celobrojnog tipa:

```
int usr_val;
```

`int` je ključna reč jezika na osnovu koje se prepoznaje da je objekat celobrojnog tipa. Broj koji pokazuje koliko puta korisnik pogađa i broj ispravnih odgovora su celi brojevi. Ono što je ovde drugačije jeste to što želimo da dodelimo i jednom i drugom inicijalnu vrednost 0. Svaki definišemo u posebnom redu:

```
int num_tries = 0;
int num_right = 0;
```

Možemo da ih definišemo jednom deklarativnom naredbom, razdvajajući ih zarezom:

```
int num_tries = 0, int num_right = 0;
```

Dobro je inicijalizovati objekte podataka čak i ako vrednost jednostavno pokazuju da taj objekat još uvek nema korisnu vrednost. Nisam inicijalizovao `usr_val` zato što se njegova vrednost uzima direktno iz korisnikovog unosa pre nego što program uopšte upotrebi objekat.

Alternativna inicijalizaciona sintaksa, koja se naziva *sintaksa konstruktora*, jeste

```
int num_tires( 0 );
```

Znam, sada se pitate čemu dve inicijalizacione sintakse? Još gore, pitate se zašto vam o tome govorim sada? Pogledajmo da li sledeće objašnjenje može da bude zadovoljavajući odgovor na oba ili bar na jedno od pitanja.

Upotreba operatora dodeljivanja (=) nasleđena je iz jezika C. Ona je dobra za objekte podataka ugrađenih tipova i za objekte klasa koji se mogu inicijalizovati jednom vrednošću, kao što je klasa string:

```
string sequence_name = "Fibonacci";
```

Ona nije dobra za objekte klasa koji zahtevaju više inicijalnih vrednosti, kao što je klasa kompleksnih brojeva iz standardne biblioteke, koja može da uzme dve inicijalne vrednosti: jednu za svoju realnu komponentu i jednu za svoju

imaginarnu komponentu. Alternativna inicijalizaciona sintaksa konstruktora uvedena je da bi se rukovalo inicijalizacijom sa više vrednosti:

```
#include <complex>
complex<double> purei( 0, 7 );
```

Čudna notacija sa zagradama koja sledi iza `complex` pokazuje da je klasa complex šablon klase. U ovoj knjizi ćete se još mnogo puta susretati sa šablonima klasa. Šabloni klasa vam omogućavaju da definišete klasu bez potrebe da navodite tip podatka jednog ili svih njenih članova.

Na primer, klasa kompleksnih brojeva sadrži dva podatka člana. Jedan član predstavlja realan deo broja. Drugi član predstavlja imaginarni deo broja. Ti članovi moraju da budu brojevi u pokretnom zarezu, ali koji? C++ podržava tri „veličine" brojeva u pokretnom zarezu: *jednostruke tačnosti*, koje predstavlja ključna reč `float`; *dvostruke tačnosti* koje predstavlja ključna reč `double`; i *proširene tačnosti*, koje predstavljaju dve ključne reči `long double`.

Mehanizam šablona klasa omogućava programeru da odloži donošenje odluke o tipu podatka koji će upotrebiti za šablon klase. On može da rezerviše mesto koje će kasnije povezati sa konkretnim tipom podatka. U prethodnom primeru korisnik je odlučio da tip podatka člana klase complex bude `double`.

Sve ovo je verovatno proizvelo mnogo više pitanja nego što je dalo odgovora. Međutim, zbog šablona, C++ podržava dve inicijalizacione sintakse za ugrađene tipove podataka. Dok su ugrađeni tipovi podataka i tipovi klasa koje definiše programer imali različite inicijalizacione sintakse, bilo je nemoguće napisati šablon koji je podržavao obe vrste tipova podataka. Uvođenje jednoobraznosti u sintaksu pojednostavilo je pravljenje šablona, ali je, nažalost, prilično iskomplikovalo objašnjavanje sintakse!

Korisnikov tekući rezultat mora da bude broj u pokretnom zarezu, jer tako može da se iskaže i procenat. Definisaćemo ga da bude tipa `double`:

```
double usr_score = 0.0;
```

Treba takođe da pratimo i korisnikove odgovore tipa *da/ne* na pitanje: *Još jedan pokušaj?*

Korisnikov odgovor možemo da sačuvamo kao objekat znakovnog tipa:

```
char usr_more;
cout << "Još jedan pokušaj? D/N? ";
cin >> usr_more;
```

Ključna reč `char` predstavlja znakovni tip. Znak pod jednostrukim navodnicima predstavlja znakovni literal: `'a'`, `'7'`, `';'` itd. Posebni ugrađeni znakovni literali su sledeći (ponekad se zovu i *iskočne sekvence*):

```
'\n'  znak za novi red (newline)
'\t'  tabulator (tab)
'\0'  prazan znak (null)
'\''  jednostruki navodnik (single quote)
'\"'  dvostruki navodnik (double quote)
'\\'  obrnuta kosa crta (backslash)
```

Na primer, da bi se generisao novi red i za njim tabulator pre nego što se korisnikovo ime prikaže, možemo da napišemo:

```
cout << '\n' << '\t' << user_name;
```

Druga mogućnost je da nanižemo znakove u nisku:

```
cout << "\n\t" << user_name;
```

Ovi posebni znakovi se obično koriste u literalnim niskama. Na primer, da bismo literalom prikazali putanju datoteke pod Windowsom, potrebno je da ponavljamo obrnutu kosu crtu:

```
"F:\\osnove\\programi\\poglavlje1\\ch1_main.cpp";
```

C++ podržava ugrađen logički tip podataka kojim se predstavljaju vrednosti tačno (*true*)/netačno (*false*). Na primer, u našem programu možemo da definišemo logički objekat da bismo kontrolisali da li se prikazuje sledeća numerička sekvenca:

```
bool go_for_it = true;
```

Logički (Bulovi) objekti se označavaju ključnom reči `bool`. Ona može da ima jednu od dve literalne vrednosti: `true` (tačno) ili `false` (netačno).

Svi objekti podataka koje smo do sada definisali modifikovani su tokom rada programa. Na primer, `go_for_it` dobija vrednost `false`. `usr_score` se ažurira sa svakim pogađanjem korisnika.

Međutim, ponekad je potreban objekat koji predstavlja konstantu: na primer, maksimalan broj dopuštenih pogađanja ili vrednost broja pi. Objekti koji čuvaju te vrednosti ne smeju se modifikovati tokom rada programa. Kako možemo da sprečimo slučajno modifikovanje takvih objekata? U tome nam pomaže jezik zato što dopušta da ovakve objekte definišemo kao `const`:

```
const int max_tries = 3;
const double pi = 314159;
```

Objektu `const` ne može da se promeni inicijalna vrednost. Svaki pokušaj da mu se dodeli druga vrednost dovodi do greške tokom izvršavanja programa. Na primer:

```
max_tries = 42; // greška: objekat const
```

1.3 Zapisivanje izraza

Ugrađeni tipovi podataka podržani su kolekcijom aritmetičkih, relacionih, logičkih operatora i složenih operatora dodeljivanja. Aritmetički operatori nisu nikakvo iznenađenje, osim operatora celobrojnog deljenja i operatora ostatka:

```
// Aritmeticki operatori
+  sabiranje  a + b
-  oduzimanje a – b
*  množenje a * b
/  deljenje a / b
%  ostatak    a % b
```

Deljenjem dve celobrojne vrednosti dobija se ceo broj. Svaki ostatak se odseca; nema zaokruživanja. Ostatku se pristupa korišćenjem operatora %:

```
5 / 3 daje 1 dok 5 % 3 daje 2
5 / 4 daje 1 dok 5 % 4 daje 1
5 / 5 daje 1 dok 5 % 5 daje 0
```

Kada se, u stvari, koristi operator ostatka? Zamislite da želimo da se prikaže najviše osam niski u redu. Ako je broj reči u redu manji od osam, iza reči ispisujemo prazan znak. Ako je niska osma reč u redu, ispisujemo novi red. Evo kako se to implementira:

```
const int line_size = 8;
int cnt = 1;

// ove naredbe se izvršavaju mnogo puta, gde
// a_string uvek predstavlja novu vrednost
// i cnt ima prirast od jedan pri svakom izvršavanju...
cout  << a_string
      << ( cnt % line_size ? ' ' : '\n' );
```

Izraz u zagradi iza izlaznog operatora vam verovatno ništa ne znači ukoliko ne znate šta je uslovni operator (`?:`). Rezultat izraza je razmak ili znak za novi red, u zavisnosti od toga da li operator ostatka daje vrednost nula ili vrednost koja nije nula. Da vidimo koliko ovo možemo da razjasnimo.

Izraz

```
cnt % line_size
```

ima rezultat nula uvek kada je `cnt` umnožak od `line_size`; u suprotnom, on uvek daje vrednost koja nije nula. Kuda nas to vodi? Uslovni operator ima sledeći opšti oblik:

```
izraz
     ? računa_se_ako_je_izraz_tačan
     : računa_se_ako_je_izraz_netačan
```

Ako `izraz` ima vrednost tačno, izračunava se izraz koji sledi za znakom pitanja. Ako `izraz` ima vrednost netačno, izračunava se izraz koji sledi za dvotačkom. U našem slučaju, vrednovanje operatoru izlaza treba da dâ razmak ili znak za novi red.

Smatra se da je uslovni izraz netačan ako je njegova vrednost nula. Svaka vrednost koja nije nula smatra se za tačno. U ovom primeru, uvek kada `cnt` nije umnožak broja osam, rezultat je vrednost koja nije nula, izračunava se tačna grana uslovnog operatora i na ekranu se pojavljuje razmak.

Složeni operator dodeljivanja pruža „stenografsku" notaciju za primenu aritmetičkih operacija na objektu kojem treba da se dodeli rezultat. Na primer, umesto da piše

```
cnt = cnt + 2;
```

C++ programer obično piše

```
cnt += 2; // dodati 2 trenutnoj vrednosti cnt
```

Svakom aritmetičkom operatoru odgovara po jedan složeni operator dodeljivanja: `+=`, `-=`, `*=`, `/=` i `%=`.

Kada se neki objekat povećava ili smanjuje za 1, programer koristi operatore inkrementiranja ili dekrementiranja:

```
cnt++; // uvećaj za 1 trenutnu vrednost cnt
cnt–; // smanji za 1 trenutne vrednosti cnt
```

Postoji prefiksna i postfiksna verzija operatora inkrementiranja i dekrementiranja. Ako se koristi kao prefiks, objekat se inkrementira (ili dekrementira) za 1 pre nego što se upotrebi vrednost objekta:

```
int tries = 0;
cout  << "Jeste li spremni za pokušaj #"
      << ++ tries << "?\n";
```

U ovom primeru, `tries` se povećava za 1 pre nego što se njegova vrednost pojavi na ekranu. Ako se koristi u postfiksnom obliku, prvo se vrednost objekta koristi u izrazu, a zatim inkrementira ili dekrementira za 1:

```
int tries = 1;
cout  << "Jeste li spremni za pokušaj #"
      << tries++ << "?\n";
```

U ovom primeru, vrednost `tries` se pojavljuje na ekranu pre nego što se poveća za 1. U oba primera na ekranu se pojavljuje vrednost 1.

Svaki relacioni operator se vrednuje kao tačno ili kao netačno. U njih spadaju sledeći operatori:

```
== jednakost             a == b
!= nejednakost           a != b
<  manje od              a < b
>  veće od               a > b
<= manje od ili jednako  a <= b
>= veće od ili jednako   a >= b
```

Evo kako možemo da koristimo operator jednakosti da bismo ispitali odgovor korisnika:

```
bool usr_more = true;
char usr_rsp;
// pitati korisnika da li želi da nastavi
// učitati odgovor u usr_rsp
if (  usr_rsp == 'N' )
      usr_more = false;
```

Naredba `if` uslovno izvršava naredbu koja za njom sledi ako se izraz u zagradama izračunava kao tačno. U ovom primeru, `usr_more` je `false` ako je `usr_rsp` jednako `'N'`. Ako `usr_rsp` nije jednako `'N'`, u ovom primeru ništa se ne događa. Prilikom upotrebe operatora nejednakosti koristi se suprotna logika:

```
if ( usr_rsp != 'Y' )
     usr_more = false;
```

Problem sa testiranjem usr_rsp samo za 'N' je u tome što korisnik može da upiše malo slovo 'n'. Moramo da priznamo oba. Jedan od načina jeste da se doda klauzula else:

```
if ( usr_rsp == 'N' )
     usr_more = false;
else
if ( usr_rsp == 'n' )
     usr_more = false;
```

Ako je usr_rsp jednako 'N', usr_more se postavlja na false i više se ništa ne radi. Ako nije jednako 'N', izračunava se klauzula else. Ako je usr_rsp jednako 'n', usr_more se postavlja na false. Ako usr_rsp nije jednako ni jednom ni drugom, usr_more se ne menja.

Česta greška programera početnika je da koristi operator dodeljivanja za ispitivanje jednakosti:

```
// greška, ovo dodeljuje literal 'N' objektu usr_rsp
// i zato je uvek true
if ( usr_rsp = 'N' )
     // ...
```

Logički operator OR (||) je druga mogućnost za testiranje istinitosti uslova od više izraza:

```
if ( usr_rsp == 'N' || usr_rsp == 'n' )
     usr_more = false;
```

Logički operator OR se vrednuje kao tačno ako je jedan od njegovih izraza tačan. Prvo se vrednuje krajnji levi izraz. Ako je on tačan, ostatak izraza se ne vrednuje. U našem primeru usr_rsp se poredi sa 'n' samo ako nije jednak 'N'.

Logički operator AND (&&) se vrednuje kao tačno samo ako su oba izraza tačna. Na primer,

```
if ( password &&
     validate ( password ) &&
     ( acct = retrieve_acct_info(password) ))
        // obradi nalog...
```

Prvo se vrednuje prvi izraz. Ako je on netačan, operator AND je netačan; ostali izrazi se ne vrednuju. U ovom primeru, podaci o nalogu se uzimaju samo ako je lozinka definisana i ako je ustanovljeno da je važeća.

Logički operator NOT (!) je tačan ako je izraz na koji se on primenjuje netačan. Na primer, umesto da napišemo

```
if ( usr_more == false )
     cout  << "Vaš rezultat ovoga puta je "
           << usr_score << " Zbogom!\n";
```

možemo da napišemo

```
if ( ! usr_more ) ...
```

Prioritet operatora

Prilikom korišćenja ugrađenih operatora, postoji „kvaka": kada se u jednom izrazu kombinuje više operatora, redosled vrednovanja izraza se ustanovljava na osnovu unapred definisanog nivoa prioriteta svakog operatora. Na primer, rezultat izraza 5 + 2 * 10 je uvek 25 i nikada 70 zato što operator množenja ima viši prioritet od sabiranja; zato se 2 uvek množi sa 10 pre sabiranja sa 5.

Ugrađene nivoe prioriteta možemo da prevaziđemo tako što ćemo operator, kojeg prvo želimo da izračunamo, staviti u zagrade. Tako (5 + 2) * 10 daje vrednost 70.

Sledi spisak u kome su dati prioriteti operatora koje ste do sada upoznali. Operator ima viši prioritet od operatora koji je naveden iza njega, a operatori navedeni u istom redu imaju isti prioritet. Tada je redosled vrednovanja sleva udesno.

```
logicki NOT
aritmeticki ( *, /, % )
aritmeticki ( +, - )
relacioni ( <, >, <=, >= )
relacioni ( ==, != )
logicki AND
logicki OR
dodeljivanje
```

Na primer, da bismo ustanovili da li je `ival` paran broj, mogli bismo da napišemo

```
! ival % 2 // nije sasvim tačno
```

Namera nam je da testiramo rezultat operatora ostatka. Ako je `ival` paran broj, rezultat je nula i logički operator NOT je tačno; u suprotnom, rezultat nije nula i logički operator NOT je netačno. Odnosno, namera nam je da bude tako.

Nažalost, rezultat našeg izraza je sasvim drugačiji. Naš izraz uvek ima vrednost netačno, osim kada je `ival` jednako 0!

Viši prioritet logičkog `NOT` znači da će se on vrednovati prvi. On se primenjuje na `ival`. Ako je `ival` vrednost različita od nule, rezultat je netačno; u suprotnom, rezultat je tačno. Vrednost rezultata tada postaje levi operand operatora ostatka. `False` se pretvara u 0 kada se koristi u aritmetičkim izrazima; `true` u 1. Uz podrazumevane prioritete, izraz postaje `0%2` za sve vrednosti `ival` osim za 0.

Iako to nije ono što smo hteli, nije ni greška, ili bar nije jezička greška. To je samo netačno predstavljanje logike programa koji smo hteli da napišemo, a kompajler to ne može da zna. Prioriteti su ono što programiranje u jeziku C++ čini komplikovanim. Da bi se ovaj izraz ispravno vrednovao, moramo eksplicitno da navedemo redosled izračunavanja, i to korišćenjem zagrada:

```
! ( ival % 2 ) // ok
```

Da biste izbegli ovaj problem, morate da prionete i naučite prioritete C++ operatora. Ja vam ne pomažem utoliko što u ovom odeljku nije predstavljen kompletan skup operatora niti je u potpunosti predstavljena tematika prioriteta. Međutim, ovo bi trebalo da bude dovoljno za početak. Ako želite kompletnu informaciju, pročitajte poglavlje 4 iz [LIPPMAN98] ili poglavlje 6 iz [STROUSTRUP97].

1.4 Pisanje uslovnih naredbi i naredbi ponavljanja

Podrazumeva se da se naredbe izvršavaju jednom u redosledu, počevši od prve naredbe funkcije `main()`. U prethodnom odeljku zavirili smo u naredbu `if`. Naredba `if` nam omogućava da uslovno izvršavamo naredbu ili niz naredbi zavisno od toga koliko je vrednost nekog izraza istinita. Opciona klauzula `else` nam omogućava da ispitamo više uslova za istinitost. Naredba petlje omogućava ponavljanje jedne ili više naredbi na osnovu izračunavanja istinitosnog izraza. Sledeći pseudokôd koristi dve naredbe petlje (`#1` i `#2`), jednu naredbu `if` (`#5`), jednu naredbu `if-else` (`#3`) i drugu uslovnu naredbu koja se zove `switch` (`#4`).

```
// Pseudokôd: Opšta logika našeg programa
dok korisnik želi da pogađa niz brojeva
{ #1
   prikaži niz
   dok odgovor nije tačan i korisnik želi da pogađa ponovo
   { #2
      pročitaj odgovor
      povećaj vrednost broja pokušaja
      ako je odgovor tačan
      { #3
         povećaj broj tačnih odgovora
         promeni got_to u true
      }  else {
         iskazati žaljenje što je korisnikov odgovor pogrešan
            generisati drugačiji odgovor na osnovu
            sadašnjeg broja korisnikovih pokušaja  // #4
        pitati korisnika da li želi da pogađa ponovo
        pročitaj odgovor
        ako korisnik kaže ne // #5
        promeni go_for_it u false
      }
   }
}
```

Uslovne naredbe

Uslovni izraz naredbe `if` mora da bude u zagradama. Ako je tačan, izvršava se naredba koja neposredno sledi za naredbom `if`:

```
// #5
if (  user_rps == 'N' || usr_rsp == 'n' )
      go_for_it = false;
```

Ako treba da se izvrši više naredbi, one moraju da se stave u vitičaste zagrade iza naredbe `if` (to se zove *blok naredbi*):

```
// #3
if ( usr_guess == next_elem )
{ // počinje blok naredbi
       num_right++;
       got_to = true;
} // kraj bloka naredbi
```

Početnici često greše i zaboravljaju na blok naredbi:

```
// greška: nedostaje blok naredbi
// samo je num_cor++ deo naredbi
// got_it = true; izvršava se bezuslovno

if (  usr_guess == next_elem )
      num_cor++;
      got_it = true;
```

Uvlačenje reda sa `got_it` pokazuje šta je programer hteo. Nažalost, to ne odražava ponašanje programa. Inkrementiranje objekta `num_cor` je povezano sa naredbom `if` i izvršava se samo kada je korisnikov odgovor jednak vrednosti objekta `next_elem`. Međutim, promena objekta `got_it` nije povezana sa naredbom `if` zato što smo zaboravili da dve naredbe uključimo u blok naredbi. Objekat `got_it` će nakon ovoga primera uvek biti tačno, bez obzira kakav je korisnikov odgovor.

Naredba `if` takođe podržava i jednu klauzulu `else`. Klauzula `else` predstavlja jedan ili više blokova naredbi koje će se izvršiti ako je ispitivani uslov netačan. Na primer,

```
if ( usr_guess == next_elem )
{
   // korisnikov odgovor tačan
}
else
{
   // korisnikov odgovor netačan
}
```

Druga upotreba klauzule `else` je da naniže dve ili više naredbi `if`. Na primer, kada korisnik odgovori netačno, želimo da naš odgovor bude drugačiji u zavisnosti od toga koliko je puta korisnik pogađao. Mogli bismo da napišemo tri provere kao nezavisne naredbe `if`:

```
if ( num_tries == 1 )
     cout << "Greška! Nije baš tako.\n";

if ( num_tries == 2 )
     cout << "Žalim. Opet nije tačno.\n";

if ( num_tries == 3 )
     cout << "Ovo je teže nego što se činilo, zar ne?\n";
```

Međutim, u datom trenutku samo jedan od tri uslova može da bude tačan. Ako je jedna od `if` naredbi tačna, ostale moraju da budu netačne. Odnos između naredbi `if` možemo da pokažemo nižući ih redom i međusobno ih povezujući nizom klauzula `else-if`:

```
if ( num_tries == 1 )
     cout << "Greška! Nije baš tako.\n";
else
if ( num_tries == 2 )
     cout << "Žalim. Opet nije tačno.\n";
else
if ( num_tries == 3 )
     cout << "Ovo je teže nego što se činilo, zar ne?\n";
else
     cout << "Sada se sigurno nervirate!\n";
```

Vrednuje se uslov prve `if` naredbe. Ako je on tačan, izvršava se naredba koja sledi za njim, a klauzule `else-if`, koje slede, ne uzimaju se u obzir. Ako je uslov prve naredbe `if` netačan, vrednuje se sledeći, pa sledeći, itd, sve dok neki uslov ne bude tačan, ili, ako je `num_tries` veće od 3, svi uslovi su netačni i izvršava se poslednja klauzula `else`.

Zbunjujući aspekt ugneždenih klauzula `if-else` je taj što je teško ispravno organizovati njihovu logiku. Na primer, želeli bismo da našu naredbu `if-else` koristimo da podelimo programsku logiku na dva slučaja: kada je korisnikov odgovor tačan i kada korisnikov odgovor nije tačan. Prvo rešenje ne radi baš onako kako smo želeli:

```
if ( usr_guess == next_elem )
{
   // korisnikov odgovor tačan
}
else
if ( num_tries == 1 )
   //... ispisati odgovor
else
if ( num_tries == 2 )
   //... ispisati odgovor
else
if ( num_tries == 3)
   //... ispisati odgovor
else
   //... ispisati odgovor
```

```
   // sada pitati korisnika da li želi da nastavi
   // ali samo ako je dao pogrešan odgovor
   // gde to možemo da smestimo?
```

Sve klauzule `else-if` su nenamerno postale alternativa za ispravno pogođenu vrednost. To je dovelo do toga da nemamo gde da stavimo drugi deo koda koji bi rešavao situaciju kada korisnik da pogrešan odgovor. Evo kako izgleda ispravno organizovan program:

```
if (  usr_guess == next_elem )
{
     // korisnikov odgovor tačan
}
else
{  // korisnikov odgovor netačan
   if (  num_tries == 1 )
         // ...
   else
   if (  num_tries == 2 )
         // ...
   else
   if (  num_tries == 3 )
         // ...
   else // ...

   cout << "Želite li da pokušate ponovo? (D/N) ";
   char usr_rsp;
   cin >> usr_rsp;
   if (  usr_rsp == 'N' || usr_rsp == 'n' )
         go_for_it = false:
}
```

Ako je vrednost uslova koji se ispituje ceo broj, možemo da zamenimo skup klauzula `if-else-if` naredbom `switch`:

```
// ekvivalentno klauzulama if-else-if iz gornjeg primera
switch ( num_tries )
{
   case 1:
         cout << "Greška! Nije baš tako.\n";
         break;

   case 2:
         cout << "Žalim. Opet nije tačno.\n";
         break;

   case 3:
         cout << "Ovo je teže nego što se činilo, zar ne?\n";
         break;
```

```
    default:
          cout << "Mora da se sada već nervirate!\n";
          break;
}
```

Za ključnom reči `switch` sledi jedan izraz u zagradi (da, ime objekta može da posluži kao izraz). Izraz mora da dâ celobrojnu vrednost. Niz labela `case` sledi za ključnom reči `switch`, od kojih svaka definiše konstantan izraz. Rezultat izraza se poredi sa svakom labelom `case`, redom. Ako se nađe odgovarajuća, izvršavaju se naredbe koje slede za `case`. Ako se ne nađe odgovarajuća i postoji labela `default`, izvršavaju se naredbe koje slede za njom. Ako se ne nađe odgovarajuća i ne postoji labela `default`, ništa se ne događa.

Zašto sam na kraju svake labele stavio naredbu `break`? Svaka labela `case` se redom poredi sa vrednošću izraza. Svaka neodgovarajuća labela `case` se redom preskače. Kada labela `case` odgovara vrednosti, izvršavanje počinje od naredbe koja sledi za labelom `case`. „Kvaka" je u tome što se izvršavaju i sve naredbe `case` koje slede za njom, sve dok se ne dođe do kraja naredbe `switch`. Kada bi `num_tries` bio jednak 2, na primer, i kad ne bi bilo naredbe `break`, izlaz bi izgledao ovako:

```
// izlaz kada je num_tries == 2 i kada smo
// zaboravili da stavimo naredbe break
Žalim. Opet nije tačno.
Ovo je teže nego što se činilo, zar ne?
Sada se sigurno nervirate!
```

Pošto se podudari jedna labela `case`, sve labele `case` koje slede za njom takođe se izvršavaju, ukoliko se izvršavanje eksplicitno ne prekine. Upravo to radi naredba `break`. Verovatno se pitate zašto je naredba `switch` tako napravljena? Evo slučaja kada je ovakvo ponašanje baš potrebno:

```
switch ( next_char )
{
    case 'a' :   case 'A' :
    case 'e' :   case 'E' :
    case 'i' :   case 'I' :
    case 'o' :   case 'O' :
    case 'u' :   case 'U' :
                 ++vowel_cnt;
                 break;
    // ...
}
```

Naredbe ponavljanja

Naredba ponavljanja izvršava naredbu ili blok naredbi sve dok je vrednost uslovnog izraza tačno. Za naš program su neophodne dve naredbe petlje, jedna ugneždena unutar druge:

```
dok korisnik želi da pogađa
{
      prikazati niz brojeva
      dok je odgovor pogrešan i korisnik želi ponovo da odgovara
}
```

C++ petlja `while` lepo zadovoljava naše potrebe:

```
bool next_seq = true; // prikazati sledeći niz?
bool go_for_it = true;// korisnik želi da pogađa?
bool got_it = false;  // korisnikov odgovor tačan?
int num_tries = 0;    // broj odgovora korisnika
int num_right = 0;    // broj tačnih odgovora

while (  next_seq == true )
{
      // korisniku prikazati niz
      while (( got_it == false ) &&
               ( go_for_it == true ))
      {
         int user_guess;
         cin >> usr_guess;
         num_tries++;
         if (  usr_guess == next_elem )
         {
               got_it = true;
               num_cor++;
         }
         else
         {     // korisnik odgovorio netačno
               // reći korisniku da je odgovor netačan
               // pitati korisnika želi li ponovo
               if (  usr_rsp == 'N' || usr_rsp == 'n' )
                     go_for_it = false;
         }
      } // kraj ugneždene petlje while

   cout << "Želite li da pokušate ponovo? (D/N) ";
   char try_again;
   cin >> try_again;

   if (  try_again == 'N' || try_again == 'n' )
         next _seq = false;

} // kraj while( next_seq == true )
```

Petlja `while` počinje vrednovanjem uslovnog izraza u zagradama. Ako je on tačan, izvršava se naredba ili blok naredbi koje slede za petljom `while`. Kada se naredba izvrši, izraz se ponovo vrednuje. Krug vrednovanje/izvršavanje se nastavlja dok izraz ne bude netačan. Obično neki uslov u bloku izvršnih naredbi