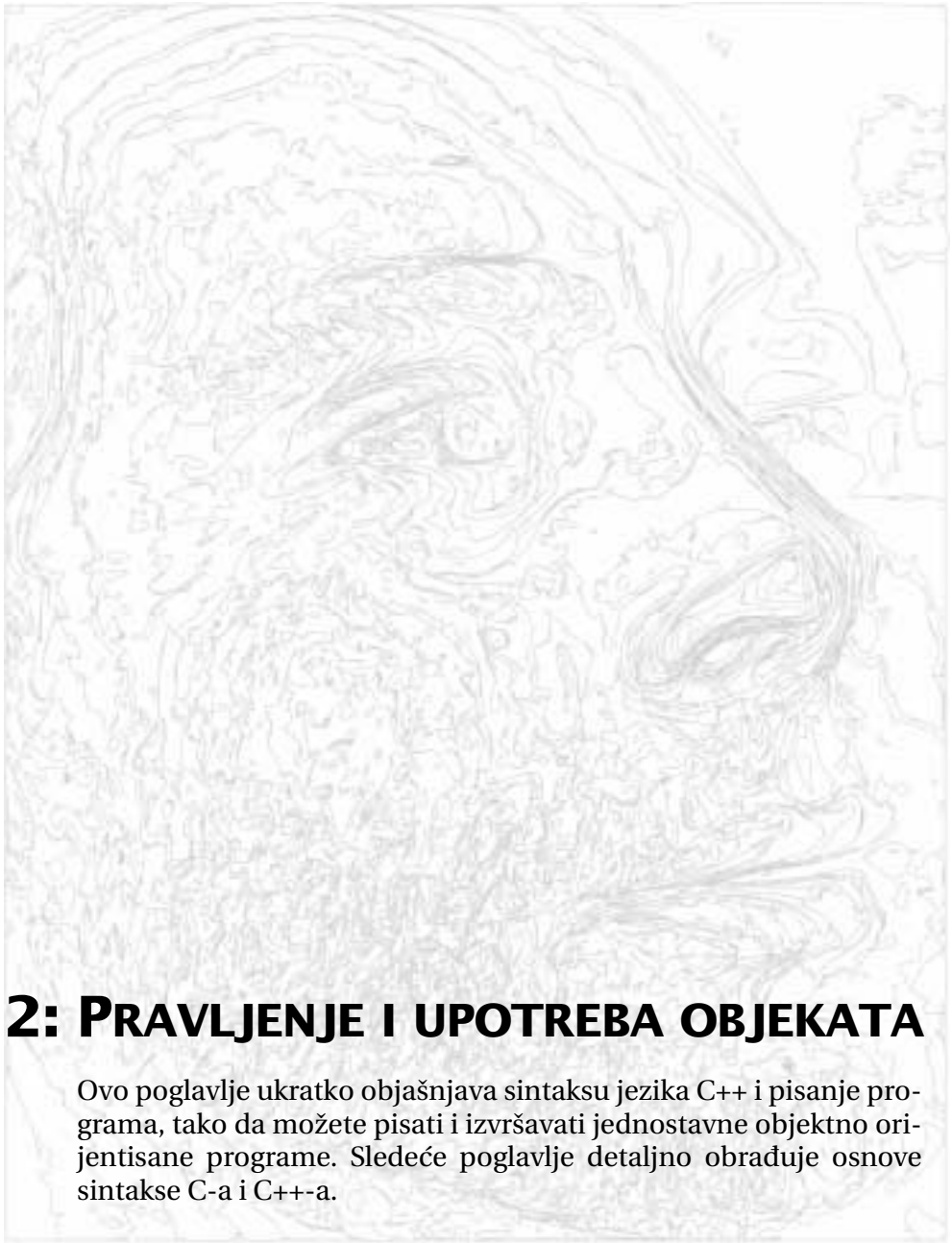

2: PRAVLJENJE I UPOTREBA OBJEKATA

Ovo poglavlje ukratko objašnjava sintaksu jezika C++ i pisanje programa, tako da možete pisati i izvršavati jednostavne objektno orijentisane programe. Sledeće poglavlje detaljno obrađuje osnove sintakse C-a i C++-a.

U ovom poglavlju steći ćete prve utiske o primeni objekata iz C++-a u programiranju, a otkrićete i neke razloge za popularnost ovog jezika. To bi trebalo da bude dovoljno da vas provede kroz poglavlje 3, koje može biti naporno pošto sadrži većinu detalja jezika C.

Tip podataka koji definiše korisnik, ili ukratko *klasa*, predstavlja glavnu razliku između C++-a i tradicionalnih proceduralnih jezika. Klasa je nov tip podataka koji pomaže pri rešavanju određene vrste problema. Napravljenu klasu koristiće i oni koji ne poznaju specifičnosti njenog rada ni način njene izrade. Ovo poglavlje se bavi klasama kao da su ugrađeni tip podataka koji se mogu koristiti u programima.

Klase drugih autora uglavnom su spakovane u biblioteku. Ovo poglavlje koristi nekoliko biblioteka klasa koje postoje u svim realizacijama C++-a. Naročito značajna standardna biblioteka je **iostream** koja, između ostalog, omogućava učitavanje iz datoteka i s tastature i ispisivanje u datoteke i na monitor. Videćete i veoma pogodnu klasu **string** i kontejner **vector** standardne C++ biblioteke. Do kraja ovog poglavlja, videćete koliko se jednostavno koriste gotove biblioteke klasa.

Pre nego što napišete prvi program, morate poznavati alatke koje se koriste za sklapanje programa.

Postupak prevođenja jezika

Svi računarski jezici se prevode iz oblika koji ljudi mogu lako razumeti (*izvorni kôd*, engl. *source code*) u nešto što računar izvršava (*mašinske instrukcije*, engl. *machine instructions*). Programi za prevođenje se tradicionalno dele na dve kategorije: *interpretatore* (engl. *interpreters*) i *prevodioce* (engl. *compilers*).

Interpretatori

Interpretator prevodi izvorni kôd u akcije (koje mogu obuhvatati skup mašinskih instrukcija) i neposredno ih izvršava. BASIC je, na primer, popularni jezik koji se interpretira. Tradicionalni BASIC interpretatori prevode i izvršavaju jedan po jedan red i zatim zaboravljaju da je taj red bio preveden. Kôd koji se ponavlja moraju ponovo prevesti i to ih čini sporim. Radi ubrzavanja napravljeni su i prevodioci za BASIC. Savremeniji interpretatori, na primer za jezik Python, prevode ceo program u međujezik koji zatim izvršava mnogo brži interpretator.[1]

Interpretatori imaju mnoge prednosti. Prelaz od pisanja do izvršavanja koda je skoro trenutan, a dostupan je i izvorni kôd, tako da interpretator može mnogo preciznije opisivati grešku. Često se navodi da su dobiti od korišćenja interpretatora jednostavnost interakcije i brzi razvoj (ne obavezno i izvršavanje) programa.

Interpretirani jezici su često ozbiljno ograničeni kada se koriste za izgradnju velikih projekata (izgleda da je Python izuzetak). Interpretator (ili njegova ograničena verzija) mora uvek biti u memoriji za vreme izvršavanja programa, a čak

[1] Granica između prevodilaca i interpretatora postala je nejasna, naročito u jeziku Python. Programi na tom jeziku imaju mnoga svojstva i moć prevedenih programa, ali i brzi odziv interpretiranih.

i najbrži interpretator može izazvati neprihvatljivo usporavanje programa. Većina interpretatora zahteva unošenje kompletnog izvornog koda. To povećava potrebni prostor, a može izazvati i ozbiljne greške ako jezik ne omogućava lokalizaciju efekata različitih delova koda.

Prevodioci

Prevodilac neposredno prevodi izvorni kôd na asemblerski jezik ili u mašinske instrukcije. Konačni proizvod je jedna ili više datoteka mašinskog koda. To je složen proces i obično se sastoji od nekoliko koraka. Put od pisanja do izvršavanja koda uz korišćenje prevodioca značajno je duži.

U zavisnosti od veštine autora prevodioca, prevedeni programi obično staju na manje prostora i mnogo su brži. Iako su veličina i brzina verovatno najčešće navođeni razlozi za korišćenje prevodioca, u mnogim situacijama nisu i najvažniji. Neki jezici (kao što je C) projektovani su tako da dozvoljavaju nezavisno prevođenje delova programa. Ovi delovi se kasnije povezuju u konačni *izvršni* (engl. *executable*) program pomoću alatke koja se zove *povezivač* (engl. *linker*). Ovaj proces se naziva *odvojeno prevođenje* (engl. *separate compilation*).

Odvojeno prevođenje ima mnoge prednosti. Program koji je suviše velik i ne bi mogao odjednom da se prevede zbog ograničenja prevodioca ili okruženja za prevođenje, može se prevoditi u delovima. Jedan po jedan deo programa se može izgrađivati i ispitivati. Ispravan deo se može sačuvati i koristiti kao gradivni blok. Skupovi testiranih radnih delova se mogu organizovati u *biblioteke* i staviti na raspolaganje drugim programerima. Prilikom pravljenja jednog dela, skrivena je složenost drugih delova. Sve to omogućuje pisanje velikih programa.[2]

Mogućnosti prevodioca da pronađu greške vremenom su značajno poboljšane. Prvobitni prevodioci su samo generisali mašinski kôd, a programer je dodavao naredbe koje ispisuju međurezultate kako bi uvideo šta se dešava. To nije uvek efikasno. Savremeni prevodioci mogu umetati informacije o izvornom kodu u izvršni program. Ove informacije koriste moćni *programi za otkrivanje grešaka u izvornom kodu* (engl. *source-level debuggers*) koji prikazuju šta se tačno dešava tako što prate izvršavanje kroz izvorni kôd.

Neki prevodioci rešavaju problem brzine *prevođenjem unutar memorije* (engl. *in-memory compilation*). Većina prevodilaca radi s datotekama, čitaju i upisuju u svakom koraku procesa prevođenja. Prilikom prevođenja unutar memorije, prevodilac je u RAM-u. U malim programima, ovo može biti efikasno kao što bi bio i interpretator.

Postupak prevođenja

Da biste programirali na C-u ili C++-u, treba da razumete korake i sredstva postupka prevođenja. Neki jezici (konkretno, C i C++) započinju prevođenje propuštanjem izvornog koda kroz *pretprocesor* (engl. *preprocessor*). Pretprocesor je jednostavan program koji zamenjuje šablone izvornog programa drugim šablonima koje je programer definisao (upotrebom *pretprocesorskih komandi*, engl. *preprocessor directives*). Pretprocesorske komande se koriste da bi se manje

[2] Python je ponovo izuzetak, pošto takođe podržava odvojeno prevođenje.

kucalo i da bi se povećala čitljivost koda. (U nastavku knjige ćete naučiti da se pri pisanju C++ programa ne preporučuje korišćenje pretprocesora, pošto može izazvati teško uočljive greške.) Pretprocesirani kôd se često smešta u međudatoteku.

Prevodioci obično rade u dva prolaza. Prvi prolaz *analizira sintaksu* (engl. *parsing*) pretprocesiranog koda. Prevodilac deli izvorni kôd u male celine i organizuje ih u strukturu *stabla*. U izrazu **„A+B“** elementi **'A'**, **'+'** i **'B'** su listovi sintaksnog stabla.

Ponekad se između prvog i drugog prolaza koristi *globalni optimizator* (engl. *global optimizer*) , da bi se napravio manji i brži program.

U drugom prolazu, *generator koda* (engl. *code generator*) prolazi kroz sintaksno stablo i na osnovu čvorova stabla generiše ili asemblerski ili mašinski kôd. Ako generiše asemblerski kôd, neophodno je da se on kasnije prevede asemblerom u mašinski kôd. U oba slučaja je krajnji rezultat objektni modul (datoteka tipa **.o** ili **.obj**). U drugom prolazu se ponekad koristi *lokalni optimizator* (engl. *peephole optimizer*) koji traži delove koda sa suvišnim asemblerskim naredbama.

Pogrešno je rečju „objektni“ opisivati delove mašinskog koda. Reč je ušla u upotrebu pre objektno orijentisanog programiranja. Kada se razmatra prevođenje, „objekat“ znači „cilj“, dok u objektno orijentisanom programiranju predstavlja „stvar sa definisanim granicama“.

Povezivač (engl. *linker*) spaja listu objektnih modula u izvršni program koji operativni sistem može učitati u memoriju i izvršiti. Kada se neka funkcija u jednom objektnom modulu obraća funkciji ili promenljivoj u drugom objektnom modulu, povezivač razrešava ove reference – na taj način se tokom prevođenja proverava postoje li sve spoljne funkcije i podaci. Povezivač dodaje i posebni objektni modul koji obavlja inicijalne aktivnosti.

Povezivač može pretraživati posebne datoteke, pod nazivom *biblioteke* (engl. *libraries*) kako bi razrešio sve reference. Biblioteka predstavlja skup objektnih modula u jednoj datoteci. *Program za upravljanje bibliotekama* (engl. *librarian*) pravi i održava biblioteke.

Statička provera tipova

Prevodilac proverava tipove u prvom prolazu. Proverom tipova ispituje da li se pravilno koriste argumenti u funkcijama i sprečavaju se mnoge vrste programskih grešaka. Pošto se tipovi proveravaju tokom prevođenja umesto za vreme izvršavanja programa, provera tipova se naziva *statička* (engl. *static type checking*).

Neki objektno orijentisani jezici (posebno Java) proveravaju tipove tokom izvršavanja programa. To je *dinamička provera tipova* (engl. *dynamic type checking*). Statička provera tipova je mnogo moćnija kada se kombinuje s dinamičkom proverom, ali takva kombinacija usporava izvršavanje programa.

Prevodioci jezika C++ statički proveravaju tipove, pošto jezik ne omogućava nikakvu podršku za dinamičku obradu takvih grešaka tokom izvršavanja. Statička provera tipova obaveštava programera o pogrešnim upotrebama tipova tokom prevođenja i tako maksimalno ubrzava izvršavanje. Dok budete učili C++, videćete da većina svojstava jezika podržava istu vrstu brzog, produktivnog programiranja po kome je poznat C.

U C++-u možete zaobići statičku proveru tipova. Možete i sami dinamički proveravati tipove, samo treba da napišete odgovarajući kôd.

Alatke za odvojeno prevođenje

Odvojeno prevođenje je naročito značajno pri izgradnji velikih projekata. Program na jezicima C i C++ se može napraviti od malih i preglednih delova koji se nezavisno testiraju. Osnovno sredstvo za podelu programa na delove je mogućnost pravljenja imenovanih potprograma. Potprogram se u C-u i C++-u naziva *funkcija,* a funkcije su delovi koda koji se mogu smestiti u različite datoteke, što omogućava odvojeno prevođenje. Drugim rečima, funkcija je nedeljiva jedinica koda, pošto se ne može jedan deo funkcije nalaziti u jednoj, a drugi u nekoj drugoj datoteci – potpuna funkcija mora biti u jednoj datoteci (datoteka može da sadrži više od jedne funkcije).

Prilikom poziva funkcije, obično joj se prosleđuju *argumenti* (engl. *arguments*), a to su vrednosti s kojima funkcija radi tokom izvršavanja. Po završetku funkcije, obično dobijate *rezultat* ili *povratnu vrednost* (engl. *return value*). Moguće je pisati i funkcije kojima se ne prosleđuju argumenti i funkcije koje ne vraćaju rezultat.

Da bi se napravio program koji se sastoji od više datoteka, funkcije iz jedne datoteke moraju pristupati funkcijama i podacima u drugim datotekama. Prilikom prevođenja datoteke, C ili C++ prevodilac moraju znati koje se funkcije i podaci nalaze u drugim datotekama, konkretno njihova imena i način upotrebe. Ovaj postupak „saopštavanja“ prevodiocu imena spoljnih funkcija i podataka i načina njihovog korišćenja naziva se *deklarisanje*. Kada deklarišete funkciju ili promenljivu, prevodilac zna kako da proveri da li se ona pravilno koristi.

Deklaracije ili definicije

Važno je razumeti razliku između *deklaracija* (engl. *declaration*) i *definicija* (engl. *definitions*), zato što će se ovi termini koristiti u celoj knjizi. Bitno je da u svim programima na jezicima C i C++ morate deklarisati promenljive i funkcije. Da biste mogli da napišete prvi program, treba da znate kako se piše deklaracija.

Deklaracija prevodiocu saopštava ime (identifikator). Ona kaže prevodiocu: „Ova funkcija ili ova promenljiva negde postoji i ovako treba da izgleda“. *Definicija* kaže: „Napravi ovde ovu promenljivu“ ili „Napravi ovde ovu funkciju“. Ona rezerviše prostor za to ime. Ovo važi, bilo da se radi o promenljivoj ili funkciji. U svakom slučaju, prevodilac rezerviše prostor kada naiđe na definiciju. Prevodilac određuje veličinu promenljive i obezbeđuje memorijski prostor u koji će biti smeštena njena vrednost. Za funkciju, prevodilac pravi kôd čime se završava zauzimanje memorijskog prostora.

Promenljiva ili funkcija se mogu deklarisati na više različitih mesta, ali mora postojati samo jedna definicija koja odgovara tim deklaracijama u programima na jezicima C-u i C++-u, što se ponekad naziva *pravilo jedne definicije* (engl. *one-definition rule,* ODR). Povezivač obično prijavljuje grešku ako tokom objedinjavanja svih objektnih modula pronađe više od jedne definicije iste funkcije ili promenljive.

Definicija može biti i deklaracija. Ako prevodilac nije prethodno naišao na ime **x**, a definišete **int x;** tada prevodilac ime vidi kao deklaraciju i istovremeno rezerviše prostor za promenljivu.

Sintaksa deklaracije funkcije

Deklaracija funkcije u C-u i C++-u dodeljuje funkciji ime, tipove argumenata koji se prosleđuju funkciji i tip rezultata funkcije. Na primer, ovo je deklaracija funkcije nazvane **func1()** koja ima dva celobrojna argumenta (rezervisana reč **int** označava cele brojeve) i vraća ceo broj:

```
int func1(int,int);
```

Prva rezervisana reč koju vidite je tip rezultata: **int**. Argumenti se nalaze između zagrada iza imena funkcije i poređani su po redosledu kojim se koriste. Znak tačka i zarez ukazuje na kraj naredbe, a u ovom slučaju saopštava prevodiocu: „To je sve, ovde nema definicije funkcije!“

Deklaracije u C-u i C++-u oponašaju način korišćenja elementa. Na primer, ako je **a** ceo broj, tada se gornja funkcija može koristiti na sledeći način:

```
a = func1(2,3);
```

Pošto je rezultat **func1()** ceo broj, prevodilac će proveriti kako se koristi **func1()** da bi se uverio da **a** može prihvatiti rezultat i da su argumenti odgovarajući.

Argumenti u deklaraciji funkcije mogu imati imena. Prevodilac zanemaruje imena, ali ona mogu objasniti namenu argumenata korisniku. Na primer, možemo deklarisati **func1()** na drugi način, a da značenje ostane isto:

```
int func1(int duzina, int sirina);
```

Zamka

Funkcije s praznim listama argumenata nisu iste u C-u i C++-u. U C-u, deklaracija:

```
int func2();
```

znači „funkcija s proizvoljnim brojem i tipovima argumenata“. Ovo sprečava proveru tipova, tako da u C++-u to znači „funkcija bez argumenata“.

Definicija funkcije

Definicija funkcije liči na deklaraciju funkcije, ali sadrži i telo. Telo je skup naredaba između vitičastih zagrada. Vitičaste zagrade označavaju početak i kraj bloka koda. Da bismo definisali funkciju **func1()** koja ima prazno telo (telo koje ne sadrži kôd), pišemo:

```
int func1(int duzina, int sirina) { }
```

Zapazite da vitičaste zagrade u definiciji funkcije zamenjuju tačku i zarez. Pošto vitičaste zagrade okružuju jednu ili više naredaba, nije potreban znak tačka i zarez. Obratite pažnju i na to da argumenti u definiciji funkcije moraju imati imena ako ih koristimo u telu funkcije (ovde su opciona, pošto se ne koriste).

Sintaksa deklaracije promenljive

Značenje fraze „deklaracija promenljive“ dugo je zbunjivalo zbog kontradiktornosti, a bitno je razumeti pravu definiciju da bi se kôd pravilno čitao. Deklaracija promenljive saopštava prevodiocu kako ta promenljiva izgleda. Ona tvrdi: „Ovo ime se prvi put pojavljuje, ali garantujem da negde postoji i da je to promenljiva tipa X“.

U deklaraciji funkcije, navode se tip rezultata, ime funkcije, lista argumenata i znak tačka i zarez. Prevodiocu je ovo dovoljno da otkrije da se radi o deklaraciji i kako funkcija treba da izgleda. Može se zaključiti da je deklaracija promenljive tip iza koga se navodi ime. Na primer, po toj logici

```
int a;
```

može biti deklaracija celobrojne promenljive **a**. Problem je sledeći: gornji kôd sadrži dovoljno informacija potrebnih prevodiocu da obezbedi prostor za ceo broj sa imenom **a** i to se i dešava. Da bi se ovaj problem rešio, C-u i C++-u je bila neophodna rezervisana reč koja znači: „Ovo je samo deklaracija, a definicija se nalazi na drugom mestu“. Ta rezervisana reč je **extern** (spoljni) i znači da se definicija nalazi izvan datoteke ili iza tog mesta u istoj datoteci.

Promenljiva se deklariše bez definisanja pomoću rezervisane reči **extern** koja se navodi ispred opisa promenljive:

```
extern int a;
```

extern se može primeniti i na deklaracije funkcija. Za **func1()** bismo pisali:

```
extern int func1(int duzina, int sirina);
```

Ova naredba je ekvivalentna prethodnim deklaracijama **func1()**. Pošto ne postoji telo funkcije, prevodilac prepoznaje naredbu kao deklaraciju, a ne kao definiciju funkcije. Rezervisana reč **extern** je suvišna u deklaracijama funkcija. Projektanti C-a, nažalost, nisu zahtevali da se koristi **extern** u deklaracijama funkcija, pošto bi se poboljšala konsistentnost i uklonile nedoumice (to što bi se više kucalo verovatno objašnjava ovu odluku).

Ovo su još neki primeri deklaracija:

```
//: C02:Declare.cpp
// Primeri deklaracija i definicija
extern int i; // Deklaracija bez definicije
extern float f(float); // Deklaracija funkcije
float b;  // Deklaracija i definicija
float f(float a) {  // Definicija
  return a + 1.0;
}

int i; // Definicija
int h(int x) { // Deklaracija i definicija
  return x + 1;
}
```

```
int main() {
  b = 1.0;
  i = 2;
  f(b);
  h(i);
} ///:~
```

Identifikatori argumenata nisu obavezni u deklaraciji funkcije. U definicijama su neophodni samo u C-u, a u C++-u nisu.

Uključivanje zaglavlja

Većina biblioteka sadrži značajan broj funkcija i promenljivih. Da bi se smanjila količina posla i obezbedila konsistentnost prilikom spoljnog deklarisanja ovih elemenata, uvedena je *datoteka zaglavlja* (engl. *header file*). Datoteka zaglavlja sadrži spoljne deklaracije za neku biblioteku, a nastavak njenog imena obično je 'h', na primer **headerfile.h**. (U nekim starijim programima možete naići na drugačije oznake tipa, kao što su **.hxx** ili **.hpp**, ali one su sve ređe.)

Datoteku zaglavlja piše autor biblioteke. Korisnik uključuje datoteku zaglavlja da bi deklarisao funkcije i spoljne promenljive biblioteke. Pretprocesorska komanda **#include** uključuje datoteku zaglavlja. Time se pretprocesoru nalaže da otvori navedenu datoteku zaglavlja i umetne njen sadržaj tamo gde se nalazi naredba **#include**. Ime datoteke u naredbi **#include** može se zadati na dva načina: između znakova < i > ili između znakova navoda.

Ako je ime datoteke između znakova < i >, kao u slučaju:

```
#include <zaglavlje>
```

pretprocesor traži datoteku na način specifičan za konkretnu realizaciju, ali obično postoji „putanja za traženje datoteka zaglavlja", koju definišete u programskom okruženju ili na komandnoj liniji. Način podešavanja putanje je različit za pojedine računare, operativne sisteme i realizacije C++-a, pa detaljnije informacije potražite sami u odgovarajućoj dokumentaciji.

Ime datoteke između navodnika, kao u slučaju:

```
#include "local.h"
```

nalaže pretprocesoru da traži datoteku na „način definisan realizacijom prevodioca" (prema specifikaciji). To najčešće znači da se datoteka traži relativno u odnosu na tekući direktorijum. Ako se ne pronađe, naredba se ponovo obrađuje kao da su upotrebljeni znaci < i > umesto znakova navoda.

Da bi se uključila datoteka zaglavlja iostream, piše se:

```
#include <iostream>
```

Pretpocesor će pronaći datoteku zaglavlja iostream (često u poddirektorijumu include) i umetnuti je.

Standardni C++ format naredbe uključivanja

Tokom razvoja C++-a, proizvođači prevodilaca su birali različite nastavke imena datoteka. Osim toga, u nekim operativnim sistemima ograničena je dužina

imena datoteka. Ovi zahtevi su doveli do problema pri prenošenju izvornog koda. Da bi se problem ublažio, propisan je standardni format koji dozvoljava da ime datoteke bude duže od čuvenih osam znakova i eliminiše nastavak imena. Na primer, umesto starog načina da se zaglavlje uključi **iostream.h**:

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

sada se može pisati:

```
#include <iostream>
```

Prevodilac može realizovati naredbe uključivanja na način koji odgovara njemu i operativnom sistemu, po potrebi skraćujući ime i dodajući nastavak. Naravno, proizvođačeve datoteke zaglavlja možete kopirati u datoteke bez nastavka imena, ukoliko hoćete da koristite ovu tehniku pre nego što je proizvođač vašeg prevodioca bude podržao.

Biblioteke nasleđene iz C-a još uvek su dostupne s tradicionalnim nastavkom imena – **'.h'**. Međutim, možete ih koristiti na savremeniji način dodajući **„c"** ispred imena. Tako

```
#include <stdio.h>
#include <stdlib.h>
```

postaje

```
#include <cstdio>
#include <cstdlib>
```

To važi za sve standardne datoteke zaglavlja C-a. Na ovaj način, čitalac može da prepozna kada se koriste C a kada C++ biblioteke.

Efekat novog formata uključivanja nije identičan sa starim: kada se koristi **.h** dobija se starija verzija, a kada se izostavi **.h** dobija se nova, šablonska verzija. Ako kombinujete ova dva oblika u istom programu, imaćete probleme.

Povezivanje

Povezivač (engl. *Linker*) objedinjuje objektne module (čiji nastavak imena je često **.o** ili **.obj**), koje je napravio prevodilac, u jedan izvršni program koji operativni sistem može učitati i izvršiti. To je poslednja faza procesa prevođenja.

Karakteristike povezivača razlikuju se od sistema do sistema. U opštem slučaju, povezivaču se navode imena objektnih modula i biblioteka koje treba povezati i ime izvršnog programa. Neki sistemi zahtevaju da sami aktivirate povezivač. Većina C++ prevodilaca automatski aktivira povezivač, pa u mnogim situacijama nije uočljivo pozivanje povezivača.

Neki stariji povezivači će samo jednom tražiti objektne datoteke i biblioteke, a lista se pretražuje sleva udesno, pa redosled navođenja objektnih datoteka i biblioteka može biti značajan. Ako se tek u trenutku povezivanja pojavio neki čudan problem, jedan od uzroka može biti redosled kojim su datoteke prosleđene povezivaču.

Upotreba biblioteka

Pošto ste ovladali osnovnom terminologijom, razumete kako treba koristiti biblioteku. Da biste upotrebili biblioteka, treba da:

1. Uključiti datoteku zaglavlja biblioteke.
2. Koristite funkcije i promenljive sadržane u biblioteci.
3. Povežete biblioteku i objektni kôd u izvršni program.

Ovi koraci se primenjuju i ako objektni moduli nisu smešteni u biblioteku. Uključivanje datoteke zaglavlja i povezivanje objektnih modula osnovni su koraci odvojenog prevođenja i u C-u i u C++-u.

Kako povezivač pretražuje biblioteku

Kada povezivač naiđe na spoljnu referencu funkcije ili promenljive može uraditi dve stvari. Ako nije prethodno našao definiciju te funkcije ili promenljive, dodaje identifikator svojoj listi „nerazrešenih referenci“. Ukoliko je povezivač ranije našao definiciju, referenca se razrešava.

Ako povezivač ne može da pronađe definiciju u objektnim modulima iz liste, tada pretražuje biblioteke. Biblioteke obično imaju i indeks sadržaja, pa povezivač ne mora da pretražuje sve objektne module biblioteke, nego samo indeks. Kada povezivač pronađe definiciju u biblioteci, u izvršni program se povezuje celokupni objektni modul, a ne samo definicija funkcije. Zapazite da se ne povezuje kompletna biblioteka, nego samo objektni modul koji sadrži potrebnu definiciju (inače bi programi bili nepotrebno veliki). Ako hoćete da smanjite izvršni program, razmotrite mogućnost da jedna datoteka izvornog koda u biblioteci sadrži samo jednu funkciju. Ovo zahteva više izmena,[3] ali može pomoći korisniku.

Pošto povezivač traži datoteke redosledom koji navedete, sprečićete korišćenje bibliotečke funkcije ako datoteku s vašom istoimenom funkcijom navedete pre imena biblioteke. Pošto će povezivač razrešiti sve odgovarajuće reference korišćenjem vaše funkcije pre nego što pretraži biblioteku, vaša funkcija se koristi umesto one iz biblioteke. Obratite pažnju na to da ovaj način može izazvati i grešku koja se sprečava upotrebom imenskih prostora C++-a.

Skriveni dodaci

Tokom pravljenja izvršnog programa, povezuju se i neki skriveni elementi. Jedan od njih je početni modul s kodom za inicijalizaciju, koji se mora svaki put izvršiti na početku rada C ili C++ programa. Ovaj kôd definiše stek i inicijalizuje neke programske promenljive.

Povezivač u standardnoj biblioteci uvek traži prevedene verzije „standardnih“ funkcija koje se pozivaju u programu. Pošto se standardna biblioteka uvek pretražuje, možete koristiti bilo šta iz te biblioteke tako što ćete u program uključiti odgovarajuću datoteku zaglavlja – ne morate posebno zahtevati povezivanje standardne biblioteke. Na primer, funkcije ulazno-izlaznih tokova su u

[3] Preporučio bih da koristite Perl ili Python za automatizaciju ovog zadatka koji je deo procesa pakovanja biblioteka (videti www.Perl.org ili www.Python.org).

standardnoj C++ biblioteci. Da biste ih koristili, dovoljno je da uključite datoteku zaglavlja **<iostream>**.

Ako koristite dodatnu biblioteku, povezivaču morate izričito navesti njeno ime.

Korišćenje uobičajenih C biblioteka

To što pišete program na C++-u ne sprečava vas da koristite funkcije C biblioteka. U stvari, podrazumeva se da je kompletna C biblioteka uključena u standardni C++. Ove funkcije su veoma korisne i mogu vam uštedeti dosta vremena.

U ovoj knjizi će se koristiti funkcije standardnih C++ (takođe i standardnih C) biblioteka, i to isključivo *standardne* funkcije, da bi se obezbedila prenosivost programa. U malobrojnim primenama funkcija biblioteka koje nisu po C++ standardu, koristiće se funkcije usaglašene sa standardom POSIX. POSIX je standard Unixa, i obuhvata funkcije izvan C++ biblioteka. Uglavnom možete očekivati da nađete POSIX funkcije na Unix (posebno Linux) platformama, a često i pod DOS/Windows sistemima. Na primer, ako pravite višenitne programe, imaćete koristi od POSIX biblioteke, pošto će vaš kôd biti razumljiviji i lakše će se prenositi i održavati (POSIX biblioteka za višenitno programiranje najčešće će samo koristiti odgovarajuće funkcije operativnog sistema).

Vaš prvi C++ program

Stekli ste dovoljno osnovnog znanja, pa možete da napišete i prevedete program. Program će koristiti standardne C++ klase ulazno-izlaznih tokova. One čitaju i upisuju u datoteke i „standardne“ ulazne i izlazne uređaje (obično je to konzola, ali se može preusmeriti na datoteke ili druge uređaje). U ovom jednostavnom programu, objekat toka će se koristiti za ispisivanje poruke na ekranu.

Upotreba klasa ulazno-izlaznih tokova

Da bi se deklarisale funkcije i spoljni podaci klase ulazno-izlaznih tokova, datoteka zaglavlja se uključuje naredbom:

```
#include <iostream>
```

Prvi program koristi standardni izlazni tok, što znači „mesto opšte namene za slanje izlaznih podataka“. Videćete i druge primere koji na različite načine koriste standardni izlazni tok, ali ćete u ovom slučaju koristiti konzolu. Paket iostream automatski definiše promenljivu (objekat) pod nazivom **cout**. Ona prima sve podatke koje treba poslati na standardni izlazni tok.

Za slanje podataka na standardni izlazni tok koristi se operator <<. Programeri na jeziku C znaju da ovaj operator označava pomeranje bitova ulevo (biće opisan u sledećem poglavlju). Dovoljno je reći da pomeranje bitova ulevo nema nikakve veze sa izlazom. Međutim, C++ omogućava *preklapanje*

operatora. Preklopljeni operator dobija novo značenje pri korišćenju sa objektom određenog tipa. Sa objektima ulazno-izlaznih tokova, operator << znači „poslati“. Na primer:

```
cout << "zdravo!";
```

šalje niz znakova „zdravo!“ objektu pod nazivom **cout** (to je skraćenica od „console output“, što znači „izlaz na konzolu“).

Za početak je dovoljno rečeno o preklapanju operatora. Poglavlje 12 se detaljno bavi tom temom.

Imenski prostori

Kao što je napomenuto u poglavlju 1, u jeziku C se nailazi na problem „iskorišćavanja svih imena“ funkcija i promenljivih kada program dostigne određenu veličinu. Naravno da nisu stvarno iskorišćena sva imena, nego posle izvesnog vremena postaje sve teže smišljati nova imena. Što je još važnije, kada program dostigne određenu veličinu, uobičajeno je da se podeli na manje celine koje razvijaju i održavaju različite osobe ili timovi. Pošto se u C programima sva imena nalaze u istom prostoru, programeri moraju paziti da slučajno ne upotrebe ista imena u situacijama kada bi to moglo biti dvosmisleno. Uskoro to postaje zamorno, troši se previše vremena, a i novca.

Standardni C++ ima mehanizam koji sprečava ove probleme: rezervisanu reč **namespace** (imenski prostor). Svaki skup C++ definicija u biblioteci, odnosno programu, „upakovan“ je u imenski prostor i neće biti nedoumice ako neka definicija ima isto ime u drugom imenskom prostoru.

Imenski prostori su pogodni i korisni, ali morate biti svesni njihovog postojanja pre nego što započnete pisanje programa. Ako samo uključite datoteku zaglavlja i koristite neke njene funkcije ili objekte, verovatno ćete pri prevođenju programa dobiti neobične poruke o greškama, čak i to da prevodilac ne može da pronađe deklaracije elemenata koje ste upravo uključili pomoću datoteke zaglavlja! Kada nekoliko puta vidite ovu poruku, postaće vam jasno njeno značenje (može se tumačiti kao: „Uključili ste datoteku zaglavlja, ali su sve deklaracije unutar nekog imenskog prostora, a niste saopštili prevodiocu da hoćete da koristite deklaracije iz tog imenskog prostora“).

Postoji rezervisana reč koja omogućava da kažete: „Hoću da koristim deklaracije i/ili definicije iz ovog imenskog prostora“. Ova rezervisana reč ima sasvim odgovarajući naziv **using** (koristeći). Sve standardne C++ biblioteke spakovane su u jedan imenski prostor, a to je **std** (od reči „standard“). Pošto se u ovoj knjizi koriste skoro isključivo standardne biblioteke, videćete sledeću *naredbu* **using** u skoro svakom programu:

```
using namespace std;
```

Ovo znači da hoćete da budu dostupni svi elementi imenskog prostora **std**. Posle ove naredbe ne morate razmišljati da li je konkretna komponenta biblioteke unutar imenskog prostora, pošto naredba **using** čini imena iz tog imenskog prostora raspoloživim u celoj datoteci.

Izlaganje svih elemenata jednog imenskog prostora, pošto je neko uložio veliki napor da ih sakrije, može delovati kontraproduktivno, i potreban je oprez, kao što ćete videti u nastavku ove knjige. Kako, naredba **using** stavlja na raspolaganje imena samo u datoteci u kojoj je napisana, opasnost nije velika kao što na prvi pogled izgleda. (Ipak, razmislite dvaput pre nego što iskoristite tu naredbu u datoteci zaglavlja – to nije dobar potez.)

Postoji veza između imenskih prostora i načina uključivanja datoteka zaglavlja. Pre nego što je standardizovan savremeni način uključivanja datoteka zaglavlja (bez pratećeg **.h**, kao u **<iostream>**), obično su se uključivale datoteke s nastavkom imena **.h**, kao u **<iostream.h>**. U to vreme nisu postojali ni imenski prostori. Da bi se obezbedila kompatibilnost sa starijim verzijama postojećih programa, ako napišete

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

to znači

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

U ovoj knjizi će biti korišćen standardni format uključivanja (bez **.h**), tako da se naredba **using** mora izričito navesti.

Zasad je to sve što je neophodno znati o imenskim prostorima, a u poglavlju 10 će ova tema biti detaljnije obrađena.

Osnovna struktura programa

Program na jeziku C ili C++ je skup promenljivih, definicija funkcija i poziva funkcija. Program prvo izvršava kôd za inicijalizaciju i poziva posebnu funkciju, **main()**. Tu se smešta osnovni kôd programa.

Kao što je spomenuto ranije, definicija funkcije se sastoji od tipa rezultata (mora biti definisan u C++-u), imena funkcije, liste argumenata između zagrada i tela funkcije unutar vitičastih zagrada. Ovo je primer definicije funkcije:

```
int funkcija() {
  // Ovo je telo funkcije (ovo je komentar)
}
```

Navedena funkcija ima praznu listu argumenata i telo koje sadrži samo komentar.

U definiciji funkcije može postojati više parova vitičastih zagrada, ali mora postojati najmanje jedan oko tela funkcije. Pošto je **main()** funkcija, ta pravila važe i za nju. Tip rezultata funkcije **main()** u C++-u je uvek **int**.

C i C++ su jezici slobodne forme. Osim malog broja izuzetaka, prevodilac ignoriše prelom redova i razmake, tako da mora postojati način da se prepozna kraj naredbe. Naredba se završava znakom tačka i zarez.

Komentari u C-u počinju sa /* i završavaju se sa */ i mogu se pisati u više redova. C++ koristi isti stil komentara, ali postoji još jedan tip komentara: //. Oznaka // započinje komentar koji se završava znakom za novi red. To je pogodnije nego /* */ za komentare u jednom redu i često se koristi u ovoj knjizi.

„Zdravo, svete!“

A sada, konačno, prvi program:

```
//: C02:Hello.cpp
// Pozdrav u C++-u
#include <iostream> // Deklaracije tokova
using namespace std;

int main() {
  cout << "Zdravo, svete! Danas imam "
       << 8 << " godina!" << endl;
} ///:~
```

Objektu **cout** se prosleđuje niz argumenata putem operatora <<. Objekat ispisuje ove argumente redom, sleva udesno. Posebna oznaka **endl** prouzrokuje prelazak na početak novog reda. Pomoću ulazno-izlaznih tokova možete na ovaj način povezati niz argumenata, što pojednostavljuje upotrebu klase.

U C-u je „string“ uobičajeni naziv za tekst između znakova navoda. Pošto standardna C++ biblioteka obuhvata moćnu klasu pod nazivom **string** za rad s tekstom,za tekst između navodnika koristiću precizniji termin *znakovni niz* .

Prevodilac rezerviše memorijski prostor za znakovne nizove i smešta ASCII vrednost svakog znaka u taj prostor. Na kraj znakovnog niza automatski se dodaje bajt vrednosti 0.

U znakovni niz se mogu umetati specijalni znaci korišćenjem *izlaznih sekvenci* (engl. *escape sequences*). One se sastoje od obrnute kose crte (\) posebnog znaka. Na primer, **\n** znači novi red. U nekom uputstvu za prevodilac ili priručniku za jezik C potražite potpun skup izlaznih sekvenci – neke od njih su: **t** (tabulator), \\ (obrnuta kosa crta) i **b** (jedan znak unazad).

Zapazite da se naredba može pisati u više redova, a da je tačka i zarez oznaka za kraj naredbe.

U naredbi se, pored objekta **cout**, pojavljuju i znakovni nizovi i konstantni brojevi. Pošto je operator << preklopljen više puta kada se koristi uz **cout**, objektu **cout** se mogu proslediti razni tipovi argumenata i on će „otkriti šta da uradi s porukom“.

Dok čitate ovu knjigu, primetićete da prvi red svake datoteke započinje oznakom komentara (obično //), iza kojeg slede dve tačke, a poslednji red programa sadrži oznaku komentara posle kojeg sledi **/:~**. Ovu tehniku koristim da bih omogućio jednostavno izdvajanje informacija iz datoteka s kodom (program koji ovo radi naći ćete u drugom tomu ove knjige). Prvi red sadrži i naziv i lokaciju datoteke, što koristim kada pominjem taj program u tekstu i u drugim datotekama. Na osnovu tih odrednica, program se lako može pronaći u izvornom kodu pridruženom ovoj knjizi.

Prevođenje

Posle prenošenja i raspakivanja izvornog koda iz knjige, pronađite program u poddirektorijumu **CO2**. Pozovite prevodilac navodeći **Hello.cpp** kao argument. U jednostavnim programima smeštenim u jednu datoteku, kao što je ovaj, većina prevodilaca će obaviti ceo proces. Na primer, ako koristite GNU C++ prevodilac (besplatno je dostupan na Internetu), napišite:

```
g++ Hello.cpp
```

Za ostale prevodioce koristi se slična sintaksa, a detalje potražite u odgovarajućoj dokumentaciji.

Više o ulazno-izlaznim tokovima

Dosad ste videli samo najosnovnije mogućnosti klasa ulazno-izlaznih tokova. Formatiranje izlaza pomoću ulazno-izlaznih tokova obuhvata i mogućnosti kao što su dekadno, oktalno i heksadecimalno formatiranje brojeva. Sledeći primer prikazuje upotrebu ulazno-izlaznih tokova:

```
//: CO2:Stream2.cpp
// Dodatne mogućnosti tokova
#include <iostream>
using namespace std;

int main() {
  // Definisanje formata pomoću manipulatora:
  cout << "broj u dekadnom zapisu: "
       << dec << 15 << endl;
  cout << "u oktalnom: " << oct << 15 << endl;
  cout << "u heksadecimalnom: " << hex << 15 << endl;
  cout << "u formatu pokretnog zareza: "
       << 3.14159 << endl;
  cout << "znak koji se ne prikazuje (escape): "
       << char(27) << endl;
} ///:~
```

Ovaj primer pokazuje ispisivanje brojeva pomoću klase ulazno-izlaznih tokova u dekadnom, oktalnom i heksadecimalnom formatu, primenom *manipulatora* koji ne ispisuju ništa, ali menjaju stanje izlaznog toka. Prevodilac automatski određuje formatiranje brojeva u pokretnom zarezu. Osim toga, bilo koji znak se može poslati objektu toka ako se konvertuje (engl. *cast*) u **char**, tip podataka koji sadrži jedan znak. Ovde se *konverzija* obavlja pozivanjem funkcije **char()**, uz ASCII vrednost znaka. U navedenom programu, **char(27)** šalje znak tastera „escape“ objektu **cout**.

Spajanje niza znakova

Važna osobina pretprocesora jezika C jeste *spajanje nizova znakova* (engl. *character array concatenation*). Ova mogućnost se koristi u nekim primerima u

knjizi. Ako su dva znakovna niza, uokvirena navodnicima, jedan pored drugog, bez znakova interpunkcije između njih, prevodilac će ih spojiti u jedan. Ovo je posebno korisno ako je ograničena širina programskog koda:

```
//: C02:Concat.cpp
// Spajanje znakovnih nizova
#include <iostream>
using namespace std;

int main() {
  cout << "Tekst je presirok da bi stao "
    "u jedan red, ali se moze podeliti "
    "bez posledica\nsamo ako nema "
    "znakova interpunkcije između susednih znakovnih "
    "nizova.\n";
} ///:~
```

Na prvi pogled, ovaj kôd deluje pogrešno, pošto nema tačke i zareza na kraju svakog reda. Prisetimo se da su C i C++ jezici slobodne forme. Iako ćete uglavnom videti tačku i zarez na kraju svakog reda, znak tačka i zarez mora stajati samo na kraju svake naredbe, a jedna naredba se može pisati u više redova.

Čitanje sa ulaznog toka

Klase ulazno-izlaznih tokova omogućavaju čitanje podataka. Za standardni ulazni tok se koristi objekat **cin** (skraćenica od „console input“, što znači „ulaz sa konzole“). Objekat **cin** podrazumevano očekuje ulaz sa konzole, ali se može preusmeriti na druge izvore. Primer preusmeravanja će biti naveden u nastavku poglavlja.

Uz objekat **cin** koristi se operator ulazno-izlaznih tokova, >>. On očekuje ulazne podatke koji po tipu odgovaraju njegovim argumentima. Na primer, ako mu prosledite celobrojni argument, on očekuje ceo broj sa konzole. Sledi primer:

```
//: C02:Numconv.cpp
// Konverzija dekadnog u oktalni i heksadecimalni broj
#include <iostream>
using namespace std;

int main() {
  int number;
  cout << "Unesite dekadni broj: ";
  cin >> number;
  cout << "oktalna vrednost = 0"
       << oct << number << endl;
  cout << "heksadecimalna vrednost = 0x"
       << hex << number << endl;
} ///:~
```

Ovaj program čita broj s tastature i konvertuje ga u oktalni i heksadecimalni zapis.

Pozivanje drugih programa

Bilo koji program se može pozvati i iz C ili C++ programa primenom standardne C funkcije **system()**, deklarisane u datoteci zaglavlja <**cstdlib**>:

```
//: C02:CallHello.cpp
// Pozivanje drugog programa drugog programa
#include <cstdlib> // Deklariše "system()"
using namespace std;
int main() {

  system("zdravo");
} ///:~
```

Funkciji **system()** treba proslediti znakovni niz koji biste uobičajeno otkucali na komandnoj liniji operativnog sistema. Mogu se dodati i argumenti komandne linije, a znakovni niz može biti napravljen i tokom izvršavanja programa (umesto statičkog niza znakova kakav je upotrebljen u primeru). Komanda se izvršava, a posle toga se izvršavanje programa nastavlja od mesta s koga je pozvana.

Ovaj program pokazuje koliko se jednostavno koriste uobičajene funkcije C biblioteka u C++-u: uključi se datoteka zaglavlja i pozove funkcija. Kompatibilnost C-a s novijim verzijama C++-a je velika prednost ako počinjete učenje C++-a već znate C.

Klasa string

Iako niz znakova može biti veoma koristan, sasvim je ograničen. To je samo grupa znakova u memoriji. Ako hoćete s njim da učinite bilo šta, morate se baviti mnoštvom sitnih detalja. Na primer, veličina niza znakova između navodnika je fiksirana u vreme prevođenja. Ako postojećem nizu znakova hoćete da dodate nove znakove, treba da znate mnogo stvari (uključujući dinamičko upravljanje memorijom, kopiranje niza znakova i spajanje). Upravo zato je potreban objekat koji će obaviti taj posao.

Standardna C++ klasa **string** napravljena je da bi se pobrinula (skriveno) o rukovanju nizovima znakova na niskom nivou, što je prethodno morao da radi C programer. Programeri na jeziku C su stalno gubili vreme na pisanje ovih operacija, a često su baš tu pravili i greške. Iako je u tomu 2 ove knjige celo poglavlje posvećeno klasi **string**, klasa je tako važna i toliko olakšava život, da ću je sada opisati i koristiti od samog početka.

Da biste koristili klasu **string**, treba da uključite datoteku zaglavlja <**string**>. Klasa **string** se nalazi u imenskom prostoru **std**, tako da je neophodno koristiti naredbu **using**. Usled preklapanja operatora, sintaksa korišćenja klase **string** je veoma intuitivna:

```
//: C02:HelloStrings.cpp
// Osnove standardne C++ klase string
#include <string>
#include <iostream>
using namespace std;
```

```
int main() {
  string s1, s2; // Prazni znakovni nizovi
  string s3 = "Zdravo, svete."; // Inicijalizovano
  string s4("Ja imam"); // Takodje inicijalizovano
  s2 = "godina danas"; // Dodeljivanje vrednosti znakovnom nizu
  s1 = s3 + " " + s4; // Kombinovanje znakovnih nizova
  s1 += " 8 "; // Dodavanje znakovnom nizu
  cout << s1 + s2 + "!" << endl;
} ///:~
```

Prva dva objekta tipa **string**, **s1** i **s2**, na početku su prazna, dok **s3** i **s4** prikazuju dva ekvivalentna načina inicijalizacije objekata tipa **string** nizovima znakova (isto tako jednostavno možete inicijalizovati objekte tipa **string** pomoću drugih objekata tipa **string**).

Bilo kom objektu tipa **string** može se dodeliti vrednost korišćenjem operatora =. Tako se vrednost objekta zamenjuje sadržajem sa desne strane i ne morate voditi računa o tome šta se dešava sa prethodnim sadržajem – to se rešava automatski. Za nadovezivanje objekata tipa **string** koristi se operator +, koji omogućava i nadovezivanje nizova znakova s objektima tipa **string**. Da biste nadovezali ili objekat tipa **string** ili niz znakova na drugi objekat tipa **string**, možete koristiti operator +=. Najzad, obratite pažnju na to da ulazno-izlazni tokovi već znaju šta treba da rade s objektima tipa **string**. Zato objekat tipa **string** (ili izraz tog tipa, kao u slučaju **s1 + s2 + !**) možete poslati direktno objektu **cout** da bi ga ispisao.

Učitavanje iz datoteka i upisivanje u njih

Proces otvaranja i rukovanja datotekama u C-u zahteva dosta predznanja da bi se mogle obaviti složene operacije. Međutim, biblioteka iostream omogućava jednostavno rukovanje datotekama, tako da se ove funkcije mogu uvesti mnogo ranije nego na jeziku C.

Da bi se datoteka otvorila za učitavanje i upisivanje, mora se uključiti zaglavlje <**fstream**>. Iako se time automatski uključuje zaglavlje <**iostream**>, dobar je potez da se <**iostream**> izričito uključi ako planirate da koristite **cin**, **cout** itd.

Da bi se datoteka otvorila za učitavanje, pravite objekat tipa **ifstream**, koji se ponaša slično objektu **cin**. Ako datoteku otvarate za upisivanje, pravite objekat tipa **ofstream**, koji se ponaša slično objektu **cout**. Pošto otvorite datoteku, možete učitavati iz nje i upisivati u nju kao što biste činili s bilo kojim drugim objektom toka. Eto kako je jednostavno (u tome je i poenta).

Jedna od najkorisnijih funkcija biblioteke iostream jeste **getline()**, koja učitava jedan red (završen znakom za novi red) u objekat tipa **string**[4]. Prvi argument je objekat tipa **ifstream** iz koga se učitava, a drugi argument je objekat tipa **string**. Po završetku funkcije, prosleđeni objekat tipa **string** sadržaće učitani red.

[4] Postoji više oblika funkcije **getline()**. Detaljno se razmatraju u poglavlju o ulazno-izlaznim tokovima, u drugom tomu.

Sledi jednostavan primer kopiranja sadržaja jedne datoteke u drugu:

```
//: C02:Scopy.cpp
// Kopiranje datoteke u drugu datoteku, red po red
#include <string>
#include <fstream>
using namespace std;

int main() {
  ifstream in("Scopy.cpp"); // Otvaranje za učitavanje
  ofstream out("Scopy2.cpp"); // Otvaranje za upisivanje
  string s;
  while(getline(in, s)) // Zanemaruje znak za novi red
    out << s << "\n"; // ... mora ga dodati
} ///:~
```

Da bi se datoteke otvorile, imena datoteka se prosleđuju objektima tipa **ifstream** i **ofstream**, kao što se vidi iz primera.

U prethodnom primeru je uveden nov pojam – petlja **while**. Iako će biti detaljno objašnjena u narednom poglavlju, sada ćemo je ukratko opisati. Zamisao je da izraz u zagradama posle reči **while** upravlja izvršavanjem sledeće naredbe (to može biti i veći broj naredaba, unutar vitičastih zagrada). Sve dok izraz u zagradama, u ovom slučaju to je **getline(in, s)**, ima vrednost true, izvršavaće se naredba kojom upravlja **while**. To znači da će **getline()** vratiti vrednost koja se tumači kao true ako je red uspešno učitan, a false kada se dođe do kraja ulazne datoteke. Na taj način petlja **while** učitava svaki red ulazne datoteke i šalje red u izlaznu datoteku.

Funkcija **getline()** učitava znakove sve dok ne naiđe na znak za novi red (završni znak se može promeniti, ali se time nećemo baviti do poglavlja o ulazno-izlaznim tokovima u tomu 2). Međutim, funkcija zanemaruje znak za novi red i ne upisuje ga u rezultujući objekat tipa **string**. Ako hoćemo da kopirana datoteka izgleda upravo kao izvorna, moramo dodati znak za novi red, kao što je i pokazano.

Sledi zanimljiv primer učitavanje cele datoteke u jedan objekat tipa **string**:

```
//: C02:FillString.cpp
// Učitavanje cele datoteke u jedan znakovni niz
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream in("FillString.cpp");
  string s, line;
  while(getline(in, line))
    s += line + "\n";
  cout << s;
} ///:~
```

S obzirom na dinamičku prirodu klase **string**, ne morate voditi računa o veličini prostora koji treba rezervisati za objekat tipa **string** – samo dodajte podatke, a znakovni niz će se širiti da bi prihvatio sve što upisujete.

Klasa **string** ima mnoge funkcije za pretraživanje i obradu, koje omogućavaju izmenu datoteke kao znakovnog niza, a to je jedna od prednosti kopiranja cele datoteke u objekat tipa **string**. Međutim, postoje i ograničenja. Često se datoteke lakše obrađuju kao skupovi redova, a ne veliki tekstualni objekti. Na primer, ako hoćete da numerišete redove, jednostavnije je da svaki red bude predstavljen posebnim objektom **string**. Da bi se problem rešio, potreban je drugačiji pristup.

Klasa vector

Objekat tipa **string** možemo popuniti ne znajući unapred koliko nam treba prostora. Problem pri učitavanju redova iz datoteke u pojedinačne objekte tipa **string** čini to što na početku ne znate koliko će vam znakovnih nizova trebati, nego saznajete tek pošto učitate celu datoteku. Da bi se problem rešio, potrebno je neko skladište koje bi se automatski širilo i prihvatalo onoliko objekata tipa **string** koliko nameravamo da u njega smestimo.

Zašto bismo se ograničavali na čuvanje objekata tipa **string**? Ispostavlja se da često tokom pisanja programa ne znamo koliko će nam objekata trebati. Mnogo korisnije bi nam bilo skladište za objekte (u programerskom žargonu se češće zove kontejner) koje bi moglo da sadrži *bilo koju vrstu objekta*! Srećom, standardna C++ biblioteka ima gotovo rešenje: standardne kontejnerske klase. Kontejnerske klase su zaista moćna alatka standarnog C++-a.

Ljudi često mešaju kontejnere i algoritme standardne C++ biblioteke i biblioteku STL. Izraz Standard Template Library (standardna biblioteka šablona) koristio je Alex Stepanov (tada je radio u Hewlett-Packardu) kada je predstavio svoju biblioteku Komitetu za C++ standarde na skupu u San Dijegu, u Kaliforniji, u proleće 1994. Naziv je prihvaćen, naročiti pošto je HP odlučio da dozvoli javno preuzimanje s Interneta. U međuvremenu je komitet integrisao biblioteku u standardnu C++ biblioteku i izmenio je. Razvoj STL-a je nastavljen u kompaniji Silicon Graphics (SGI; videti *http://www.sgi.com/Technology/STL*). SGI STL se razlikuje od standardne C++ biblioteke po mnogim detaljima. Popularna je zabluda da C++ standard „uključuje“ STL. To može malo da zbunjuje pošto kontejneri i algoritmi standardne C++ biblioteke imaju isti koren (često i ista imena) kao SGI STL. U ovoj knjizi ću koristiti nazive „standardna C++ biblioteka“ ili „standardni kontejneri biblioteke“ ili nešto slično, a izbegavaću izraz „STL“.

Iako realizacija standardnih kontejnera i algoritama C++ biblioteke koristi napredne pojmove i zauzima dva velika poglavlja drugog toma ove knjige, ova biblioteka može da se koristi i bez detaljnog poznavanja. Zato se klasa **vector**, najosnovniji od standardnih kontejnera, uvodi u ovom početnom poglavlju i koristi kroz celu knjigu. Otkrićete da mnogo toga možete uraditi primenjujući osnovne mogućnosti kontejnera **vector** i ne vodeći računa o njegovoj realizaciji

(da ponovimo, to je važan cilj OOP-a). Pošto ćete o ovom i drugim kontejnerima naučiti više u tomu 2, u poglavljima posvećenim standardnoj biblioteci, sasvim je u redu da se klasa **vector** u početnim poglavljima knjige koristi drugačije nego što bi to radio iskusni C++ programer. Otkrićete da je ovakva upotreba adekvatna u najvećem broju slučajeva.

Klasa **vector** je *šablon* (engl. *template*), što znači da se može efikasno primeniti na različite tipove. Tako se može napraviti vektor figura, vektor mačaka, vektor objekata tipa **string** itd. Pomoću šablona, može se napraviti „klasa bilo čega". Da bi se saopštilo prevodiocu sa čim će klasa raditi (u ovom slučaju, šta će **vector** sadržati), ime tipa se smešta između znakova < i >. Tako će **vector** za klasu **string** biti označen sa **vector<string>**. Kada ovo napišete, dobijate prilagođeni vektor koji će sadržati samo objekte klase **string** i prevodilac će prijaviti grešku ako u vektor pokušate da smestite objekat nekog drugog tipa.

Pošto je **vector** skladište za objekte, tj. kontejner, mora postojati način da nešto stavite u kontejner i uzmete iz njega. Da bi se na kraj kontejnera dodao nov element, koristi se funkcija članica **push_back()**. (Prisetimo se da koristimo '**.**' da bismo pozvali funkciju članicu nekog objekta.) Ime ove funkcije članice može delovati preopširno – **push_back()** umesto jednostavnijeg naziva „put" – zato što postoje i drugi kontejneri i druge funkcije članice za smeštanje novih elemenata. Na primer, postoji funkcija članica **insert()** za smeštanje elemenata na proizvoljno mesto u kontejneru. Klasa **vector** ovo podržava, ali je korišćenje složenije i nije potrebno da ga proučavamo do toma 2 ove knjige. Funkcija **push_front()** (nema je u klasi **vector**) smešta elemente na početak. Postoje i mnoge druge funkcije članice klase **vector** i mnogi drugi kontejneri standardne C++ biblioteke, ali ćete se iznenaditi koliko se može učiniti iako se zna samo mali broj jednostavnih osobina.

Ako nove elemente dodajete u **vector** funkcijom **push_back()**, kako ih uzimate odatle? Rešenje je jednostavno i veoma elegantno – koristi se preklapanje operatora da bi **vector** izgledao kao *niz* (engl. *array*). Niz (biće detaljnije opisan u sledećem poglavlju) jeste tip podataka koji postoji u skoro svim programskim jezicima, tako da bi trebalo da vam već bude poznat. Nizovi su *agregati*, što znači da se sastoje od izvesnog broja grupisanih elemenata. Posebna osobina niza je što su ovi elementi iste veličine i uređeni su tako da slede jedan za drugim. Najvažnije je da se ovi elementi mogu izabrati „indeksiranjem", što znači da možete reći. „Treba mi element broj n" i uglavnom ćete brzo dobiti taj element. Mada postoje izuzeci u programskim jezicima, indeksiranje se obično postiže korišćenjem uglastih zagrada. Ako imate niz **a** i treba vam peti element, navodite **a[4]** (indeksiranje počinje od nule).

Ova kompaktna i moćna notacija indeksiranja uključena je u **vector** preklapanjem operatora, na isti način kao što su << i >> uključeni u ulazno-izlazne tokove. Ni sada ne treba da znate kako je realizovano preklapanje – to je ostavljeno za neko kasnije poglavlje – ali je korisno saznanje da postoji neki tajanstveni mehanizam koji omogućava da **vector** koristi operator [ ].

Imajući sve ovo na umu, pogledajte program koji koristi **vector**. Da bi se koristio **vector**, uključuje se datoteka zaglavlja **<vector>**:

```
//: C02:Fillvector.cpp
// Kopiranje kompletne datoteke u vektor objekata tipa string
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
  vector<string> v;
  ifstream in("Fillvector.cpp");
  string line;
  while(getline(in, line))
    v.push_back(line); // Dodavanje reda na kraj
  // Dodavanje brojeva redova:
  for(int i = 0; i < v.size(); i++)
    cout << i << ": " << v[i] << endl;
} ///:~
```

Najveći deo programa sličan je prethodnom: datoteka se otvara i jedan po jedan red se učitava u objekte tipa **string**. Ovi objekti tipa **string** smeštaju se na kraj vektora **v**. Po završetku petlje **while**, cela datoteka se nalazi u memoriji, u vektoru **v**.

Sledeća naredba programa je petlja **for**. Ona je slična petlji **while**, ali se tom petljom drugačije upravlja. Iza **for** sledi „kontrolni izraz“ između zagrada, slično kao kod petlje **while**. Taj kontrolni izraz ima tri dela: inicijalizacioni deo, deo koji ispituje uslov za izlazak iz petlje i deo koji nešto menja, kako bi se prošlo kroz sekvencu elemenata. Ovaj program sadrži petlju **for** u obliku u kome se najčešće koristi: inicijalizacioni deo **int i = 0** definiše ceo broj **i** kako bi ga koristio kao brojač petlje i dodeljuje mu početnu vrednost nula. Deo za ispitivanje tvrdi da se ostaje u petlji sve dok je **i** manje od broja elemenata vektora **v**. (To dobijamo upotrebom funkcije članice **size()**, koju sam neprimetno ubacio, ali morate priznati da je njeno značenje očigledno.) Poslednji deo koristi skraćenicu C-a i C++-a, operator za „automatsko uvećavanje“, koji vrednost **i** uvećava za jedan. U suštini, **i++** znači da treba uvećati vrednost **i** za jedan i rezultat dodeliti promenljivoj **i**. Znači da je ukupni učinak petlje **for** da promenljivoj **i** dodeljuje redom vrednosti od nule do broja za jedan manjeg od veličine vektora. Za svaku vrednost **i** ispisuje se red u kome su vrednost **i** (na tajanstveni način konvertuje se u niz znakova), dve tačke i razmak, red iz datoteke i znak za novi red dobijen pomoću **endl**. Kada prevedete i izvršite program, videćete da je rezultat numerisanje redova datoteke.

S obzirom na način rada operatora >> sa ulazno-izlaznim tokovima, prethodni program možete jednostavno izmeniti tako da se ulazni tok deli na reči razdvojene razmacima umesto na redove:

```
//: C02:GetWords.cpp
// Podela datoteke na reči razdvojene razmacima
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
  vector<string> words;
  ifstream in("GetWords.cpp");
  string word;
  while(in >> word)
    words.push_back(word);
  for(int i = 0; i < words.size(); i++)
    cout << words[i] << endl;
} ///:~
```

Izraz

```
while(in >> word)
```

uzima sa ulaznog toka jednu po jednu reč (engl. *word*) i kada izraz dobije vrednost false znači da je dostignut kraj datoteke. Razdvajanje reči razmacima je svakako grub primer, ali je zato jednostavan. U nastavku knjige videćete bolje primere koji omogućavaju podelu ulaznog toka na bilo koji način.

Da bi se pokazala jednostavnost korišćenja klase **vector** s bilo kojim tipom, u sledećem primeru se definiše **vector<int>**:

```
//: C02:Intvector.cpp
// Definisanje vektora koji sadrži cele brojeve
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> v;
  for(int i = 0; i < 10; i++)
    v.push_back(i);
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;
  for(int i = 0; i < v.size(); i++)
    v[i] = v[i] * 10; // Dodeljivanje
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;
} ///:~
```

Da bi se napravio vektor koji sadrži neki drugi tip, navedite taj tip kao argument šablona (argument se navodi između znakova < i >). Šabloni i dobro definisane biblioteke šablona upravo su namenjeni tako jednostavnoj upotrebi.

Ovaj primer prikazuje još jednu značajnu osobinu klase **vector**. U izrazu

```
v[i] = v[i] * 10;
```

vidite da **vector** nije ograničen na smeštaj i uzimanje elemenata. Možete i *da dodelite* vrednost bilo kom elementu vektora (samim tim ga i izmenite), tako što ćete upotrebiti uglaste zagrade koje predstavljaju operator indeksiranja. To znači da je **vector** „privremeni smeštajni prostor“ opšte namene za rad sa zbirkama objekata i mi ćemo ga svakako koristiti u narednim poglavljima.

Sažetak

Namera je da se u ovom poglavlju pokaže koliko jednostavno može biti objektno orijentisano programiranje *ako* neko drugi prione na posao i definiše objekte. U tom slučaju treba samo da uključite datoteku zaglavlja, napravite objekte i šaljete im poruke. Ako su tipovi koje koristite dovoljno funkcionalni i dobro projektovani, nećete imati puno posla i program će biti funkcionalan.

Prilikom upotrebe bibliotečkih klasa, u ovom poglavlju su uvedeni neki od osnovnih i korisnih tipova standardne C++ biblioteke: porodica ulazno-izlaznih tokova (konkretno, onih za čitanje i upis na konzolu i u datoteke), klasa **string** i šablon **vector**. Videli ste da se jednostavno primenjuju i sada verovatno možete zamisliti šta sve pomoću njih možete uraditi, ali one imaju i mnoge druge mogućnosti.[5] Iako ćemo u početnim poglavljima knjige koristiti samo ograničen podskup mogućnosti ovih alatki, to je ipak veliki korak napred u odnosu na elementarno učenje jezika niskog nivoa kakav je C. Mada je učenje aspekata niskog nivoa jezika C korisno, ono iziskuje puno vremena. Na kraju, glavna *svrha* OOP-a je da prikrije detalje, tako da se možete baviti važnijim problemima.

Bez obzira na apstraktnost OOP-a, postoje nezaobilazni osnovni pojmovi C-a, a njih ću obraditi u narednom poglavlju.

Vežbe

Rešenja izabranih zadataka nalaze se u elektronskom dokumentu *The Thinking in C++ Annotated Solution Guide* koji se, uz malu nadoknadu, može preuzeti s lokacije www.BruceEckel.com

1. Izmenite **Hello.cpp** tako da ispisuje vaše ime i broj godina (ili broj cipela ili starost vašeg psa, ako vam je to lakše). Prevedite i izvršite program.
2. Koristeći ideje programa **Stream2.cpp** i **Numconv.cpp** napišite program koji učitava poluprečnik kruga i izračunava i ispisuje površinu kruga. Možete upotrebiti operator * za izračunavanje kvadrata poluprečnika. Ne ispisujte vrednost oktalno ili heksadecimalno (ovi formati se koriste samo za cele brojeve).
3. Napišite program koji otvara datoteku i u njoj prebrojava reči razdvojene razmacima.
4. Napišite program koji prebrojava pojavljivanja određene reči u datoteci (koristite operator == klase **string** da biste prepoznali reč).

[5] Ako jedva čekate da odmah vidite sve što se može učiniti pomoću ovih i drugih komponenata biblioteke, pogledajte drugi tom ove knjige, koji možete preuzeti s adrese www.BruceEckel.com, odnosno www.dinkumware.com.

5. Izmenite **Fillvector.cpp** tako da ispisuje redove unazad, od poslednjeg do prvog.
6. Izmenite **Fillvector.cpp** tako da spaja sve elemente vektora u jedan znakovni niz pre ispisivanja, ali nemojte numerisati redove.
7. Prikažite datoteku red po red, čekajući da korisnik pritisne taster „Enter“ posle svakog reda.
8. Definišite **vector<float>** i u njega smestite 25 brojeva u formatu pokretnog zareza koristeći petlju **for**. Prikažite sadržaj vektora.
9. Definišite tri objekta tipa **vector<float>** i popunite prva dva kao u prethodnom zadatku. Napišite petlju **for** koja sabira odgovarajuće elemente prva dva objekta tipa **vector** i smešta rezultat u odgovarajući element trećeg objekta. Prikažite sva tri objekta tipa **vector**.
10. Definišite objekat tipa **vector<float>** i smestite u njega 25 brojeva kao u prethodnim zadacima. Zatim izračunajte kvadrat svakog broja i rezultat dodelite istom elementu vektora. Prikažite sadržaj vektora pre i posle množenja.