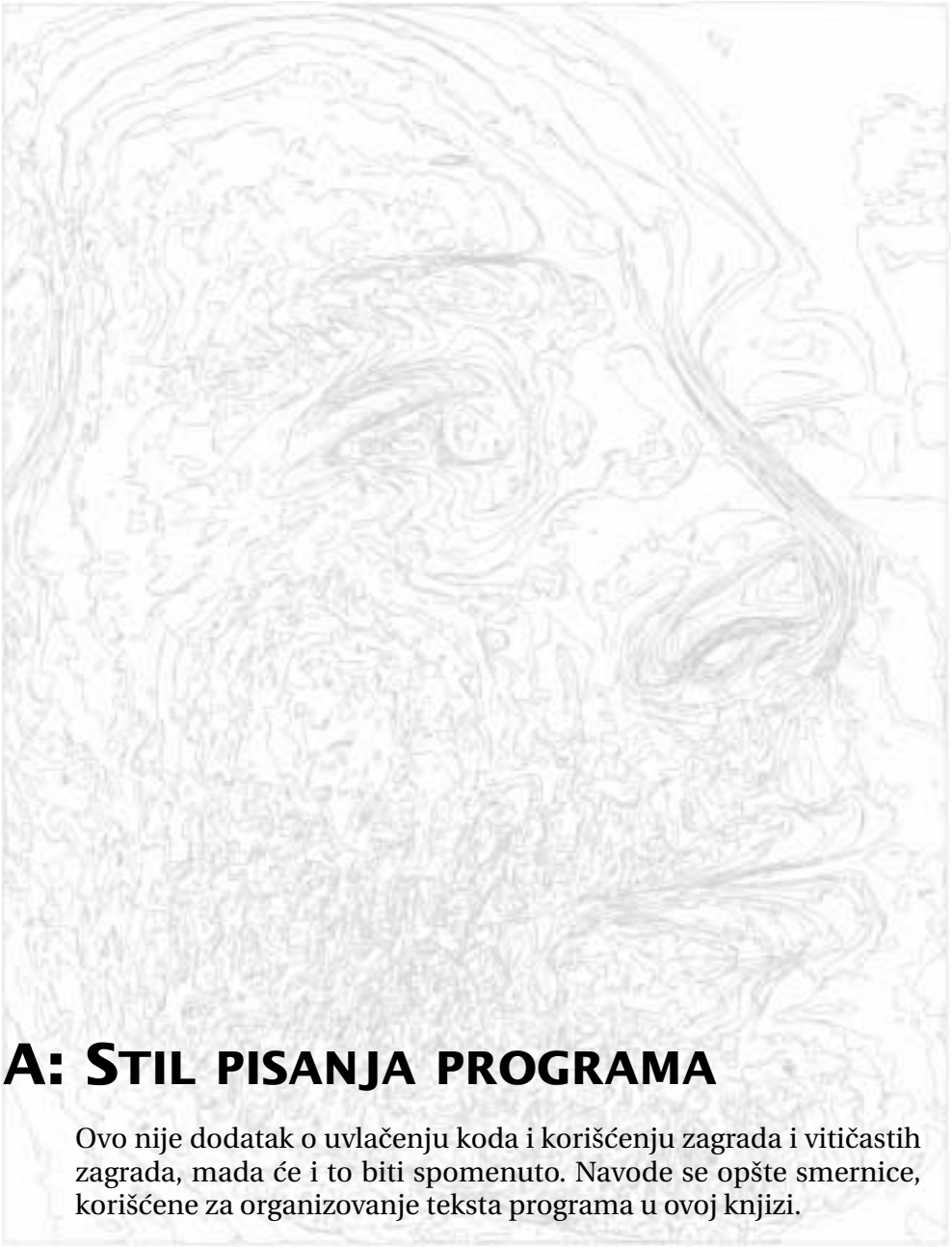

A: Stil pisanja programa

Ovo nije dodatak o uvlačenju koda i korišćenju zagrada i vitičastih zagrada, mada će i to biti spomenuto. Navode se opšte smernice, korišćene za organizovanje teksta programa u ovoj knjizi.

Iako su mnoge teme o kojima će biti reči već razmotrene u knjizi, ovaj dodatak se pojavljuje na kraju, tako da zaokružuje celinu. Ako nešto ne razumete, možete to potražiti u odgovarajućem odeljku.

Sve odluke o načinu pisanja programa u ovoj knjizi pažljivo su promišljane i donošene, ponekad i tokom niza godina. Naravno, svako ima svoje razloge za pisanje programa na određeni način, pa ću vam preneti kako sam došao do svojih odluka i koja su me ograničenja i uslovi okruženja do njih doveli.

Opšta pravila

U tekstu knjige su identifikatori (imena funkcija, promenljivih i klasa) napisani **polucrnim** slovima. Većina rezervisanih reči je takođe napisana polucrno.

U primerima koristim određeni način pisanja. On je razvijan tokom više godina i delimično je inspirisan stilom Bjarnea Stroustrupa iz njegove knjige *Programski jezik C++*. O formatiranju se može voditi višesatna vruća rasprava, tako da preko primera ne pokušavam da nametnem način koji smatram jedino ispravnim; imam svoje razloge za korišćenje prikazanog stila. Pošto je C++ programski jezik slobodnog oblika, možete nastaviti da pišete na način koji vam odgovara.

Napomenuću da je važno dosledno oblikovati program u okviru jednog projekta. Ako pretražite Internet, naći ćete veći broj alatki koje možete iskoristiti za preoblikovanje celokupnog koda u svom projektu, kako biste postigli ovu dragocenu doslednost.

Programi, navedeni u ovoj knjizi, jesu datoteke koje se automatski izdvajaju iz teksta knjige, što omogućava da se testira njihova ispravnost. Zato bi trebalo da svi programi u knjizi rade bez grešaka pri prevođenju, ako realizacija prevodioca odgovara standardu jezika C++ (imajte u vidu da ne podržava svaki prevodilac sve mogućnosti). Greške koje *treba* da prouzrokuju poruke pri prevođenju stavljene su pod komentar oznakom **//!**, jednostavno se otkrivaju i mogu se automatski testirati. Greške koje čitaoci otkrivaju i prijavljuju autoru, prvo će se pojaviti u elektronskoj verziji ove knjige (na Web lokaciji *www.BruceEckel.com*), a zatim i u izmenjenim izdanjima.

Jedno od pravila u ovoj knjizi je da se svi programi prevode i povezuju bez grešaka (iako će ponekad prouzrokovati upozorenja). Radi toga će neki programi, koji služe samo kao primeri, a ne predstavljaju samostalne programe, imati praznu funkciju **main( )**:

```
int main() {}
```

To omogućava povezivanje bez greške.

Pravilo je da funkcija **main( )** vraća vrednost tipa **int**, ali standard C++-a ustanovljava da, ako ne postoji iskaz **return** unutar funkcije **main( )**, prevodilac automatski generiše kôd za **return 0**. Ova opcija (bez iskaza **return** u funkciji **main( )**) koristitiće se u knjizi (neki prevodioci i dalje mogu ispisivati upozorenja, ali oni nisu usklađeni sa standardom jezika C++).

Imena datoteka

U jeziku C tradicionalno se datotekama zaglavlja (sadrže deklaracije) dodeljuju imena s nastavkom **.h**, a datotekama realizacije (dovode do zauzimanja memorijskog prostora i generisanja koda) imena s nastavkom **.c**. Jezik C++ se postepeno razvijao. Prvo je razvijen pod Unixom, gde su za operativni sistem značajna velika i mala slova u imenima datoteka. Prvobitno su imena datoteka imala nastavke kao u C-u, ali napisane velikim slovima: **.H** i **.C**. To nije funkcionisalo za operativnim sistemima koji nisu pravili razliku između malih i velikih slova, kao što je DOS. Proizvođači C++-a za DOS koristili su nastavke **hxx** i **cxx** za datoteke zaglavlja i datoteke realizacije, ili **hpp** i **cpp**. Kasnije je neko otkrio da različiti nastavci samo postoje da bi prevodilac mogao odrediti da li da datoteku prevodi kao C ili C++. Pošto prevodilac nikada nije neposredno prevodio datoteke zaglavlja, trebalo je promeniti samo nastavak imena datoteke realizacije. Sada je uobičajeno, u skoro svim sistemima, da se za datoteke realizacije koristi **cpp**, a za datoteke zaglavlja **h**. Zapazite da se standardne datoteke zaglavlja C++-a, mogu uključiti i ako se izostavi nastavak imena, na primer: **#include <iostream>**.

Oznake za početak i kraj komentara

Veoma značajan zahtev autora u vezi s ovom knjigom jeste da se mora proveriti ispravnost koda (bar jednim prevodiocem). To se postiže automatskim izdvajanjem datoteka iz knjige. Da bi se postupak olakšao, svi tekstovi programa koje treba prevesti (za razliku od malobrojnih delova koda) imaju oznake komentara na početku i na kraju. Ove oznake koristi program za izdvajanje koda **ExtractCode.cpp** u drugom tomu, kako bi izvukao tekstove programa iz ASCII verzije teksta knjige.

Oznaka za kraj programa saopštava programu **ExtractCode.cpp** da je to kraj teksta programa. Iza oznake za početak programa nalazi se informacija o poddirektorijumu kome pripada datoteka (raspoređene su po poglavljima, pa bi datoteka koja pripada poglavlju – engl. *chapter* – 8 imala oznaku **C08**), praćenu znakom dve tačke i imenom datoteke programa.

Pošto program **ExtractCode.cpp** pravi i datoteku **makefile** za svaki poddirektorijum, u tekstove programa su uključene i informacije o pravljenju programa i komandni red za testiranje. Ako je program samostalan (nije potrebno povezivanje s nekim drugim programom), ne postoje dodatne informacije. Isto važi i za datoteke zaglavlja. Međutim, ako datoteka ne sadrži funkciju **main( )** i podrazumeva se povezivanje s nekim drugim programom, tada se iza imena datoteke nalazi **{O}**. Ako tekst programa predstavlja glavni program, ali je potrebno povezivanje s drugim komponentama, tada postoji poseban red koji počinje oznakom **//{L}** i sadrži imena svih datoteka koje treba povezati (bez nastavaka, pošto oni zavise od računarskog sistema).

Primere ćete pronaći na više mesta u knjizi.

Ako treba izdvojiti datoteku, ali oznake početka i kraja teksta programa ne treba uključiti u izdvojenu datoteku (na primer, ako datoteka sadrži podatke za testiranje), tada se neposredno iza oznake početka nalazi '**!**'.

Zagrade, vitičaste zagrade i uvlačenje

Primetili ste da se način pisanja programa u ovoj knjizi razlikuje od mnogih tradicionalnih načina programiranja na jeziku C. Podrazumeva se da svako smatra svoj način najlogičnijim. Način koji sam koristio zasnovan je na jednostavnoj logici, koja će biti opisana u kombinaciji sa razlozima razvijanja nekih drugih načina.

Tekst programa se oblikuje prema načinu prezentacije, u štampanom obliku i na predavanjima. Možda smatrate da su vaše potrebe drugačije, pošto ne pravite veliki broj prezentacija. Međutim, tekst programa se mnogo češće čita nego što se piše i trebalo bi da čitaocu bude lako shvatljiv. Moja dva najznačajnija kriterijuma su preglednost (koliko je čitaocu lako da usvoji značenje jednog reda) i broj redova koji mogu stati na jednu stranu. Drugi kriterijum može delovati smešno, ali pri prezentaciji uživo, publiku dekoncentriše kad izlagač mora da pomera slajdove unapred i unazad, zbog samo nekoliko suvišnih redova.

Svi se slažu da kôd unutar vitičastih zagrada treba da bude uvučen. Najviše nesuglasica i nedoslednosti javlja se pri donošenju odluke: gde napisati otvorenu vitičastu zagradu. Ovo pitanje, po mom mišljenju, dovodi do mnogobrojnih različitih načina programiranja (opise raznih stilova možete potražiti u knjizi Toma Pluma i Dana Saksa *C++ Programming Guidelines*, Plum Hall 1991). Pokušaću da vas ubedim da mnogi današnji načini potiču od ograničenja jezika C pre standardizacije (pre prototipa funkcija) i da zato više nisu pogodni.

Prvo ću odgovoriti na to ključno pitanje: otvorena vitičasta zagrada uvek treba da bude u istom redu, kao „prethodnica" (dalje podrazumevam „onoga što telo predstavlja: klasu, funkciju, definiciju objekta, uslovni iskaz itd."). To pravilo dosledno primenjujem na sve tekstove programa koje pišem i ono pojednostavljuje oblikovanje koda. Navedeno pravilo poboljšava „preglednost" – na primer, kada pogledate ovaj red:

```
int func(int a);
```

po tački i zarezu na kraju reda znate da je to deklaracija i da se ne nastavlja, ali kada vidite red:

```
int func(int a) {
```

odmah znate da je to definicija, pošto se red završava otvorenom vitičastom zagradom, a ne tačkom i zarezom. Ako koristite ovaj pristup, nema razlike između položaja otvorene zagrade u višerednoj definiciji:

```
int func(int a) {
  int b = a + 1;
  return b * 2;
}
```

i definiciji u jednom redu, koja se često koristi za umetnute funkcije:

```
int func(int a) { return (a + 1) * 2; }
```

Slično važi i za klasu:

```
class Klasa;
```

jeste deklaracija imena klase, a definicija klase je:

```
class Klasa {
```

U svakom slučaju, ako pogledate samo jedan red, već ćete razlikovati deklaraciju od definicije. Osim toga, ako otvorenu vitičastu zagradu ne pišete u posebnom redu, više iskaza staje na jednu stranu.

Zašto onda postoji toliko drugih načina pisanja koda? Zapazićete da većina ljudi pravi klase koristeći navedeni način (Stroustrup ga koristi u svim izdanjima svoje knjige *Programski jezik C++*, izdavač Addison-Wesley), ali piše definicije funkcija smeštajući otvorenu vitičastu zagradu u poseban red (to prouzrokuje i više različitih načina uvlačenja redova programa), osim u slučaju kratkih umetnutih funkcija. Uz pristup koji sam ovde opisao, sve je dosledno: imenujete ono što definišete (klasu, funkciju, nabrojivi tip itd.) i u isti red smeštate otvorenu vitičastu zagradu da biste ukazali da sledi telo definicije. Osim toga, otvorena zagrada je na istom mestu za kratke umetnute funkcije i obične definicije funkcija.

Tvrdim da ovaj način definisanja funkcija, koji mnogi koriste, potiče iz C-a pre uvođenja prototipa funkcija, kada se argumenti nisu deklarisali unutar zagrada, nego između zatvorene zagrade i otvorene vitičaste zagrade (to pokazuje da su koreni C-a u asemblerskom jeziku):

```
void bar()
  int x;
  float y;
{
  /* telo funkcije */
}
```

Ovde bi bilo prilično nezgodno pisati otvorenu vitičastu zagradu u istom redu, pa to niko nije ni radio. Postojala su različita mišljenja o tome da li vitičaste zagrade treba uvlačiti s telom koda ili one treba da budu na nivou „prethodnice“. Tako smo dobili više različitih načina oblikovanja teksta programa.

Postoje i drugi argumenti za postavljanje vitičaste zagrade u red koji neposredno sledi iza deklaracije (klase, strukture, funkcije itd.). Navodim tekst jednog čitaoca kako biste znali šta su sporna pitanja:

> Iskusni korisnici programa ‘vi’ (vim) znaju da će dva pritiska na taster ‘]’ dovesti korisnika do sledećeg pojavljivanja ‘{‘ (ili ^L) u koloni 0. Ova mogućnost je izuzetno korisna pri kretanju kroz program (skok na sledeću definiciju funkcije ili klase). [Moj komentar: kada sam prvobitno radio pod Unixom, GNU Emacs se upravo pojavljivao i ja sam se upecao. Posledica je da ‘vi’ meni nikada ništa nije značio i zato ne razmišljam o „lokaciji kolone 0“. Međutim, prilično je velik broj korisnika programa ‘vi’ i ova tema je za njih značajna.]

Pisanjem '{' u sledećem redu uklanja se zbunjujući kôd u slučaju složenih uslovnih iskaza i poboljšava se preglednost. Primer:

```
if(cond1
   && cond2
   && cond3) {
   statement;
}
```

Navedeni kôd [tvrdi čitalac] nije pregledan. Međutim:

```
if (cond1
&& cond2
&& cond3)
{
statement;
}
```

odvaja 'if' od tela i poboljšava čitljivost. [Mišljenja o tome da li je to tačno razlikovaće se, u zavisnosti od navika.]

Najzad, mnogo je lakše vizuelno poravnati vitičaste zagrade ako se nalaze u istoj koloni. One su mnogo bolje vizuelno „izdvojene". [Kraj komentara čitaoca]

O mestu otvorene vitičaste zagrade verovatno postoji najviše nesuglasica. Naučio sam da gledam oba načina pisanja i na kraju postaje nevažno na koji ste oblik navikli. Zapazio sam da je zvanični standard pisanja programa na Javi (objavljen na Sunovoj Web stranici posvećenoj Javi) isti kao onaj koji vam ovde prikazujem – pošto sve više ljudi počinje da programira i na C++-u i na Javi, korisno je da se na oba jezika programi pišu istim stilom.

Pristup koji koristim uklanja sve izuzetke i specijalne slučajeve, a logično sledi i jedinstven stil uvlačenja. Doslednost se zadržava i unutar tela funkcije, kao u:

```
for(int i = 0; i < 100; i++) {
  cout << i << endl;
  cout << x * i << endl;
}
```

Ovaj stil je jednostavno preneti drugima i zapamtiti – uvek dosledno koristite jedno pravilo, tako da ne postoji jedno pravilo za klase, dva za funkcije (za umetnute funkcije u jednom redu i za funkcije definisane u više redova), a možda i druga pravila za petlje **for**, iskaze **if** itd. Sama doslednost čini ovo pravilo vrednim razmatranja. Jezik C++ je noviji od jezika C i, iako moramo praviti mnoge ustupke C-u, ne bi trebalo da zadržimo navike koje će nam stvarati probleme u budućnosti. Mali problemi pomnoženi sa velikim brojem redova koda postaju veliki problemi. Ova tema se sveobuhvatno obrađuje u knjizi *C Style: Standards and Guidelines* Davida Strakera (Prentice-Hall 1992).

Drugo ograničenje stila pisanja jeste širina reda, pošto je za knjigu postavljena granica od 50 znakova. Šta se dešava ako je tekst preširok da bi se uklopio

u jedan red? Ponovo nastojim da dosledno primenjujem pravilo preloma redova, tako da se mogu lako pregledati. Sve dok je nešto deo jedne definicije, liste argumenata itd., naredni redovi treba da budu za jedan nivo uvučeni u odnosu na početak te definicije, liste argumenata itd.

Imena identifikatora

Oni kojima je bliska Java, zapaziće da sam za pisanje imena identifikatora primenjivao standardni način Jave. Nisam bio, potpuno dosledan, pošto se taj način ne koristi u standardnim bibliotekama jezika C i C++.

Način pisanja je sasvim jednostavan. Samo identifikatori klasa počinju velikim slovom. Malim slovom počinju funkcije ili promenljive. Ostatak identifikatora se sastoji od jedne ili više spojenih reči i prvo slovo svake reči je veliko. Klasa izgleda ovako:

```
class FrenchVanilla : public IceCream {
```

identifikator objekta izgleda ovako:

```
FrenchVanilla myIceCreamCone(3);
```

a funkcija izgleda ovako:

```
void eatIceCreamCone();
```

(važi i za funkcije članice i za ostale funkcije).

Izuzetak su konstante i simboli (**const** i **#define**), čiji identifikatori se pišu velikim slovima.

Ovo pravilo je značajno jer na osnovu prvog slova vidite da li se radi o klasi, objektu ili metodi. To je posebno korisno kada pristupate statičkim članovima klase.

Redosled uključivanja datoteka zaglavlja

Datoteke zaglavlja se uključuju redosledom „od najposebnije do najopštije“. Prvo uključujem datoteke zaglavlja u lokalnom direktorijumu, zatim datoteke zaglavlja svojih „alatki“, kao što je **require.h**, zatim datoteke zaglavlja biblioteka drugih proizvođača, pa datoteke zaglavlja standardne biblioteke C++-a i na kraju datoteke zaglavlja C biblioteka.

Ovaj pristup je potkrepljen izvodom iz knjige *Large-Scale C++ Software Design*, Johna Lakosa, Addison-Wesley, 1996:

> *Skrivene greške se mogu izbeći ako se obezbedi da se .h datoteka komponente pojedinačno analizira – bez spoljnih deklaracija ili definicija... Uključivanjem .h datoteke u prvom redu .c datoteke, obezbeđuje se da delovi informacija, značajni za fizički interfejs komponente, neće nedostajati u .h datoteci (ako nedostaju, to ćete otkriti pri pokušaju prevođenja .c datoteke).*

Ako je redosled uključivanja datoteka zaglavlja „od najposebnije do najopštije“, verovatno ćete na vreme otkriti nekompletne datoteke zaglavlja i spečićete naknadne neprilike.

Zaštita od uključivanja u datotekama zaglavlja

Zaštita od uključivanja (engl. *include guards*) uvek se koristi u datotekama zaglavlja, da bi se sprečilo višekratno uključivanje iste datoteke zaglavlja tokom prevođenja jedne **.cpp** datoteke. Prevođenje se kontroliše korišćenjem pretprocesorske naredbe **#define** i proverom da li je ime već definisano. Ime identifikatora, koji se koristi za zaštitu od uključivanja, pravi se na osnovu imena datoteke zaglavlja. Sva slova su velika, a tačka se zamenjuje znakom za podvlačenje. Na primer:

```
// IncludeGuard.h
#ifndef INCLUDEGUARD_H
#define INCLUDEGUARD_H
// telo datoteke zaglavlja
#endif // INCLUDEGUARD_H
```

Identifikator u poslednjem redu naveden je radi jasnoće. Iako neki pretprocesori zanemaruju sve znakove iza naredbe **#endif**, to nije standardno ponašanje i zato je identifikator ispod komentara.

Korišćenje imenskih prostora

U datoteci zaglavlja se pažljivo mora izbegavati bilo kakvo „zagađivanje“ imenskog prostora u koji je uključena datoteka zaglavlja. Ako promenite imenski prostor u kome se nalazi funkcija, odnosno klasa, prouzrokovaćete tu promenu u svakoj datoteci koja uključuje datoteku zaglavlja, što izaziva raznovrsne probleme. Deklaracije **using** nisu dozvoljene izvan definicija funkcija, a globalni iskazi **using** nisu dozvoljeni u datotekama zaglavlja.

U **cpp** datoteci, svi globalni iskazi **using** uticaće samo na tu datoteku i zato se oni koriste u ovoj knjizi, kôd malih programa postaje čitkiji.

Korišćenje funkcija require( ) i assure( )

Funkcije **require( )** i **assure( )**, definisane u **require.h**, dosledno se koriste u većem delu knjige, tako da pravilno prijavljuju probleme. Ako su vam bliski *preduslovi* (engl. *preconditions*) i *izlazni uslovi* (engl. *postconditions*), koje je uveo Bertrand Meyer, prepoznaćete da se upotrebom funkcija **require( )** i **assure( )** u izvesnoj meri obezbeđuju preduslovi (obično) i izlazni uslovi (povremeno). Na početku funkcije, pre početka izvršavanja „jezgra“, proveravaju se preduslovi, kako bismo se uverili da je sve ispravno i da su ispunjeni svi potrebni uslovi. Zatim se izvršava „jezgro“ funkcije i ponekad se proveravaju neki izlazni uslovi, da bismo se uverili da je novo stanje podataka odgovarajuće. Zapazićete da se izlazni uslovi retko proveravaju u primerima iz knjige, a funkcija **assure( )** je uglavnom korišćena za proveru da li su sve datoteke uspešno otvorene.